Self-Supervised Learning

From Foundations to Research

A Comprehensive Course Companion and Research Guide

Based on the course *Self-supervised Learning*
taught by Hermann Blum
Robot Perception and Learning Lab, University of Bonn / Lamarr Institute

Compiled as an expanded, self-contained study text

2026

Contents

Preface xii

Course-to-Book Mapping xv

Notation xvii

I Foundations 1

1 Why Self-Supervised Learning? 2

1.1 Supervised and self-supervised: one loop, one difference 4
1.2 The cost of labels 6
1.3 What a representation is 9
1.4 Pretext tasks and downstream tasks 12
1.5 A map of the book: three families 15
1.6 Distance in embedding space: a worked example 17
1.7 How do we know it is good? The evaluation taxonomy 20
1.8 How we got here: a short prehistory 27
1.9 Chapter summary 28

2 The Mathematical and ML Toolkit 34

2.1 How to Read This Chapter 35
2.2 Neural Networks in Fifteen Minutes 36
2.3 Loss Functions 42
2.4 Similarity and Distance 46
2.5 Optimization 52
2.6 Information Theory 58
2.7 Classical Retrieval and Clustering Toolkit 62

2.8 Summary and Outlook 79

II The Course: Lecture by Lecture 83

3 Clustering and Classical SSL Hybrids 84

3.1 Nearest Neighbors: From Metric to Search 86
3.2 Clustering as a Signal Generator 95
3.3 DeepCluster: Closing the Loop 108
3.4 STEGO: Distilling Feature Correspondences into Segmentation 117
3.5 Chapter summary 134
3.6 Check Your Understanding 136

4 Autoencoders and VAEs 140

4.1 The Autoencoder Idea: Compress, Then Reconstruct 142
4.2 The Information-Bottleneck View, and Why It Is Not a Guarantee 145
4.3 Training Autoencoders: Reconstruction Loss, Depth, BatchNorm 148
4.4 The Variational Autoencoder: A Latent-Variable Model 151
4.5 The ELBO, Step by Step 155
4.6 Making the ELBO Computable 157
4.7 Why Does the VAE Actually Work? 169
4.8 Where the Lecture Goes Next: Masking Instead of Bottlenecking 170

5 Masked Autoencoders 174

5.1 The pretext task: masking as generation 176
5.2 The MAE machinery, step by step 178
5.3 The asymmetric design 186
5.4 What to reconstruct: pixels, not tokens 189
5.5 Why does this actually work? 194
5.6 What MAE is for: representation learning 195
5.7 Headline results 196
5.8 Failure modes and boundaries 199
5.9 Beyond still images 200
5.10 Check Your Understanding 200
5.11 Concept Map 203

6 Generative Models I: KDE, Flows, GANs **204**
6.1 Modeling $p(X)$ Is Self-Supervision . . . 206
6.2 Kernel Density Estimation: The Simplest Generative Model . . . 207
6.3 Normalizing Flows: Exact Likelihoods by Inversion . . . 216
6.4 Generative Adversarial Networks: From Likelihood to Game . . . 259
6.5 Taxonomy of Generative Models . . . 303
6.6 Summary . . . 303

7 Denoising, Diffusion, Flow Matching **309**
7.1 The generative-model taxonomy, revisited . . . 311
7.2 Denoising as a learning signal . . . 313
7.3 Diffusion models as learned gradual denoising . . . 319
7.4 DDPM under the hood: seven derivations the exercises rely on . . . 327
7.5 Rectified flow and flow matching, code first . . . 342
7.6 Continuous normalizing flows and the density question . . . 357
7.7 Conditioning and classifier-free guidance . . . 361
7.8 Latent diffusion: composing everything . . . 365
7.9 Autoregressive interlude: the tractable corner . . . 367
7.10 Applications back in SSL territory . . . 368
7.11 Closing the taxonomy loop . . . 371

8 Pretext Tasks and Augmentations **377**
8.1 From Generating Data to Representing It . . . 378
8.2 Augmentations: The Raw Material . . . 380
8.3 Augmentations as Inductive Bias . . . 384
8.4 The Organizing Question: Invariance versus Covariance . . . 386
8.5 Reconstruction-Based Pretext: Inpainting . . . 389
8.6 Prediction-Based Pretext Tasks . . . 392
8.7 Worked Example: RotNet . . . 398
8.8 Judging What We Learned: An Evaluation Segue . . . 403
8.9 Closing the Arc: From Predicting the Augmentation to Ignoring It . . . 406

9 Contrastive Learning **412**
9.1 From Pretext Prediction to Pretext Invariance . . . 414
9.2 Deep Roots: Siamese Networks and the Agreement Principle . . . 417

9.3 The Loss Ladder: Triplet → N-pair → NCE → InfoNCE 422
9.4 The Temperature τ on the Hypersphere 444
9.5 InfoNCE as a Mutual-Information Lower Bound 446
9.6 False Negatives and Class Collision 452
9.7 SimCLR: A Simple Framework in Nine Lines 454
9.8 MoCo: Momentum Contrast and the Dictionary as a Queue 465
9.9 Putting the Pieces Together: SimCLR vs. MoCo 472
9.10 Why Does Contrastive Learning Work? (Interpretation, Not Settled Theory) 473
9.11 Chapter summary 475
9.12 Exercises 477

10 Collapse and Non-Contrastive Learning 480
10.1 The collapse problem, stated precisely 482
10.2 Three families of collapse avoidance 486
10.3 SimSiam: stop-gradient plus a predictor, and nothing else 487
10.4 BYOL: SimSiam plus EMA — and still no negatives 498
10.5 Barlow Twins: redundancy reduction on the cross-correlation matrix 500
10.6 VICReg: variance, invariance, covariance — an explicit triad 506
10.7 Why does any of this work? Two sufficient conditions 508
10.8 Head-to-head comparison 508
10.9 Research frontier: collapse as a black box 509
10.10Summary 511

11 Self-Distillation and the DINO Family 514
11.1 Self-distillation, and what this chapter leaves to Chapter 12 516
11.2 The joint-embedding family, diagrammed 517
11.3 DINO: self-distillation with no labels 521
11.4 Evaluating DINO: k-NN and emergent attention 528
11.5 DINOv2: better data, two more losses 532
11.6 Scaling with data engines: DINOv3 and beyond 536
11.7 The BERT interlude: masked-token modelling for language 544

12 Self-Training, Distillation, Cycles 550
12.1 Self-Training: The Model Becomes Its Own Teacher 552
12.2 Knowledge Distillation: Soft Targets 561

12.3 Cycle Consistency: Learning Without Pairs . . . 571
12.4 CycleGAN: Two Intertwined Adversarial Games . . . 572
12.5 Why Self-Training Actually Works . . . 578
12.6 Chapter summary . . . 578
12.7 Exercises . . . 579
12.8 Concept Map . . . 582

13 World Models and Planning **583**
13.1 Abstraction: the goal of all feature extraction . . . 584
13.2 World models: are they self-supervised? . . . 587
13.3 DINO-WM: a world model in a frozen representation space . . . 592
13.4 Planning as test-time optimization . . . 604
13.5 Failure modes . . . 613
13.6 The systems connection: who records the experience? . . . 615
13.7 Chapter summary . . . 615
13.8 Check Your Understanding . . . 616
13.9 Concept Map . . . 619

14 System-Level Self-Supervision **621**
14.1 From Datasets to Systems . . . 623
14.2 A Zero-Background Guide to 3-D and Robot Words . . . 625
14.3 Maps as Supervision: A Robot That Fixes Its Own Segmentation . . . 636
14.4 Geometry as Supervision: Depth, Camera Motion and Flow . . . 654
14.5 Own Experience as Supervision: Where Can I Walk? . . . 667
14.6 Other Sensors as Supervision: Teaching Microphones to See Cars . . . 679
14.7 Interaction as Supervision: A Robot That Practises on Its Own . . . 689
14.8 The Big Picture: Three Ingredients of System-Level SSL . . . 701
14.9 Chapter summary . . . 702
14.10 Check Your Understanding . . . 703

III Research Perspective **706**

15 Why Does SSL Work? **707**
15.1 How to Read This Chapter: Four Levels of Evidence . . . 709
15.2 The Central Dogma and When It Breaks . . . 710

15.3 Why Joint-Embedding Works . . . 715
15.4 Why Masked Reconstruction Works . . . 719
15.5 Why Self-Distillation Without Negatives Works — Sometimes . . . 722
15.6 A Collapse Taxonomy Across All Book Families . . . 726
15.7 Scaling Beats Architecture at the Far End . . . 729
15.8 Landscape of the Book's Methods . . . 732
15.9 Summary . . . 733

16 Comparing the Families **739**
16.1 Reading the comparison tables . . . 740
16.2 Master comparison tables . . . 742
16.3 The confusion clearinghouse . . . 749
16.4 Decision guidance: which family first . . . 753
16.5 Evaluation cross-reference . . . 757
16.6 Summary . . . 758

IV Tutorials, Quizzes, Code Labs and the Exam **761**

17 Quiz: Autoencoders, VAEs and MAE **762**
17.1 The six ideas the quiz tests . . . 763
17.2 Questions 1–9: knowledge . . . 765
17.3 Questions 10–20: case studies . . . 769
17.4 The quiz on one page . . . 776

18 Code: a VAE and a Tiny MAE **777**
18.1 Coding Tutorial 1: a VAE, cell by cell . . . 777
18.2 Coding Tutorial 1: the play-around questions, solved . . . 789
18.3 Coding Tutorial 2: a tiny MAE, cell by cell . . . 790
18.4 Coding Tutorial 2: the play-around questions, solved . . . 803

19 Quiz: KDE, Flows and GANs **805**
19.1 The one table the quiz tests . . . 805
19.2 The ten questions . . . 806
19.3 The quiz on one page . . . 812

20 Quiz: Denoising and Diffusion **813**
20.1 Four facts about diffusion . . . 813
20.2 The sixteen questions . . . 814
20.3 The quiz on one page . . . 823

21 Quiz: Pretext Tasks and Contrastive Learning **824**
21.1 The map of the quiz . . . 825
21.2 Questions 1–7: predicting a transformation . . . 825
21.3 Questions 8–10: PIRL . . . 828
21.4 Questions 11–12: evaluating features . . . 829
21.5 Questions 13–18: contrastive learning . . . 830
21.6 The quiz on one page . . . 833

22 Quiz and Lab: Representation Learning **834**
22.1 The quiz . . . 835
22.2 The lab: making collapse visible . . . 841

23 Quiz and Case Study: Self-Training, Cycles, World Models **854**
23.1 The quiz: questions 1–12 . . . 855
23.2 The quiz: case studies 13–18 . . . 860
23.3 The case-study notebook, cell by cell . . . 863
23.4 What the notebook shows . . . 882
23.5 The final reflection questions, answered . . . 885

24 Quiz: System-Level Self-Supervision **888**
24.1 Where the supervision comes from . . . 889
24.2 The ten questions and the bonus . . . 889
24.3 The quiz on one page . . . 895

25 The First-Cycle Exam, Solved **896**
25.1 The format and how to spend 120 minutes . . . 897
25.2 The questions, solved . . . 898
25.3 What the paper tells you about the next one . . . 906

26 Question Bank I: Multiple Choice **908**
26.1 Part A: theory, lecture by lecture . . . 909
26.2 Part B: calculations . . . 919
26.3 Answer key . . . 924
26.4 The formula sheet for the calculations . . . 925

27 Question Bank II: Coding on Paper **926**
27.1 Clustering . . . 926
27.2 Autoencoders and generative models . . . 929
27.3 Representation learning . . . 932
27.4 Self-training, cycles, world models, systems . . . 934

28 Question Bank III: Case Studies and a Mock Exam **937**
28.1 Design questions . . . 937
28.2 Compare questions . . . 941
28.3 Procedure questions . . . 942
28.4 A complete mock exam (120 points, 120 minutes) . . . 943

V Exercises and Solutions **945**

29 Exercises and Solutions **946**
Part A: The Exercise Bank . . . 950
29.1 A.0 The course's own example exam questions (Lecture 07, slides 2–5) . . . 950
29.2 A.1 Chapter 1 — Why self-supervised learning? . . . 952
29.3 A.2 Chapter 2 — Technical foundations . . . 953
29.4 A.3 Chapter 3 — Clustering and classical–SSL hybrids . . . 955
29.5 A.4 Chapter 4 — Autoencoders and VAEs . . . 957
29.6 A.5 Chapter 5 — Masked autoencoders . . . 959
29.7 A.6 Chapter 6 — Flows and GANs . . . 960
29.8 A.7 Chapter 7 — Denoising, diffusion, flow matching . . . 961
29.9 A.8 Chapter 8 — Pretext tasks and augmentations . . . 963
29.10 A.9 Chapter 9 — Contrastive learning . . . 965
29.11 A.10 Chapter 10 — Beyond negatives . . . 966
29.12 A.11 Chapter 11 — Self-distillation and the DINO family . . . 968
29.13 A.12 Chapter 12 — Self-training, distillation, cycle consistency . . . 970

29.14A.13 Chapter 13 — World models . . . 971
29.15A.14 Chapter 14 — System-level SSL . . . 973
29.16A.15 Chapter 15 — Why does SSL work? . . . 974
29.17A.16 Chapter 16 — Comparing the families . . . 975
Part B: How to write a strong exam answer . . . 978
Part C: Capstone problems . . . 983

VI Reference **989**

A Mathematical Background **990**
How to use this appendix . . . 990
A.1 Probability and Statistics . . . 991
A.2 Linear Algebra . . . 995
A.3 Calculus: Derivatives, Gradients, and the Chain Rule . . . 999
A.4 Information Theory . . . 1002
A.5 Optimization . . . 1005
A.6 Symbol Index: Everything Not in the Notation Table . . . 1007

B Glossary **1011**

C Cheat Sheets **1031**
C.1 Sheet 1 — Key Equations . . . 1031
C.2 Sheet 2 — Methods Comparison Master Table . . . 1033
C.3 Sheet 3 — Papers Index (every entry in `references.bib`) . . . 1035
C.4 Sheet 4 — Terminology Card . . . 1038
C.5 Sheet 5 — Researcher's Checklist . . . 1039

D Slide Coverage Index **1041**

E Study-Session Transcripts **1051**
E.1 Lecture 02: classical methods — metrics, indexing, clustering, DeepCluster, STEGO . . . 1051
E.2 Lecture 03: autoencoders — bottlenecks, the ELBO, reparameterization, MAE 1063
E.3 Lecture 04: generative models — KDE, normalizing flows, GANs . . . 1070
E.4 Lecture 05: denoising generative models — DDPM algebra, rectified flows, latent diffusion, DIFT . . . 1075

E.5 Lecture 06: pretext tasks and augmentations — RotNet, PIRL, InfoNCE, SimCLR, MoCo, Barlow Twins 1079
E.6 Lecture 07: representation learning — SimSiam, BERT, DINO, DINOv2, data engines 1085
E.7 Lecture 08: mixtures — self-training, distillation, CycleGAN, world models, and the case study 1089

Preface

This book accompanies the university course *Self-supervised Learning* (Hermann Blum, University of Bonn). It is not a summary of the lecture slides. The slides define the *structure and intellectual flow*; this book expands every idea until a motivated reader can understand it from first principles, re-derive its mathematics, implement it, and locate it in the research literature.

Who this book is for. Someone meeting self-supervised learning for the first time. You should know what a neural network is and roughly how one is trained; everything beyond that — the linear algebra, the probability, the clustering, the information theory — is built up in Chapter 2 and pointed to from wherever it is needed. You should finish able to read current SSL papers.

Every major idea is presented at four levels, in this order, and you can stop at whichever one you need. *Plain words*: the idea with no notation at all. *Mechanism*: what actually happens to the data, step by step. *Mathematics*: the objective written out, with every symbol named. *Research picture*: the assumptions, the failure modes, the competing explanations, and what is still unknown.

How the chapters are shaped. Each chapter opens with a *Chapter roadmap* — what you need first, the idea in one sentence, the route through the chapter, and what you will be able to do at the end — followed by its own contents list. Every named model then gets the same treatment: what it is, why it was introduced, what was broken in the model before it, a *Model at a glance* card, the intuition, a step-by-step walk-through of how data moves through it, the mathematics, a small numerical example you can check by hand, a diagram, a *What changed* comparison with its predecessor, and a *Key takeaway*. Chapters close with a summary and a set of exercises.

About the diagrams. Diagrams in this book are meant to be read, not admired. Most carry a *How to read it* line telling you what to look at first. Conceptual figures are drawn from scratch; photographs, sample grids, attention maps and result tables are reproduced from the course slides and credited underneath.

How to use the book.

- **Reading once, quickly.** Read the intuition and worked examples, and the "Why does this actually work?" sections. Skim proofs.
- **Studying for the exam.** Follow Part II chapter by chapter, starting from each chapter's roadmap. Every slide of the course is mapped in the Slide Coverage Index (Appendix D). Do the *Check Your Understanding* boxes; fully worked answers are in Part V. The course's own tutorial quizzes, coding notebooks and case studies are worked through in Part IV.
- **Using it as a reference.** The glossary, cheat sheets, and method comparison tables in Part VI are designed for quick lookup.
- **Going into research.** Part III collects the hardest material: why SSL works (and when it does not), how the method families differ, and what is still unresolved. Research Frontier boxes mark open problems explicitly.

The coloured boxes, and what each one is for. Boxes are not decoration; each colour means one thing, consistently, in every chapter.

Box	What it means
Chapter roadmap	Where you are, what you need first, where this chapter goes.
In this chapter	The chapter's own contents list.
In plain words	The same idea again, with no notation. Read this first if a passage loses you.
Everyday analogy	A picture from ordinary life — always followed by where the analogy breaks.
New words used from here on	Vocabulary defined before it is used.
Step by step	An ordered walk-through with no stage skipped.
Model at a glance	The A-to-Z summary card for one named model.
What changed	The delta from the previous model in a lineage: what stayed, what changed, what it cost.
Key Concept	The precise definition, at first use.
Worked Example	Numbers small enough to check on paper.
Simplified intuition vs. more precise view	Used whenever the easy story hides a real caveat.
Common Confusion	A mistake readers actually make, and why.
Failure Modes	Where the method breaks, stated plainly.
Key takeaway	The one thing to remember if everything else is forgotten.
Paper Analysis	A close reading of one paper.
Research Frontier	What is genuinely unresolved.
Historical Note	Where the idea came from.
Check Your Understanding	Exercises; full worked answers in Part V.
Concept Map	How the chapter connects backwards and forwards.

Equations introduce every symbol on first use. Code is short and explanatory (PyTorch-style pseudocode) rather than production-complete.

Honesty rule. Throughout, we distinguish explicitly between *empirical observations*, *mathematical results*, *theoretical interpretations*, and *hypotheses*. Where the literature disagrees about why a method works, this book says so rather than presenting one story as settled.

Course-to-Book Mapping

The course has eight technical lecture decks, 02–09 (lecture 1, an organizational introduction, has no technical content). Every slide is accounted for; the fine-grained per-slide index is Appendix D.

Lecture deck	**Topic**	**Book chapter(s)**	**Material expanded beyond slides**
02-classical_ methods	Nearest neighbors, metrics, KD-trees, LSH; k-means, DBSCAN, OPTICS; DeepCluster, STEGO	Ch. 2 (tools), Ch. 3	ANN ecosystem; clustering theory; CRFs; pseudo-label round trip
03-autoencoder	Autoencoders, VAE, Masked Autoencoders	Ch. 4, Ch. 5	ELBO derivation; reparameterization; information bottleneck; BEiT/iBOT context
04-generative_ models	KDE, normalizing flows (Real NVP, GLOW), GAN, StyleGAN, Pix2Pix, taxonomy	Ch. 6	Change-of-variables theory; invertible layers; GAN training dynamics; disentanglement
05-generative_ denoising	Denoising AEs, diffusion, rectified flow/flow matching, CNF, classifier-free guidance, latent diffusion, DIFT, Diffusion Policy	Ch. 7	Score matching; SDE/ODE views; guidance theory; correspondence evaluation

Lecture deck	Topic	Book chapter(s)	Material expanded beyond slides
06-pretext_ augmentations	Augmentations; pretext tasks (RotNet, jigsaw, colorization); evaluation; PIRL; contrastive losses; SimCLR; MoCo; Barlow Twins teaser	Ch. 8, Ch. 9, Ch. 10	Augmentation theory; invariance vs. covariance; InfoNCE/MI; false negatives; projection heads
07-representation_ learning	Evaluation taxonomy; SimSiam; BERT; DINO; DINOv2; DINOv3 data engines	Ch. 10, Ch. 11	Collapse analyses; masked language modeling; self-distillation; data curation at scale
08-mix	Self-training, pseudo-labels, distillation, Noisy Student, DUNE, CycleGAN, world models, DINO-WM	Ch. 12, Ch. 13	Semi-supervised learning theory; knowledge distillation; cycle consistency; planning with learned models
09-systems	System-level SSL: map supervision, geometric consistency, proprioceptive supervision, cross-modal distillation, autonomous RL	Ch. 14	Continual learning; uncertainty estimation; self-supervised RL; sensor-fusion geometry

Notation

Consistent notation is used across all chapters.

Symbol	Meaning
$\boldsymbol{x} \in \mathbb{R}^d$ (or $\mathbb{R}^{C\times H\times W}$)	a raw input sample (image, text, ...)
$\boldsymbol{z} \in \mathbb{R}^{d_z}$	a latent variable or learned representation/embedding
$\boldsymbol{h}$	an intermediate representation (e.g. before a projection head)
$\hat{\boldsymbol{x}}$	a reconstruction / prediction of $\boldsymbol{x}$
$f_{\boldsymbol{\theta}}, g_{\boldsymbol{\phi}}$	neural networks with parameters $\boldsymbol{\theta}, \boldsymbol{\phi}$
$p(x)$, $q(\boldsymbol{z} \mid \boldsymbol{x})$	probability distributions (data, approximate posterior)
$\mathbb{E}_{x\sim p}[\cdot]$	expectation under distribution p
$D_{\mathrm{KL}}(p \,\|\, q)$	Kullback–Leibler divergence of p from q, $\mathbb{E}_p[\log(p/q)]$ (not symmetric)
$I(X; Z)$	mutual information between random variables X and Z
$\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v})$	similarity (cosine unless stated): $\boldsymbol{u}^\top \boldsymbol{v}/(\|\boldsymbol{u}\| \, \|\boldsymbol{v}\|)$
τ	temperature scaling a softmax / InfoNCE logits
$\mathcal{L}$	a loss/objective; subscripts identify the variant (e.g. $\mathcal{L}_{\mathrm{NCE}}$)
$\mathrm{sg}(\cdot)$	stop-gradient: identity in forward pass, zero gradient
m	EMA/momentum coefficient for a slowly-updated teacher/encoder
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	multivariate Gaussian with mean $\boldsymbol{\mu}$, covariance $\boldsymbol{\Sigma}$
N, B, K	dataset size, minibatch size, number of clusters/negatives (context defines)
$T(\cdot), t(\cdot)$	(data) augmentation / transformation sampled from a family $\mathcal{T}$
$[n]$	the set $\{1, \ldots, n\}$
$\odot$	element-wise (Hadamard) product

Chapter-specific notation is defined where it appears; when a choice collides with this table, the table wins.

Part I

Foundations

CHAPTER 1

Why Self-Supervised Learning? Labels, Tasks, and Evaluation

> The essence of many SSL methods: anything better than chance can be exploited as a learning signal.
>
> *Hermann Blum*, course lecture on clustering (Lecture 02, slide 23)

In this chapter

1.1 Supervised and self-supervised: one loop, one difference 4
1.2 The cost of labels 6
1.3 What a representation is 9
1.4 Pretext tasks and downstream tasks 12
1.5 A map of the book: three families 15
1.6 Distance in embedding space: a worked example 17
1.7 How do we know it is good? The evaluation taxonomy 20
1.8 How we got here: a short prehistory 27
1.9 Chapter summary 28

Chapter roadmap

Before you start. Nothing. This is the first chapter and it assumes no machine learning background beyond the idea that a program can be *trained* on examples. Every tool it mentions in passing — nearest neighbours, clustering, cosine similarity, the softmax — is built from scratch in Chapter 2, and you can read that chapter afterwards, or dip into it whenever a word here feels unearned.

The idea in one sentence. If you can write a rule that turns a raw input into its own training target, you can train a network on mountains of unlabelled data and then use the resulting features for the task you actually cared about.

What we cover.

- *Why bother* (Section 1.2) — what a human label actually costs, and why the unlabelled pile grows faster than anybody can annotate it.
- *What we are trying to build* (Section 1.3) — a *representation*: a space in which "close together" means "similar in the way I care about".
- *The two-step recipe* (Section 1.4) — invent a fake task with free answers, train on it, throw the fake task away, keep the features.
- *The three ways people invent that fake task* (Section 1.5) — the map of the whole book, in one figure.
- *How you know it worked* (Section 1.7) — the hardest part, because the thing you care about was deliberately excluded from training.

Afterwards you will be able to (i) say in plain words what makes a method self-supervised rather than unsupervised or semi-supervised; (ii) name the three method families and say which fake task each one uses; (iii) read a table of SSL results and say what each column is actually measuring; and (iv) explain why a falling training loss is *not* evidence that a self-supervised model is learning anything.

In plain words — the whole book in five sentences

1. Labels — the human-written answers that ordinary machine learning trains on — are expensive, slow, and sometimes impossible to get, while raw unlabelled data is nearly free and piling up everywhere.
2. So we invent a *fake* question whose answer can be computed automatically from the raw input itself ("which quarter-turn did I apply to this photo?", "what was under the patch I deleted?"), and we train a network to answer it.
3. Answering the fake question well is impossible without understanding something real about the input, so the network is forced to build useful internal descriptions of what it sees.
4. We then throw the fake question away and keep those internal descriptions — the *representation* — and attach a very small, cheap piece of machinery on top to do the real job we wanted.
5. Everything hard in this field is one of two questions: which fake question to ask, and how to tell whether the representation you got is any good.

This book is about a bargain. Ordinary supervised learning trades labels for knowledge: show it enough input–answer pairs and it learns to produce answers. Self-supervised learning (SSL) makes a different trade. It pulls out knowledge that was already sitting in the structure of the input, so that nobody ever has to write an answer down. This chapter explains why that bargain is worth making, fixes the vocabulary we will use for the rest of the book, and — most importantly — defines how we know when it worked. That last part is harder than it sounds, because the task we actually care about is deliberately left out of the training objective.

New words used from here on — three words used on every page from here on

- A **label** is an answer a *person* wrote down for one example: the word "dog" attached to a photograph, the outline a radiologist drew around a tumour. Labels are the raw material of supervised learning and the expensive thing this book is trying to avoid. A target computed by a fixed rule from the input — "the patch I just deleted" — is deliberately *not* called a label; it is a *pretext target*.
- A **representation** (or *embedding*, or *feature vector*) is the list of numbers a trained network produces for an input before any final decision is made. Write it $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$: the input is $\boldsymbol{x}$, the network is f, the numbers it learned are $\boldsymbol{\theta}$, and the output list is $\boldsymbol{z}$. The whole point of a good representation is that *arithmetic on those numbers means something*: two inputs you would call similar should give two lists of numbers that are close together.
- **Transfer** means taking a network trained for one purpose and using it for a different one. In this book it almost always means: pretrain on a fake task with no labels, then reuse the representation for a real task with a few labels. "Does it transfer?" is the only question that finally matters about a self-supervised method.

1.1 Supervised and self-supervised: one loop, one difference

Before any method, one picture. Supervised learning and self-supervised learning run *the same training loop*. Data comes in; the network makes a guess; the guess is compared against a target; the difference is used to nudge the network. Four of those five boxes are identical in both settings. Figure 1.1 draws the loop twice and highlights the one box that differs: *who writes the target*.

Two consequences follow immediately, and they shape everything after.

First, the target in the lower lane is *free and unlimited*. You can run the recipe on every image you will ever collect, and the recipe never gets tired, never disagrees with itself, and never charges by the hour.

Second, the target in the lower lane is *not the thing you want*. Nobody needs a machine that predicts deleted patches. You want a machine that recognises diseases, or navigates a corridor, or finds the photo you are thinking of. So the self-supervised lane has an extra step the supervised lane does not: after training, you throw away the part of the network that predicted patches and keep the middle — the representation — then attach something small to it that does the real job. Section 1.4 makes that two-stage structure explicit, and Section 1.7 is entirely about how to measure whether the kept middle is any good.

Key takeaway

Self-supervised learning is not a new kind of training. It is ordinary training with the human replaced by a rule. Everything interesting in the field is a consequence of that

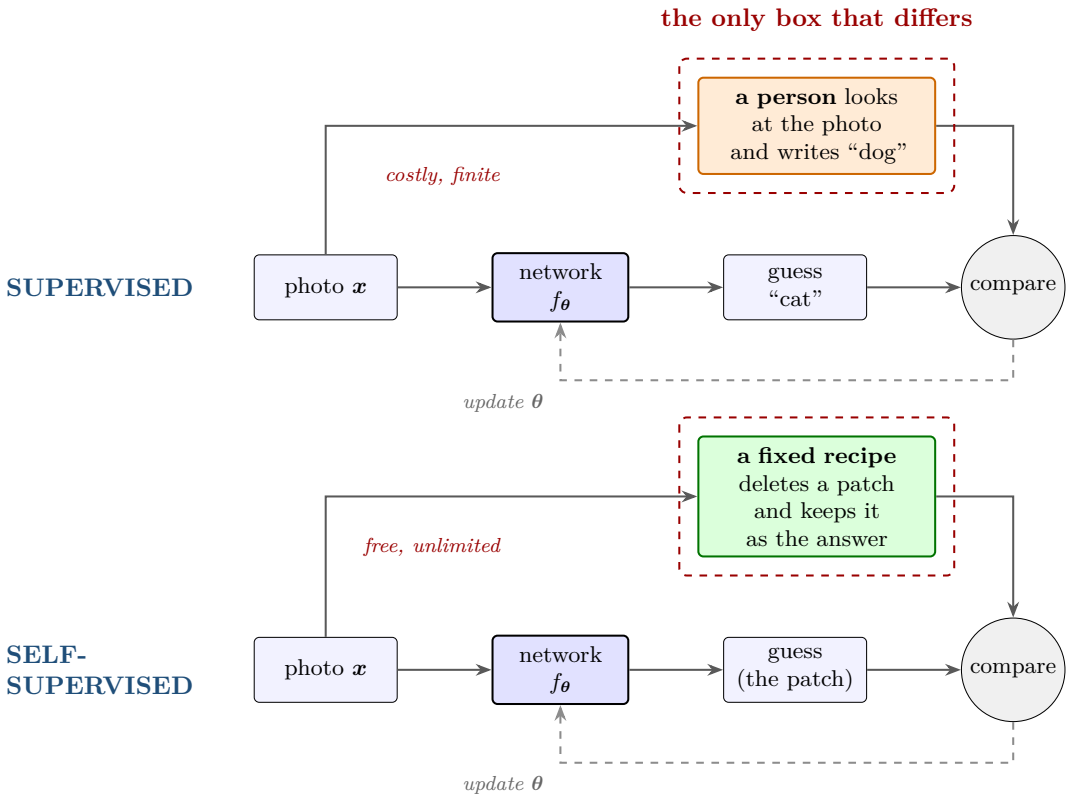


Figure 1.1. What “self-supervised” actually means. Both lanes run one training loop: a photograph goes in, the network guesses, the guess is compared with a target, and the difference nudges the network’s numbers. In the supervised lane a *human* produces the target by looking at the photo and writing a word. In the self-supervised lane a *fixed automatic recipe* produces it — here, “delete a patch and keep the deleted patch as the answer” — so the target costs nothing and there is an unlimited supply of it. Nothing else changes. This is the entire definition of the field; every method in this book is a different choice of recipe for that one box.

How to read it: Cover the two coloured boxes with your thumb. What is left is identical in the two rows — that is the point. Then uncover them: the top one costs a human’s attention per image, the bottom one costs a few lines of code, once.

one substitution — including the fact that the training loss no longer tells you whether things are going well.

1.2 The cost of labels

Modern computer vision stands on supervised training. LeCun et al. (1998) showed that gradient-based learning on tens of thousands of handwritten digits (the 60,000-image MNIST training set) yields a working document reader; Deng et al. (2009) then assembled ImageNet, roughly 1.3 million training images sorted into 1000 object categories, and the deep learning era of vision was built on it. Both share a hidden dependency: every sample carries a label written by a human. Supervised accuracy scales with labeled data, and labels scale with people, so progress is gated by annotation throughput and annotation money.

How expensive are labels, really? It depends on what the label is:

Label	Rough cost
"dog: yes/no" on a photo	a few seconds of a crowdworker's time; millions are easy to get
which of thousands of bird species	trained observers
pixel-accurate map of a street scene	on the order of one to two hours of skilled work *per image* (reported for benchmarks like Cityscapes)
medical images	scarce specialists whose time is expensive — and whose disagreements with each other cap how clean the labels can be

Meanwhile unlabelled data keeps growing for free: every camera, telescope, microscope and web crawl adds more. A method limited by labelling speed is tied to the slowest-growing resource in the system. A method that can learn from unlabelled data turns the fastest-growing resource in machine learning — raw data — directly into better models.

This is not hypothetical. The DINO family (Chapter 11) gives a clean comparison: same method family, four years, growing unlabelled data.

- **DINO** (Caron et al., 2021): ImageNet's 1.3 million images — with the labels deleted.
- **DINOv2** (Oquab et al., 2024): a curated pool of 142 million images, retrieved from a 1.2-billion-image raw crawl — far more than any labelling effort could cover.
- **DINOv3** (Siméoni et al., 2025): starting from about 17 billion public Instagram photos, a moderated and hierarchically clustered set of 1.689 billion images (Lecture 07, slide 51), plus curated retrieval sets and a small 16-million-image portion including ImageNet and street-level imagery.

No label is ever read. And the payoff is not just size: DINOv3's frozen features match or beat supervised and weakly supervised networks of similar size on classification and dense-prediction benchmarks. The missing labels only cost us information while nobody knew how to find structure without them.

Those four numbers are hard to feel as text: the first jump (from 1.3 to 142 million) is about a hundredfold, and each later one is roughly tenfold. Figure 1.2 plots them on a logarithmic scale, where every gridline to the right means "ten times as many images".

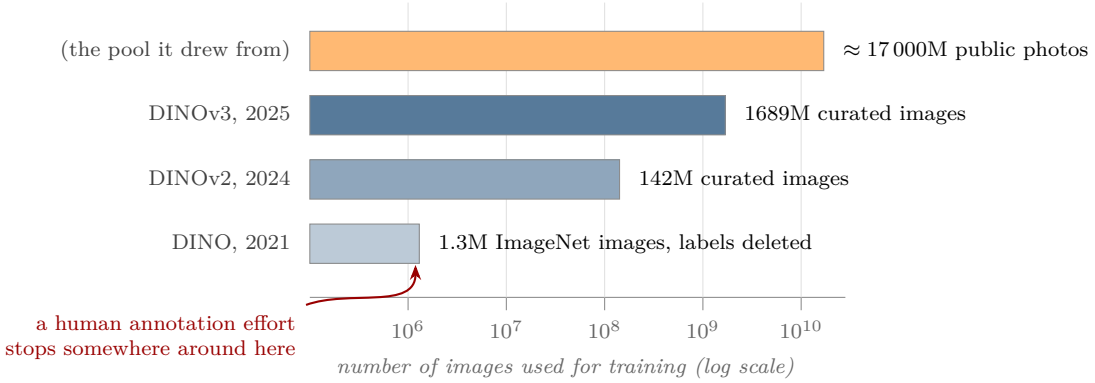


Figure 1.2. Why the field stopped waiting for labels. Each bar is one self-supervised training run from the DINO lineage (Caron et al., 2021; Oquab et al., 2024; Siméoni et al., 2025); the axis is logarithmic, so each gridline to the right means ten times as many images. ImageNet — the largest hand-labelled vision dataset most researchers ever touch, and itself a multi-year annotation project (Deng et al., 2009) — is the *smallest* bar here, and DINO used it with the labels thrown away. The top bar is the raw pool DINOv3 curated from; no annotation budget reaches it, so any method that requires labels simply cannot be trained on it.

How to read it: Look only at how far each bar sticks out past the one below. Every gridline it crosses is another factor of ten in data that self-supervision can use and supervision cannot — two gridlines for the first step, about one for each step after.

Key takeaway

> Labelled data grows at the speed of human attention. Unlabelled data grows at the speed of cameras. A method that can only learn from the first is tied to the slowest-growing resource in the system.

ImageNet is the *best case* for this model, and even it shows the strain. Categories like "dog" have crisp, broadly shared definitions, so thousands of annotators can label pictures of dogs quickly and consistently; even so, assembling a million clean labels was itself a research contribution (Deng et al., 2009). Three pressures appear as soon as we step away from object-centric categories:

- **Expertise-gated labels.** Medical images need radiologists, traversability labels need field roboticists, industrial defects need trained inspectors. The price of a million labels is no longer a budget line but a years-long organizational project — and for novel sensors or new terrain, the experts may not yet agree on what the classes are.
- **Ambiguous labels.** Many problems have no unanimous ground truth at all: scene boundaries in video, intent of an observed agent, "species" of a plant from one photograph (the case study in Lecture 07, slide 4: two million plant images, an unknown number of

species, and a colleague who labels 2000 of them — the intended solution is to learn features without labels first, then exploit the 2000). Supervision inherits every annotator disagreement as label noise.

- **The scaling asymmetry.** Unlabeled data is essentially free and gets cheaper every year, while labels inherit human hourly wages forever. As the DINO lineage above shows — 1.3 million images in 2021 (Caron et al., 2021), 142 million in DINOv2 (Oquab et al., 2024), 1689 million in DINOv3, drawn from a pool of about 17 *billion* (Siméoni et al., 2025) — no conceivable annotation effort labels even a percent of what a modern pipeline trains on. Depending on labels means choosing to learn from a vanishing slice of the data that exists.

The plant case study, solved. This exam question is the whole book in miniature. Keep three things apart:

- the **images**: two million, unlabelled, and more are free;
- the **species**: an unknown number, decided by biology, not by whoever labels;
- the **labels**: two thousand, from a colleague whose time is the real bottleneck.

Plain supervised learning trains on the 2000 labelled images and ignores the other 99.9% — backwards, because the unlabelled pool is where the species structure actually lives.

The self-supervised answer reverses the order:

1. **Learn from everything, without labels.** Train a feature extractor $f_{\boldsymbol{\theta}}$ on all two million images with any objective from this book. No knowledge of species is needed.
2. **Use the labels only to name things.** On top of the frozen features, fit a cheap readout with the 2000 labels — a kNN vote or a linear probe (Section 1.7).

If step 1 worked, images of the same species already sit together in the feature space, and the colleague's labels merely say which group is which. The general rule: *do the learning on the plentiful resource, and spend the scarce one only on a small readout.*

Learning note from study sessions

Reduce every SSL pitch to the two sentences of the plant case study: *what is your free signal* (the structure in two million unlabeled images) and *what is your cheap readout* (kNN or a linear probe on the 2000 labels)? A method that cannot answer both crisply is missing either its pretext task or its evaluation — the two load-bearing pieces of this chapter.

Key Concept — Self-supervised learning

Self-supervised learning trains a model on a supervisory signal computed automatically from the unlabeled input $\boldsymbol{x}$ alone — predicting a missing part, identifying which transformation was applied, or aligning two views of one scene — with no externally provided

target $\boldsymbol{y}$. The synthetic task is rarely the goal; the goal is a representation $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ that transfers to tasks we actually care about.

Why should we hope this works at all? The course gives a memorable answer while introducing DeepCluster (Caron et al., 2018) (Lecture 02, slide 23): *anything better than chance can be exploited as a learning signal.* The argument goes like this. Cluster unlabeled images in whatever feature space you currently have; the assignments are crude, but as long as they agree with semantic categories a little better than random guessing, they can serve as *pseudo-labels*. Train a classifier to predict them, and its features improve; cluster again and the assignments improve; iterate, and a signal barely above chance has been bootstrapped into a strong representation. Chapter 3 develops this loop in full. For now, keep the thesis sentence: it is the logical core of every method in this book. Rotation prediction, inpainting, masked-token prediction, and joint-embedding alignment differ only in *where* they find their above-chance signal.

1.3 What a representation is

Nearly every method in this book produces a feature extractor $f_{\boldsymbol{\theta}}$ mapping a raw input $\boldsymbol{x} \in \mathbb{R}^d$ (or $\mathbb{R}^{C\times H\times W}$ for images) to an embedding

$$\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{R}^{d_z}, \tag{1.1}$$

where $\boldsymbol{\theta}$ are the learned parameters and typically $d_z \ll d$. The lecture states the goal plainly (Lecture 07, slides 6–7): map the input space X into a space Z that is an *abstraction* of the data, more convenient to process than the raw input, in which *points carry a meaningful distance metric.*

In plain words — what a representation actually is

Feed a photograph into a trained network and stop just before the final decision. What you are holding is a list of a few hundred numbers. That list is the representation. Nobody assigned meanings to the individual numbers — there is no "ear-pointiness" slot. What the training arranged is something weaker and far more useful: if you compute that list for two photographs and the two lists come out close together, the two photographs are similar *in whatever way the training objective decided to care about.* The representation is not a description of the image. It is a position, and the only thing positions are good for is comparing them to other positions.

Everyday analogy — a subway map

A subway map is a picture of a city in which almost every measurement is wrong. The distances are wrong: two stops that look a centimetre apart may be four kilometres apart

underground. The directions are wrong: lines are bent onto 45° angles that no tunnel follows. The shapes are wrong: the river is a blue ribbon, the parks have vanished.
And yet nobody is confused, because the map is not trying to preserve geometry. It preserves exactly one thing — *which stops are next to which* — and that is the only thing a traveller needs. Everything else was deliberately destroyed to make that one relationship easy to read.
A learned representation is a subway map of a dataset. The individual coordinates mean nothing, the absolute distances are not to scale, and the picture would be useless for any question the training objective did not care about. What survives is closeness: images that should be treated alike end up near each other, and that is enough for a nearest-neighbour lookup, a clustering, or a straight-line classifier to work.
Where the analogy breaks. A transit authority *chose* which relationship to preserve, wrote it in a design brief, and hired someone to draw it. Nobody chooses for a representation. An optimiser does, implicitly, by minimising a loss — and the loss was chosen for reasons that had nothing to do with your downstream task. So you never get to read the design brief; you have to reverse-engineer it from the map, which is what Section 1.7 is about. Worse, a subway map that lost the river is still a subway map; a representation that lost the information your task needed looks exactly as healthy as one that kept it, until you measure.

The last clause is the heart of the definition.

Pixel space has no useful geometry. Dim an image slightly and its pixel distance to the original can be larger than the distance to a completely different, bright image.

A good representation space fixes that: distance follows meaning. Two black Labradors sit close together and far from a Pomeranian, whatever the brightness, crop or camera.

Once distance means something, classic tools — nearest neighbours, clustering, retrieval — work with no further learning. That is why the classical methods of Lecture 02 (kNN, k-means, DBSCAN) keep returning in this book, both to evaluate representations and as parts of SSL methods. The two standard ways to compare feature vectors are the Euclidean distance $\|\boldsymbol{u} - \boldsymbol{v}\|_2$ and the cosine similarity

$$\operatorname{sim}(\boldsymbol{u}, \boldsymbol{v}) \;=\; \frac{\boldsymbol{u}^\top \boldsymbol{v}}{\|\boldsymbol{u}\|\,\|\boldsymbol{v}\|}, \tag{1.2}$$

where $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^{d_z}$ are two embeddings. Cosine similarity discards magnitude and measures only the angle between the vectors, which is the right notion for most learned representations; the worked example in Section 1.6 makes this concrete. Figure 1.3 shows the whole point of the exercise in one picture: the same three photographs, laid out first by their raw pixels and then by a trained representation.

A complementary motto (Lecture 07, slide 8): representation learning learns the *invariances* and *co-variances* of the data so that the representation keeps only relevant information. An

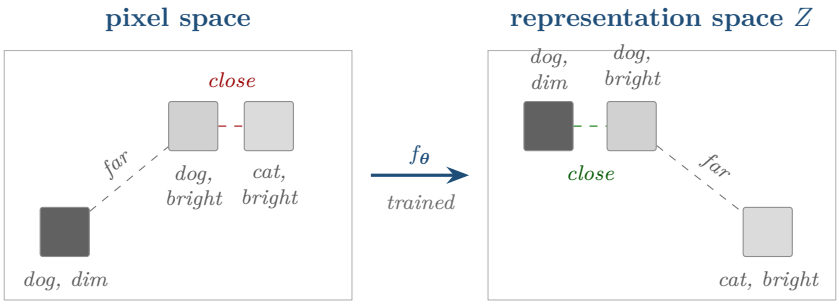


Figure 1.3. What training buys you. On the left, the three photographs are placed by their raw pixel values: the two dogs are far apart because one is dim and one is bright, while the bright dog and the bright cat are neighbours because most of their pixels happen to have similar values. Brightness — a property nobody cares about — dominates the geometry. On the right, the same three photographs after a trained encoder $f_{\boldsymbol{\theta}}$ has mapped them into Z: the two dogs are neighbours and the cat is far from both. Nothing about the photographs changed. What changed is which differences the space is allowed to notice.

How to read it: Compare the same pair across the two panels — the bright dog and the bright cat. They swap from "close" to "far" without anyone editing the images. That swap is the entire product of representation learning.

invariance is a transformation the representation should ignore (cropping, recoloring, masking); a co-variance is one it should track because it carries signal (the pose of an object, the position of a patch). Chapter 8 studies this trade-off in depth; for now, note that "relevant" is never defined by data alone — it is defined implicitly by the training objective and measured by the evaluations of Section 1.7.

What can a loss demand without labels?

- **Supervised learning** can demand invariance to *content*. Two photos of the same dog get the same label, so the classifier is penalised unless it maps them to the same place.
- **Self-supervised learning** cannot, because the content is exactly what has no label. Nothing tells it "both of these are Labradors".

What it *can* demand is invariance to a *process*. Take one image, apply two known changes that meaning should survive — different crops, different colour jitter — and require the two outputs to agree. The network is never told what the image is; only that whatever it computes must survive those changes.

So the list of augmentations stands in for the missing list of labels. That is why augmentation design is the hidden curriculum of SSL, why the same three objects (an input, an augmentation, an embedding) appear in every method chapter, and why Chapter 8 spends so long asking which invariances are safe to enforce and which throw away information a later task needs.

Key Concept — What makes a good representation

Three criteria recur throughout the book, each measurable and each in tension with the others:

- **Disentangledness.** Semantically distinct factors of variation (identity, pose, lighting, color) should correspond to independently varying directions or subspaces of Z, so a downstream model can use one factor without interference from the others.
- **Hierarchy.** The representation should expose structure at multiple levels — texture, parts, objects, scenes. Features that answer "which dog breed?" but have forgotten "where is the leg?" fail tasks the raw pixels could have solved. DINO's patch-level features emerging from a purely image-level loss (Chapter 11) are the canonical example of hierarchical structure appearing unasked.
- **Transferability.** The representation should be useful for tasks, datasets, and label budgets the pretraining run never saw. This is the operational criterion — the one the frozen-feature protocols actually compute.

A representation is not "true" in any absolute sense; it is a commitment about which differences in $\boldsymbol{x}$ matter. SSL methods differ precisely in which commitments their objectives induce — which is why evaluation matters as much as method design.

1.4 Pretext tasks and downstream tasks

Key Concept — Pretext and downstream tasks

A *pretext task* is a synthetic supervised problem constructed automatically from unlabeled data: predict the rotation applied to an image (Gidaris et al., 2018), predict a masked word (Devlin et al., 2019), decide whether two views came from the same image (Chen et al., 2020a). Its labels are free and unlimited. A *downstream task* is the real target application — classification, segmentation, retrieval — evaluated with human-provided labels. The pretext task is the means; the downstream task is the end. SSL research studies which means serve which ends.

In plain words — pretext and downstream, without the words

You want a machine that spots a disease. You do not have enough labelled scans to train one. So you train the machine on a different job entirely — one you can grade automatically, for free, on every scan you own. Then you fire it from that job and rehire it for the one you wanted, keeping only what it learned along the way. The first job is the *pretext task*; the second is the *downstream task*. The pretext task is a means and is thrown away; nobody ever reports how well the final system predicts rotations.

Every method in this book, without exception, is the same five-stage recipe with a different rule at stage 2. Learn the shape once and each later chapter becomes a single substitution.

Step by step — the pretext-to-downstream recipe, once and for all

Step 1. Collect raw inputs. Images, audio, text, sensor logs — whatever you have a lot of. No labels are collected, read, or paid for. In the plant case study of Section 1.2 this is the two million photographs.

Step 2. Choose a rule that manufactures targets. The rule is a fixed piece of code, not a person. It takes one raw input $\boldsymbol{x}$ and returns a (possibly damaged) input together with its own answer: "rotate by a random multiple of 90°, and the answer is which multiple"; "hide 75% of the patches, and the answer is the hidden pixels"; "make two different crops, and the answer is that these two belong together". Run it on every input and you have an unlimited supervised dataset that cost nothing. This is the *pretext task*, and it is the only stage where the three families of Section 1.5 differ.

Step 3. Train, exactly as in supervised learning. Forward pass, loss, backward pass, update. Nothing exotic happens here: the network is an encoder $f_{\boldsymbol{\theta}}$ followed by a small head that solves the pretext task, and both are trained together, for a long time, on a lot of data.

Step 4. Throw away the head; keep and freeze the encoder. The head that predicted rotations is useless to you — discard it. Keep $f_{\boldsymbol{\theta}}$, stop updating its numbers, and from now on treat it as a fixed function that turns an input into a vector $\boldsymbol{z}$. This frozen encoder *is* the product. Everything before this line was manufacturing it.

Step 5. Attach a cheap readout and spend your real labels here. Now, and only now, do human labels enter — and only a few thousand of them. Attach the smallest thing that can do your real job: a nearest-neighbour vote, a single linear layer, a tiny head. Section 1.7 is entirely about choosing that "smallest thing", because how much machinery you allow decides how much of the final score you are allowed to credit to $f_{\boldsymbol{\theta}}$.

Figure 1.4 lays the five stages out as one strip so you can see where the labels enter — late, and in small numbers.

Lecture 06 (slides 13–26) lists the classic pretext tasks:

- predict where one image patch sits relative to another (Doersch et al., 2015);
- put shuffled tiles back in order, like a jigsaw (Noroozi and Favaro, 2016);
- fill in a blanked-out region (Pathak et al., 2016);
- colour a black-and-white image (Zhang et al., 2016) or video (Vondrick et al., 2018);
- say which of four rotations was applied (Gidaris et al., 2018).

Each one needs some real knowledge of the world to solve — what objects normally look like, how parts fit together, that things persist over time. And, remarkably, each produces features that are useful for other tasks. The RotNet results table (slide 28) sets up the rest of

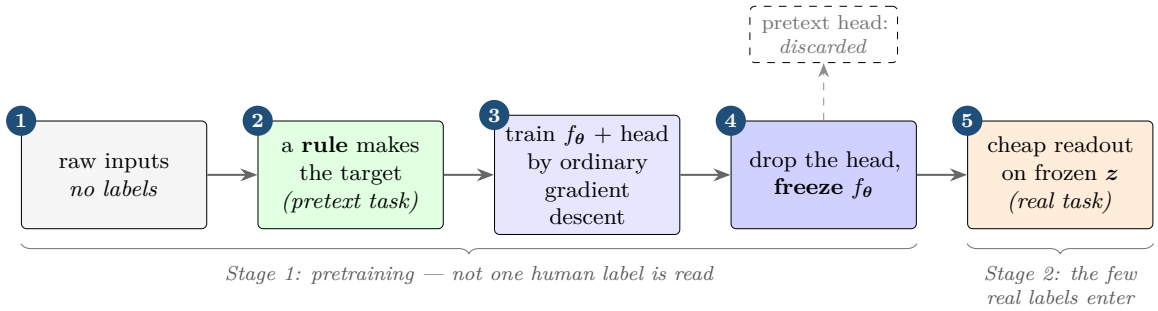


Figure 1.4. The five-stage recipe every method in this book instantiates. The long, expensive part (stages 1–3) uses no labels at all and can therefore run on as much data as you can store; the part that consumes labels (stage 5) is deliberately tiny. Stage 2 is the only box that differs between methods: put the damage back, name the damage, or ignore the damage (Section 1.5). Stage 4 is the step beginners leave out of their mental model — the network that ships is not the network that was trained, because the head that solved the pretext task is thrown away.

How to read it: Follow the labels, not the arrows. They appear for the first time in box 5, and there are only a few thousand of them. Everything to the left of that ran on free data.

the book: every column is a different way of testing the features, and the gap between the "trained with ImageNet labels" row and the best self-supervised row is the score the whole field is trying to close.

Common Confusion

Pretext loss is not downstream progress. In supervised learning, training loss and test accuracy reflect one quantity, so a decreasing loss is progress. In SSL the objective deliberately does not model the downstream task. The SimSiam experiments of Lecture 07 (slide 13) make this vivid: without stop-gradient, the alignment loss drops to its minimum almost immediately while kNN accuracy stays at essentially 0% — the network has *collapsed* to a constant representation that trivially satisfies the objective and encodes nothing. With stop-gradient, the loss decreases slowly while kNN accuracy climbs toward 50% (Chen and He, 2021). Reading SSL training curves as if they were supervised curves is the most common beginner's mistake; the remedy is the evaluation taxonomy of Section 1.7, never the pretext loss itself.

Common Confusion

Self-supervised, unsupervised, and semi-supervised are different settings. *Unsupervised learning* is the broad classical category: learning from unlabeled data, with the unlabeled problem (cluster the data, estimate its density) often an end in itself. *Self-supervised learning* is the modern subset in which a supervised-style procedure — a model, a loss, gradient descent — uses targets derived from the input, usually as *pretraining*; Caron et al. (2018) describe clustering assignments as pseudo-labels inside an otherwise standard classification loop, and Caron et al. (2021) frame DINO as distillation

with no labels at all. *Semi-supervised learning* is different in kind: some real labels *for the actual target task* exist and are used alongside unlabeled data, as in pseudo-labeling with a labeled seed set (Lee, 2013) — hence the course's remark that such methods are "semi-supervised, strictly." A quick detector settles borderline cases: if training used any human-provided label for the task being reported, the result is not purely self-supervised.

1.5 A map of the book: three families

The course organizes SSL methods into three recurring paradigms, drawn side by side in Lecture 06 (slide 73) and Lecture 07 (slide 9). They are three positions of a single control: hold the augmentation family fixed and vary what the network must do with it.

In plain words — three families, in nine words

Every method starts the same way: take an input and *damage* it on purpose — hide part of it, turn it, recolour it, crop it. The three families differ only in what they then ask the network to do about the damage.

- **"Put it back."** Show the network the damaged input and make it reproduce the original. To fill in a hole convincingly you have to know what usually goes in holes like that, so the network is forced to learn how the world is put together. *(Autoencoders, masked autoencoders, denoising and diffusion models: Chapters 4–7.)*
- **"Name the damage."** Show the network the damaged input and make it say *what you did*. To tell a 90° rotation from a 270° one you have to know which way up things normally sit, so again the network is forced to learn about the world — but a different part of it. *(Rotation prediction, jigsaws, relative patch location: Chapter 8.)*
- **"Ignore the damage."** Damage the *same* input twice, in two different ways, and make the network produce the same answer for both. It never learns what the input is; it learns what does not matter. *(Contrastive and joint-embedding methods: Chapters 9–11.)*

Notice the third one is strange in a way the first two are not: there is no right answer anywhere in it, only a demand for consistency. That is why it can cheat by outputting the same thing for every input — the failure called *collapse* — and why three chapters of this book are about stopping it.

The same three families, with their notation and their arrows, are drawn in Figure 1.5.

- **Reconstruction: be invariant to the augmentation.** Destroy part of the input (mask patches, add noise, drop color) and require the network to output $\hat{\boldsymbol{x}}$, a reconstruction of the original. To recover what was destroyed, the representation must encode the structure of $\boldsymbol{x}$. Autoencoders and variational autoencoders (Kingma and Welling, 2014; Vincent et al., 2008) (Chapter 4) and masked autoencoders (He et al., 2022) and

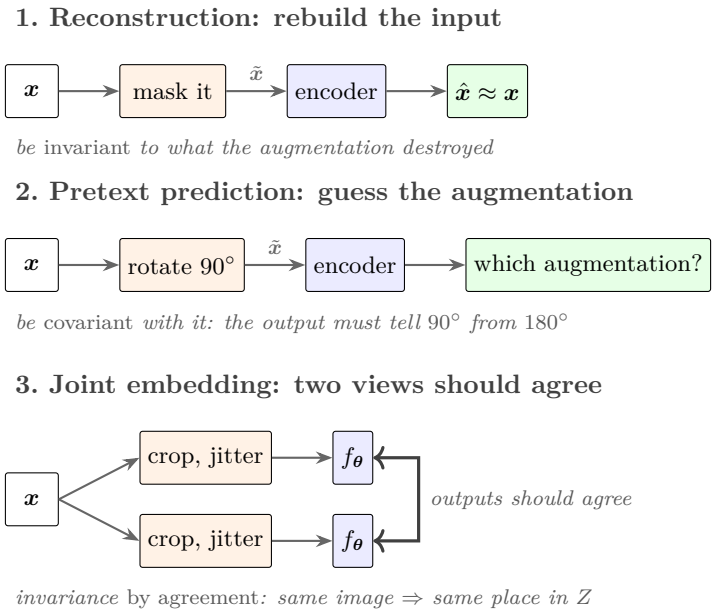


Figure 1.5. The three paradigms that structure this book, after the comparison schematics of Lecture 06 (slide 73) and Lecture 07 (slide 9). Every panel has the same three objects — an input $\boldsymbol{x}$, an augmentation, an encoder — and differs only in what the encoder's output is asked to do. *Reconstruction*: augment $\boldsymbol{x}$ into a damaged $\tilde{\boldsymbol{x}}$ (here, some patches masked), then predict $\hat{\boldsymbol{x}} \approx \boldsymbol{x}$ — the representation must be invariant to what was destroyed. *Pretext prediction*: augment $\boldsymbol{x}$ by a known transformation (here a 90° rotation), then predict *which* augmentation was applied — the representation must co-vary with it. *Joint embedding*: two different augmentations of one $\boldsymbol{x}$ are encoded separately and their outputs aligned — invariance by agreement. The rest of the book expands these three panels.

denoising models (Chapters 5 and 7) live here, as does masked token modeling in text (Chapter 11).

- **Predict-the-augmentation: be covariant with it.** Apply a known transformation and require the network to say which one: the RotNet recipe and its pretext siblings. Solving the prediction requires "visual commonsense" about what the unperturbed input looks like. Chapter 8 is devoted to this family and the augmentation theory beneath all three.
- **Joint-embedding alignment: two views should agree.** Apply two *different* augmentations of the same $\boldsymbol{x}$, encode both separately, and require the outputs to match — usually while pushing apart outputs of different images. Contrastive methods (van den Oord et al., 2018b; Chen et al., 2020a; He et al., 2020) (Chapter 9), negative-free and redundancy-reduction methods (Grill et al., 2020; Chen and He, 2021; Zbontar et al., 2021) (Chapter 10), and self-distillation (Caron et al., 2021; Oquab et al., 2024) (Chapter 11) implement this paradigm, differing mainly in how they prevent the collapse previewed above.

The families are not rivals so much as different bets on what a good representation is, and state-of-the-art systems mix them: DINOv2 combines a joint-embedding image-level loss with a reconstruction-flavored masked-patch loss (Oquab et al., 2024). Chapters 15 and 16 return to the comparison with the evidence assembled.

Which family, for which problem? The course never pretends the choice is free, and a rough decision logic emerges from the chapters that follow. Four questions, in the order to ask them:

1. *What does your downstream task need?* Dense prediction (segmentation, depth, tracking) needs features that retain spatial structure; reconstruction objectives, which force the network to model pixel-space content, deliver exactly that. In practice the strongest dense features come from *combining* the two: DINOv2 adds a masked-patch loss to a joint-embedding loss (He et al., 2022; Oquab et al., 2024). Pure invariance objectives that treat the whole image as one vector are the weakest choice here.
2. *What recognizably survives your augmentations?* Joint embedding requires augmentations that preserve identity (random crops and color jitter, in their classic SimCLR/DINO formulations); if no such augmentation family exists for your modality — a frequent problem outside photographic vision and text — pretext prediction of modality-specific structure (word order, keypoint layout, sensor channels) may be the only available lever.
3. *Can you state the structure you assume?* Reconstruction assumes that inputs lie near a low-complexity set the network can model; pretext prediction assumes you know a transformation whose inverse is diagnostic of content; joint embedding assumes invariance to a chosen nuisance. Pick the assumption you would defend in print.
4. *How much collapse risk can your optimizer absorb?* Invariance families all flirt with the collapsed constant-representation solution and differ mainly in the counterweight — negatives (Chapter 9), stop-gradient and predictors (Chapter 10), centering and EMA teachers (Chapter 11); reconstruction families are collapse-free by construction but pay for it by spending capacity on fine detail the downstream task may not need.

There is no universal winner; there is only a match between the information your task needs and the invariance your objective enforces. Sections of Chapters 15 and 16 make this comparison quantitative.

1.6 Distance in embedding space: a worked example

The course twice advises checking which distance a representation was trained to respect (Lecture 02, slides 4–5): cosine similarity is invariant to magnitude and measures only angle, and DINO descriptors are compared by cosine precisely because their training shapes directions, not lengths. Before the metric question, though, it helps to see what joint-embedding training does to the metric in the first place — and the effect is small enough to follow by hand.

Worked Example — Becoming neighbors: one joint-embedding step

Take the same 2-D toy setup, and one image: a photograph of a dog. An augmentation pipeline draws two *crops* of it — a tight crop on the head, $\boldsymbol{x}^{(1)}$, and a wider crop including the torso, $\boldsymbol{x}^{(2)}$. At initialization the network is a random function, so the two embeddings land wherever the random weights send them; suppose

$$\boldsymbol{z}^{(1)} = f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) = (0.40,\ 0.30), \qquad \boldsymbol{z}^{(2)} = f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}) = (0.10,\ -0.50).$$

Their cosine similarity before training is

$$\mathrm{sim}(\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}) = \frac{0.40 \cdot 0.10 + 0.30 \cdot (-0.50)}{\sqrt{0.40^2 + 0.30^2}\,\sqrt{0.10^2 + 0.50^2}} = \frac{-0.11}{0.50 \times \sqrt{0.26}} \approx \frac{-0.11}{0.255} \approx -0.43.$$

The two views of one dog point in nearly opposite directions: at initialization, the representation does not yet know they are related.
A joint-embedding loss (Chapter 9 onward) penalizes exactly this: it wants $\mathrm{sim}(\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)})$ large and, depending on the family, pushes embeddings of *different* images apart. A gradient step cannot move the vectors all the way, but it tilts them toward agreement; suppose that after some training the same forward passes give

$$\boldsymbol{z}'^{(1)} = (0.45,\ 0.44), \qquad \boldsymbol{z}'^{(2)} = (0.42,\ 0.50).$$

Now

$$\mathrm{sim}(\boldsymbol{z}'^{(1)}, \boldsymbol{z}'^{(2)}) = \frac{0.45 \cdot 0.42 + 0.44 \cdot 0.50}{\sqrt{0.45^2 + 0.44^2}\,\sqrt{0.42^2 + 0.50^2}} = \frac{0.409}{\sqrt{0.3961}\,\sqrt{0.4264}} \approx \frac{0.409}{0.411} \approx 0.995.$$

Two views that started at -0.43 (about 116° apart) now point within about 6° of each other ($\arccos 0.995 \approx 5.7°$). Nothing about dogs was ever stated; the only signal was "these two augment the same photograph." Repeated over millions of images and augmentation pairs, that single signal drags the geometry of Z into semantic shape: views of one object become neighbors, neighbors share neighborhoods, and clusters of similar objects condense — which is the same thing the t-SNE training animations of Lecture 07 (slides 35–36) show happening for real DINO features.

Figure 1.6 draws those four vectors. It is worth looking at, because it is the smallest honest picture of what a joint-embedding loss does: it does not move images, and it does not learn categories. It rotates arrows.

Notice what the example does *not* assume. The loss never compared either crop to a labeled dog image, and it never saw the word "dog." All semantic content entered through the choice of augmentation family: crops and jitter that *we* believe preserve object identity. If the augmentation destroys identity (say, a crop so tight it shows a textureless patch of fur), the same loss happily aligns two semantically unrelated inputs — the invariance was

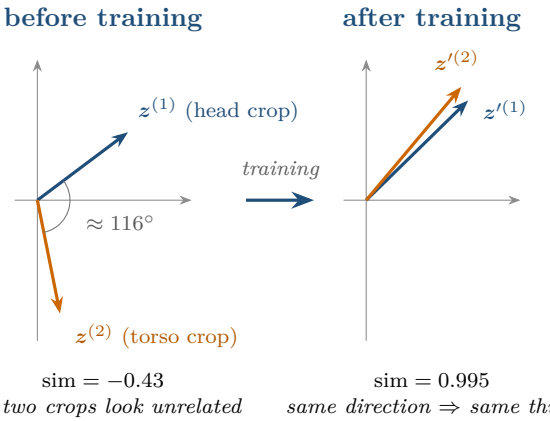


Figure 1.6. One joint-embedding step, drawn. Left: at initialisation the two crops of one dog photograph are sent to arrows pointing roughly *opposite* ways, cosine similarity −0.43 — the untrained network has no idea they are related. Right: after training the same two crops give arrows within a few degrees of each other, cosine 0.995. The loss never saw the word "dog" and never compared either crop with a labelled example; all it knew was that these two pictures came from one photograph. Repeat over millions of photographs and this single instruction drags the whole space into semantic shape.

How to read it: Watch the angle *between the two arrows, not their lengths. Cosine similarity reads the angle alone, and the angle is what training collapsed from* 116° *to almost nothing.*

bought, and misspecified invariances are the price. This is the operational reading of the invariance/covariance motto: the objective enforces whatever the augmentation family declares irrelevant, no more and no less.

Once training has made direction the carrier of meaning, the companion question — which distance reads that meaning back — stops being a matter of taste. A toy example shows it mattering in four numbers.

Worked Example — Cosine vs. Euclidean on direction-based features

Consider a 2-D toy embedding where *direction* encodes category and *magnitude* encodes a nuisance factor such as visual contrast. A query

$$\boldsymbol{q} = (1,\ 0),$$

and four gallery vectors: a confident dog $\boldsymbol{u}_1 = (4,\ 0)$, a faint, low-contrast dog $\boldsymbol{u}_2 = (0.1,\ 0.02)$, a cat $\boldsymbol{u}_3 = (0.5,\ 2.0)$ (a different direction, so a different category), and a bright, high-contrast dog $\boldsymbol{u}_4 = (5,\ 0.1)$. (Recall from the Notation table that $\boldsymbol{q}$ and $\boldsymbol{u}_i$ are query/key-style vectors; their roles here are as four gallery candidates matched against $\boldsymbol{q}$.)
Euclidean distances from $\boldsymbol{q}$:

$$\|\boldsymbol{q}-\boldsymbol{u}_1\|_2 = 3, \qquad \|\boldsymbol{q}-\boldsymbol{u}_2\|_2 \approx 0.90, \qquad \|\boldsymbol{q}-\boldsymbol{u}_3\|_2 \approx 2.06, \qquad \|\boldsymbol{q}-\boldsymbol{u}_4\|_2 \approx 4.00.$$

The nearest Euclidean neighbor of $\boldsymbol{q}$ is $\boldsymbol{u}_2$, the faint vector, simply because its magnitude is close to $\|\boldsymbol{q}\| = 1$; the two vectors that *point the same way* as the query, $\boldsymbol{u}_1$ and $\boldsymbol{u}_4$, are ranked last because their confidence (magnitude) differs.
Cosine similarities (via Equation (1.2)): the query has unit norm, so each score is the dot product divided by the gallery norm alone:

$$\text{sim}(\boldsymbol{q}, \boldsymbol{u}_1) = \frac{4.0}{4.0} = 1.000, \qquad \text{sim}(\boldsymbol{q}, \boldsymbol{u}_2) = \frac{0.10}{1 \times \sqrt{0.0104}} = \frac{0.10}{0.1020} \approx 0.981,$$

$$\text{sim}(\boldsymbol{q}, \boldsymbol{u}_3) = \frac{0.50}{1 \times \sqrt{4.25}} = \frac{0.50}{2.062} \approx 0.243,$$
$$\text{sim}(\boldsymbol{q}, \boldsymbol{u}_4) = \frac{5.00}{1 \times \sqrt{25.01}} = \frac{5.00}{5.001} \approx 0.9998.$$

Cosine restores the semantic order: the direction-aligned candidates $\boldsymbol{u}_1, \boldsymbol{u}_4, \boldsymbol{u}_2$ all score above 0.98, and the orthogonality-leaning $\boldsymbol{u}_3$ — the cat — scores far lower. The faint dog is the instructive case: its dot product with $\boldsymbol{q}$ is a negligible 0.10, but dividing by its own negligible norm 0.1020 normalizes magnitude away and lifts the score to 0.981 — cosine reads its direction as near-perfect agreement. Where Euclidean distance confuses confidence with identity, cosine sees only identity — which is exactly the property wanted when a training objective has made direction carry category. The price: if magnitude *does* carry task-relevant signal (it must have been trained to do so), cosine silently discards it, hence the lecture's instruction to check the training loss before choosing the metric.

An operational note completes the picture (Lecture 02, slide 5): in high dimensions the two metrics often agree in practice because contrastive-style training normalizes features onto a hypersphere, where Euclidean distance is a monotone function of cosine similarity: for unit vectors, $\|\boldsymbol{u} - \boldsymbol{v}\|_2^2 = 2 - 2\ \text{sim}(\boldsymbol{u}, \boldsymbol{v})$. The metric choice matters most exactly when this precondition fails.

1.7 How do we know it is good? The evaluation taxonomy

Evaluation is where SSL is most often done badly, so the course devotes a full taxonomy to it twice (Lecture 06, slides 29–37; Lecture 07, slide 10). The top-level split is between *direct inspection* of the feature space and *downstream* use of the features.

Key Concept — Direct vs. downstream evaluation

Direct (qualitative) inspection probes the representation itself — 2-D projections, attention maps — and answers structural questions: is the space organized? does it vary with the right things? *Downstream evaluation* measures performance after attaching some labeling

machinery to the features; the variants differ in how much capacity and how much label data that machinery is allowed.

Everyday analogy — four ways to assess a candidate you cannot let do the job

You are hiring, and for legal reasons the candidate cannot simply be given the job for a year to see how it goes. So you use proxies, and each one trades honesty for cost.

- **Look through their notebooks.** You cannot grade notebooks, but you can see at a glance whether the thinking is organised or a mess. This is **t-SNE and attention maps**: fast, revealing, and it never produces a number you can put in a table.
- **Hand them one small problem, cold, no preparation.** They either solve it from what they already know or they do not. This is **kNN on frozen features**: nothing is trained, nothing is tuned, so a good score can only be credited to what the candidate already had.
- **Give them a one-hour briefing, then test them.** A briefing cannot install competence, but it can let existing competence be applied. This is the **linear probe**: you fit one thin layer on top of frozen features, just enough to point existing structure at your question.
- **Put them on the job for three months.** Now you are measuring something real — but you are also measuring three months of training, your onboarding process, and how lucky the first project was. This is **full fine-tuning**: the highest number, and the one that tells you least about what the candidate brought with them.

Where the analogy breaks. A candidate does not change when you look at their notebooks; a representation does change under fine-tuning, and the change can be total. After three months on the job, the person you are assessing is not the person you hired — so a fine-tuning score is not a measurement of the frozen representation at all, but of the place in parameter space it started from. Also, real candidates improve monotonically with support, whereas the rungs here can disagree in *rank*: method A can beat method B by kNN and lose by linear probe, which has no sensible counterpart in hiring and is exactly what makes SSL evaluation interesting.

Direct inspection. Project the embeddings of a held-out set to 2-D with PCA or t-SNE (Jolliffe, 2002; van der Maaten and Hinton, 2008) and check whether clusters correspond to semantic categories, or animate the embedding space over training epochs and watch categories condense, as the course does for DINO features (Lecture 07, slides 35–36). For vision transformers, plot the self-attention maps of the classification token: after DINO training these maps segment foreground objects without any segmentation supervision, and different heads attend different parts (Caron et al., 2021). For pretext models, check covariance: the attention maps of RotNet's features rotate with the rotated input (Lecture 06, slide 34) — evidence the features track rotation, not ignore it. Visualizations rank hypotheses cheaply

and are indispensable debugging tools, but they never yield a number and can mislead when the projection destroys the geometry it displays: t-SNE exaggerates some clusters and tears others, so treat it as a microscope, not a scoreboard.

Downstream protocols, ordered by increasing capacity:

1. **kNN on frozen features.** Classify each test image by the majority vote of its k nearest neighbors in the embedding space, computed with the training split only — no parameters trained at all. The protocol comes from Wu et al.'s instance-discrimination work; DINO made it a headline metric as a deliberate fix to a methodological complaint: finetuning evaluations are sensitive to hyperparameters with large variance between runs, so the paper also evaluates "with a simple weighted nearest neighbor classifier (k-NN)" (Caron et al., 2021). Because the classifier has essentially no capacity, kNN accuracy measures how raw the semantic structure of Z is: the DINO ImageNet table (Lecture 07, slide 33) shows ViT-S features reaching 74.5% top-1 by kNN alone, where supervised ResNet-50 and ViT-S score 79.3% and 79.8% respectively.
2. **Linear probe.** Freeze $f_{\boldsymbol{\theta}}$ and train a single linear layer on top, on labeled downstream data. This is the field's default quality metric: a linear boundary on a good representation separates classes because the hard work (straightening the decision surface) was done during pretraining. The supervised ceiling and all SSL contenders are compared in this protocol in the PIRL table (Lecture 06, slide 43) and the DINO tables (Misra and van der Maaten, 2020; Caron et al., 2021).
3. **Mildly nonlinear heads.** Train a small multi-layer head — still modest capacity — on frozen features. Head-training answers "how good are the features when the downstream task is slightly nonlinear?" without entangling the answer with backbone optimization. The course groups kNN and head training under the same logic: *low capacity in the head*, so that success must be credited to the features.
4. **Clustering evaluation.** Run a clustering algorithm (k-means, DBSCAN) on frozen features and score the agreement of clusters with true classes (normalized mutual information, adjusted Rand index). This tests whether categories exist in Z at all before any labels — and closes the loop with Lecture 02, where clustering is not just an evaluation but the source of the learning signal itself (Caron et al., 2018).
5. **Few-shot and full finetuning.** Unfreeze everything and finetune end-to-end with labeled downstream data: with a small fraction of labels (e.g. 1% or 10% of ImageNet, as in the SimCLR transfer table (Chen et al., 2020a)) for the few-shot regime, or with all labels for the full regime. Finetuning measures the representation's *potential* as initialization rather than its frozen structure — with the variance caveat above.

Read as a ladder — and Figure 1.7 draws it as one — the protocols are not five versions of one measurement but a controlled transfer of credit. Each rung adds capacity to the readout and subtracts evidence that the representation itself was doing the work. It is worth stating, for the three rungs used most, *what a high score certifies* and *what the rung is blind to*.

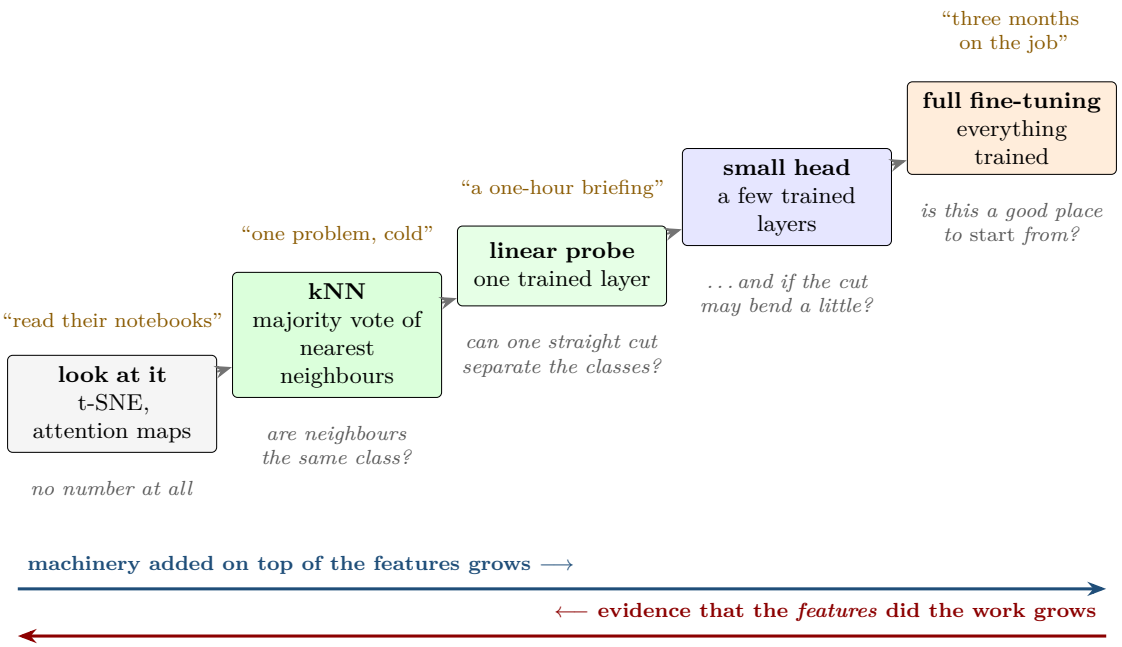


Figure 1.7. The evaluation ladder. Going right, you hand the readout more capacity and more labelled data; going left, more of whatever score you get is attributable to the frozen representation itself. No rung is "the" measurement: a high fine-tuning number and a low kNN number describe a representation that is a good *starting point* but a badly organised *space*, and that distinction matters enormously when you have five hundred labels rather than a million. Quote at least two rungs, always.

How to read it: Read the two long arrows at the bottom first — they point in opposite directions, and that opposition is the whole content of the figure. Then pick a rung and read the question underneath it: that is what a number from that rung actually answers.

kNN rewards local purity and nothing else. A high kNN score certifies that images of a class form neighborhoods whose nearest points are other members of that class — the embedding is *locally* organized. Correspondingly, kNN is blind to global geometry: two classes may sit in cleanly separated half-spaces, or in two tangled interleaved sheets; kNN sees none of it, because it never looks past the k nearest points. It punishes exactly one failure: ragged local geometry, classes interspersed at small scales. Note also that "no trained parameters" does not mean "no choices": the practitioner still fixes k, the distance metric (Section 1.6), and the neighbor weighting — the DINO paper adopts the variant of Wu et al.'s instance-discrimination classifier (their reference [60]), which weights each neighbor's vote by a softmax over its distance with a temperature. The recipe must be fixed across methods, exactly as with probes.

The linear probe rewards one global property: linear separability. Fitting a linear layer on frozen features is a convex optimization that only knows whether classes can be separated by a hyperplane. It rewards global straightness and punishes tangled-but-locally-pure geometry; it rewards disentangledness in the sense of Section 1.3, because a linear

readout survives only if the factors it needs are linearly readable. Unlike kNN it *is* a learning algorithm, and therein lies the ambiguity the field argues about: a probe is a small mechanism that can partially *repair* representational deficits (re-weighting, re-scaling, discarding nuisance directions), so a high probe score credits the features and the fitting jointly. Holding the probe recipe fixed keeps the fitting's contribution constant across methods, so that score differences can be read as representation differences.

Finetuning rewards a location, not a geometry. Once all parameters move, the evaluation no longer measures what Z *is*; it measures where $\boldsymbol{\theta}$ sits in the optimization landscape and how good a basin gradient descent finds from there. Brittle local structure can be entirely rebuilt during finetuning, so finetuning punishes almost nothing the frozen protocols would catch, and its numbers are correspondingly optimistic. The few-label regimes (1%, 10% of ImageNet) sharpen the question to "given very little supervision, from this starting point, how far can we get?" — the question that matters when choosing a pretrained backbone for a label-starved application.

The cleanest way to feel these three readings is to freeze one network and read it three ways:

Worked Example — One embedding, three readouts: DINO ViT-S on ImageNet

Freeze the ViT-S backbone that DINO trains (Chapter 11; Caron et al. (2021)) and attach three readouts to exactly the same frozen features. The ImageNet top-1 numbers that come back are the ones in Figure 1.8: 74.5% by kNN, 77.0% by linear probe, and — as the yardstick for what the same backbone achieves with end-to-end supervision — $\approx$ 79.8% for a supervised ViT-S, the reference the finetune rung is chasing. (These are the slide numbers of Lecture 07, slides 33–34, quoted here as slide evidence, free of any added citation machinery.)

Interpret the gaps as properties of Z — the features in all three columns are bit-for-bit identical. The 2.5-point step from kNN to linear says the representation's classes are *more* linearly separable than they are neighbor-pure: the space is straighter globally than it is clean locally. A few hundred thousand readout parameters (384 features $\times$ 1000 classes $\approx$ 384,000 weights) buy back structure that was present in the features but not arranged into pure neighborhoods — precisely the repair ambiguity discussed above. The step from linear to the supervised reference says something different: with end-to-end optimization and every ImageNet label, the same backbone reaches $\approx$ 79.8%, so the distance between the frozen self-supervised representation and a fully supervised solution of identical capacity is a handful of points. "SSL closes the gap" is, on these numbers, a statement about three different credit assignments of one embedding — not three embeddings. Were the order inverted (a hypothetical encoder with linear below kNN), the interpretation would flip: neighbor-pure but not linearly straight, the signature of a representation organized for retrieval rather than classification. The field's most-cited real example of a wide split is MAE: 49.4% kNN against 76.6% linear in the DINOv2

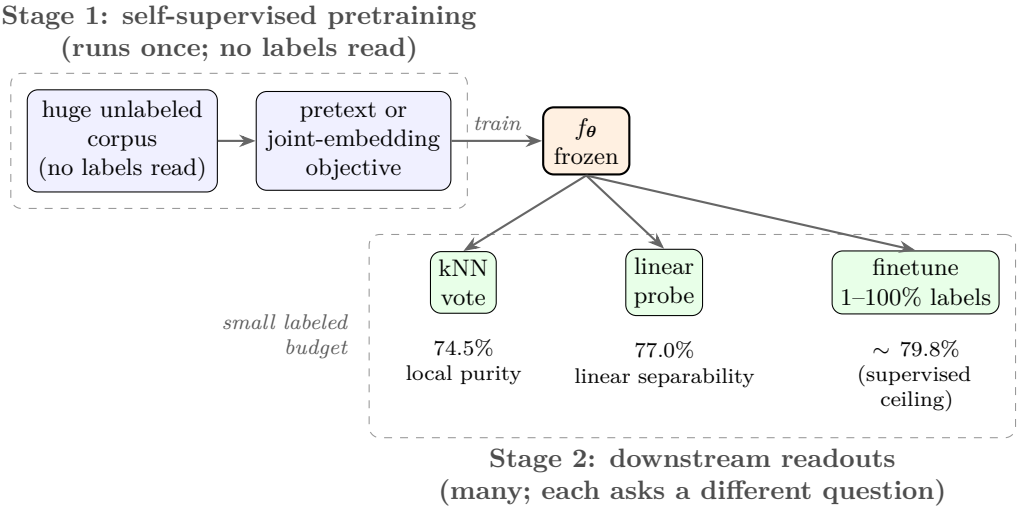


Figure 1.8. The standard two-stage evaluation pipeline, after Lecture 06 (slides 30–31), drawn to make the credit-attribution logic visible. Stage 1 trains the feature extractor once, on unlabeled data only. Stage 2 freezes $\boldsymbol{\theta}$ and attaches labeling machinery of increasing capacity — kNN, a linear layer, or full finetuning — on a small labeled budget. The numbers are a worked reading of one frozen backbone (DINO's ViT-S features, the chapter's toy three-way readout): kNN 74.5%, linear 77.0%, with the supervised same-backbone number $\approx$ 79.8% as the finetuning reference. Moving right along the bottom row transfers credit from the frozen geometry to the added machinery and the labels: the reading of the representation changes even though the representation does not.

comparison table (a large MAE model re-measured under the DINOv2 evaluation protocol (Oquab et al., 2024); the MAE paper itself reports 73.5% linear under its own protocol for ViT-L (He et al., 2022)), analyzed in the research-frontier box below.

Three conclusions from the toy survive contact with real leaderboards. First, *one number never describes one representation*; at minimum quote kNN and linear together. Second, the ordering across rungs is itself diagnostic — flat ladders and climbing ladders describe different geometries. Third, every rung presumes the recipe beneath it was held fixed; a kNN number with tuned k is a claim about k-engineering, not geometry.

Simplified intuition vs. more precise view

Simple story. A linear probe measures representation quality; a kNN measures it with even less machinery in the way; finetuning measures what the representation can become.

More precise view. Each protocol answers a *different* question, and none is a scalar order on representations. kNN asks whether geometry is semantically organized pointwise; a linear probe asks whether classes are linearly separable (a global, convex property of Z); heads ask about mild nonlinearities; finetuning asks whether $\boldsymbol{\theta}$ is a good basin from which to descend. The rank order of methods can legitimately differ across protocols — and does

in the DINOv2 table (Lecture 07, slide 42), where MAE under the DINOv2 evaluation protocol jumps from 49.4% kNN to 76.6% linear while DINO stays nearly flat (Oquab et al., 2024): MAE's geometry is tighter locally than its linear separability, and only by reading two protocols together does the representation's character become visible. (Under its own evaluation protocol the MAE paper reports a leaner 73.5% linear (He et al., 2022); the 49.4/76.6 pair above is DINOv2's uniform re-measurement, and the gap between the two linear numbers is a protocol difference, not a discrepancy in the representation.)

Failure Modes

- **Tuning the probe on pretext-task intuition.** The linear probe has its own hyperparameters (epochs, optimizer, augmentation at eval time); over-tuning them per method turns the benchmark into a measurement of probe-engineering effort. Fix the probe recipe or sweep it identically for all methods.
- **Variance amnesia in finetuning.** End-to-end finetuning reintroduces everything the frozen protocols try to exclude; the DINO authors switched to kNN precisely because finetuning numbers have large between-run variance (Caron et al., 2021). Report several seeds or use a frozen protocol.
- **Test-set leakage via pretraining data.** If unlabeled pretraining data contains near-duplicates of evaluation images, all downstream numbers inflate silently. Industrial pipelines therefore dedicate a data-processing step to removing near-duplicates of test data (Oquab et al., 2024); small academic reproductions rarely bother and rarely say so.
- **In-benchmark ceiling.** Reporting only ImageNet linear accuracy rewards invariance to exactly ImageNet's nuisances (centered objects, iconic views) and can rank a brittle specialist above a more transferable generalist. Cross-benchmark transfer (Places205, VOC, iNaturalist in the PIRL table (Misra and van der Maaten, 2020)) is the standard antidote.
- **Confusing the readout with the representation.** A low-accuracy linear probe on a well-clustered space is not automatically a bad representation — it may be a non-linearly-organized one; check a second protocol before concluding.

Research Frontier

When probes and representations disagree. The linear-probe protocol is under principled attack. Consider the masked autoencoder's numbers again: 84.9% top-1 after full finetuning of ViT-L on ImageNet in the course's MAE discussion (Lecture 03; He et al. (2022)), against 76.6% in the linear-probe column of the DINOv2 comparison (a larger MAE model, so the comparison is indicative rather than exact) — an 8-point spread no "representation quality" scalar can express, because the two numbers measure different objects: frozen geometry vs. trainable initialization. Deeper still, probe accuracy is only a *lower* bound on how much task information the features contain (a linear probe

finds only what is linearly readable, and never beats the Bayes rate of the features), and the training dynamics of the probe (label efficiency, robustness of the boundary) carry information a scalar accuracy discards. A complementary line of work scores features by information-theoretic criteria rather than by fitted probes — the KoLeo regularizer used inside DINOv2 is the Kozachenko–Leonenko entropy of the feature distribution estimated from nearest-neighbor distances (Sablayrolles et al., 2019; Oquab et al., 2024), usable in principle as an unsupervised readout of how uniformly features spread. And at system scale, "evaluation" stops being a number at all: DINOv3 is compared on dense probes across ADE20k, Cityscapes, PASCAL VOC, and depth benchmarks (Siméoni et al., 2025), and whether a representation transfers depends on properties — patch-level structure, robustness to off-distribution imagery — that a single classification probe cannot see. Chapter 16 assembles the evaluation-artifact checklist that current practice recommends.

1.8 How we got here: a short prehistory

Historical Note

The lineage starts long before 2012. None of this began with contrastive learning; the modern field assembled itself from five older threads, most of them older than the deep-learning boom.

Deciding what to keep — information theory (2000). The first thread is not a network at all but a principle: the information bottleneck of Tishby et al. (2000) formalized the trade-off this chapter stated as a motto — keep the information in $\boldsymbol{x}$ that is relevant to some target and discard the rest — as the optimization of $I(\boldsymbol{x};\boldsymbol{z})$ against $I(\boldsymbol{z};\boldsymbol{y})$. SSL methods do not compute these mutual informations directly, but every invariance-versus-covariance decision in the book (Chapter 8, and Chapter 15's explanations) is the information bottleneck expressed in algorithmic clothing: the augmentation family declares which factors are $\boldsymbol{y}$-irrelevant, and the objective squeezes them out.

Dimensionality reduction as representation learning. Principal component analysis (Jolliffe, 2002) is the oldest fully worked example of this chapter's thesis: from unlabeled data alone, compute a low-dimensional $\boldsymbol{z} = f(\boldsymbol{x})$ that preserves the directions of maximal variance, and read structure off the result. The classical toolkit of Lecture 02 and Chapter 2 — PCA, kNN, k-means (Steinhaus, 1956; Lloyd, 1982), DBSCAN (Ester et al., 1996), t-SNE (van der Maaten and Hinton, 2008) — doubles as the evaluation toolkit of modern SSL for exactly this reason: the question "is the space organized?" predates the networks that organize it.

Unsupervised pretraining of deep nets. Before ImageNet-scale supervision existed, the standard push to make deep nets learn features from unlabeled data was the autoencoder: compress $\boldsymbol{x}$ through a bottleneck and reconstruct $\hat{\boldsymbol{x}}$. When *denoising* replaced plain copying, Vincent et al. (2008, 2010) showed the learned features gain precisely the robust

invariance that modern masked-modeling methods (Chapters 5, 7, and 11) still rely on; the 2008 analysis already frames reconstruction of corrupted inputs as forcing the encoder to discover the structure of the data distribution.

Language context as free supervision. Predicting a word from its neighbors — at scale, on raw web text — had already proven, years before BERT, that context supplies free training signal from which usable structure (word analogies emerging as vector arithmetic) falls out; masked word prediction is simply the pretext task the transformer era made famous (Devlin et al., 2019), and ELMo's bidirectional language-model features (Peters et al., 2018) had already demonstrated the transfer pattern the whole field would adopt: pretrain self-supervised, attach cheap readouts.

Pairwise agreement. Forcing two views of one input to agree appears in Becker and Hinton's 1992 Imax work, which learned features by maximizing mutual information between outputs computed from adjacent patches of one image (Becker and Hinton, 1992), and in Siamese networks for signature verification (Bromley et al., 1993), whose deep-metric descendant (Chopra et al., 2005) is the direct ancestor of contrastive embedding losses — the exact skeleton of every joint-embedding method in Chapters 9–11.

What the deep era added was not the ideas but the instrumentation: modern backbones (LeCun et al., 1998), billion-scale unlabeled corpora, and — the subject of this chapter — evaluation protocols rigorous enough to make the progress of the ideas measurable.

1.9 Chapter summary

Strip this chapter to its bones and three sentences remain.

Why. Human labels are the slowest-growing, most expensive input to machine learning, and for most interesting problems — medicine, robotics, novel sensors, anything where the categories are not yet agreed — they are not merely costly but structurally unavailable. Unlabelled data, meanwhile, accumulates for free. Self-supervised learning is the discipline of learning from the abundant side of that ledger.

How. Replace the human in the training loop with a fixed rule that manufactures targets from the input itself (Figure 1.1). Train on those manufactured targets, then discard the machinery that solved the manufactured task and keep the middle: a frozen encoder $f_{\boldsymbol{\theta}}$ mapping inputs to vectors in which closeness means similarity. The rules people use split into three families — put the damage back, name the damage, ignore the damage — and the rest of Part II is those three families in detail.

How you know. Because the training objective deliberately excludes the task you care about, the training loss is not evidence of progress; a collapsed network achieves a perfect loss and encodes nothing. Progress is measured by attaching readouts of controlled capacity to the frozen features and reporting several of them (Figure 1.7). Each rung of that ladder

answers a different question, the rungs can disagree in rank, and a single number is never a description of a representation.

One more thing is worth naming explicitly before Part II, because it is the sentence the whole book turns on: *nothing in an unlabelled dataset says which differences matter.* A representation is a commitment about that, and the commitment is smuggled in through the choice of pretext rule — which crops count as "the same image", which corruptions must be undone, which transformations must be named. Designing a self-supervised method is therefore not primarily an architecture problem. It is the problem of stating, in code, which differences you are willing to throw away.

Vocabulary introduced in this chapter

Term	In one line
Label	An answer a person wrote for one example. The expensive thing.
Representation / embedding	The vector $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ a trained network produces before any final decision; the product of SSL.
Encoder / feature extractor	The network $f_{\boldsymbol{\theta}}$ that produces the representation; what you keep and ship.
Transfer	Reusing a network trained for one purpose on a different one.
Self-supervised learning	Supervised-style training on targets computed automatically from the input, with no human label.
Unsupervised learning	The older, wider category: any learning from unlabelled data, often with the unlabelled task as the goal itself.
Semi-supervised learning	Different in kind: real labels *for the target task* exist and are used alongside unlabelled data.
Pretext task	The manufactured job with free answers, used for training and then thrown away.
Downstream task	The real job, evaluated with human labels.
Pseudo-label	A target produced by the model's own earlier output (e.g. a cluster index) and then treated as if it were a label.
Augmentation	A transformation applied on purpose to an input — crop, colour jitter, mask, rotation. The vehicle for every pretext rule.
Invariance	A change the representation is required to *ignore*.
Covariance	A change the representation is required to *track*.
Collapse	The cheat where a network maps every input to the same vector: perfect agreement loss, zero information.
Frozen features	An encoder whose weights are no longer updated, so any downstream score is attributable to the representation.
Euclidean distance	$\|\boldsymbol{u} - \boldsymbol{v}\|_2$: straight-line distance; sensitive to vector length.
Cosine similarity	$\boldsymbol{u}^\top \boldsymbol{v}/(\|\boldsymbol{u}\|\|\boldsymbol{v}\|)$: angle only, length discarded (Eq. (1.2)).
Hypersphere / L_2 normalisation	Rescaling every feature to unit length; afterwards Euclidean and cosine give the same neighbour ordering.
Direct inspection	Looking at the space (t-SNE, PCA, attention maps). Cheap, revealing, never a number.
kNN evaluation	Classify by the majority vote of nearest frozen features. No parameters trained; measures local purity.
Linear probe	One trained linear layer on frozen features. Measures global linear separability.
Clustering evaluation	Cluster the frozen features and score the agreement with true classes. Tests whether categories exist before any label.
Fine-tuning	Unfreeze everything and train on labels. Measures the weights as a *starting point*, not the frozen geometry.
Disentangledness	Separate factors of variation occupy separate directions of Z.
Hierarchy	The representation exposes structure at several levels — texture, part, object, scene.
Transferability	The representation helps on tasks and datasets the pretraining never saw. The criterion that actually gets measured.

Key takeaway — the one thing to carry into Chapter 2

A self-supervised method is two decisions, not one. The first is which rule manufactures the targets — that is the method, and it is what papers argue about. The second is which readout you use to judge the result — that is the measurement, and it silently decides which methods look good. Read every result in this book as a pair, and half the field's apparent contradictions dissolve.

Check Your Understanding

Level 1 (recall).

1. State the course's one-sentence thesis about what can serve as a learning signal, and name the method it was introduced with.
2. Define a pretext task and a downstream task, and give one example of each.
3. Which two distances on feature vectors does the book use, and which of them is invariant to vector magnitude?
4. In a linear probe, which parameters are trained and which are frozen?

Level 2 (comprehension).

5. Why does the SimSiam collapse experiment warn against monitoring the SSL loss the way one monitors supervised training loss?
6. A colleague says "our system is self-supervised because only 5% of our training data has labels." Identify the confusion and classify the system correctly.
7. Why does kNN have "low capacity in the head," and why is that a feature of the protocol rather than a bug?

Level 3 (application).

8. Using Equation (1.2), compute the cosine similarity between $(3, 4)$ and $(4, 3)$, then between $(3, 4)$ and $(6, 8)$; explain why the second pair's score does not reflect their distance.
9. You trained an SSL model. Its linear-probe accuracy rose 4 points but kNN accuracy fell 6. Sketch two different explanations and one extra measurement that would discriminate between them.
10. Order the downstream protocols from "most credit to the frozen features" to "most credit to the initialization" and justify the ordering.

Level 4 (analysis).

11. The DINOv2 linear-evaluation table shows MAE jumping from 49.4% (kNN) to 76.6% (linear). What does this tell you about the *geometry* of MAE's embedding space that neither number alone reveals?
12. Section 1.3 lists three criteria for good representations. For each, name one evaluation protocol from Section 1.7 that operationalizes it and one failure mode in which the protocol could mislead.

Level 5 (synthesis).

13. Design an evaluation battery for a self-supervised encoder intended for robot perception in novel terrain (dense depth, traversability, few-shot object retrieval, no ImageNet-like labels). Defend each choice against the failure modes in this chapter.
14. Equation for unit vectors, $\|\boldsymbol{u} - \boldsymbol{v}\|_2^2 = 2 - 2\ \mathrm{sim}(\boldsymbol{u}, \boldsymbol{v})$, says Euclidean and cosine agree on a hypersphere. Under what training objectives would you expect features to sit on a hypersphere, and when might they drift off it, restoring the distinction?

Solution sketches.

1. "Anything better than chance can be exploited as a learning signal"; introduced with DeepCluster (Caron et al., 2018).
2. Pretext: synthetic task with free labels from the input itself, e.g., predict image rotation (Gidaris et al., 2018). Downstream: the real labeled application, e.g., ImageNet classification.
3. Euclidean and cosine; cosine is magnitude-invariant.
4. Trained: the linear readout layer only. Frozen: all of $f_{\boldsymbol{\theta}}$.
5. Because the objective does not model the downstream task: collapse gives *low* loss and useless features (kNN $\approx 0\%$), while slow loss decay with stop-gradient accompanies kNN climbing to $\approx 50\%$ (Chen and He, 2021).
6. Confuses self-supervised with semi-supervised: labels for the target task were used in training, so the system is semi-supervised (Lee, 2013).
7. kNN only memorizes and votes; no parameters are fit, so success must come from the geometry of Z. High capacity would let the readout fix geometric deficits, confounding representation quality with readout power.
8. sim $= 24/25 = 0.96$ for the first pair; exactly 1 for the second, since cosine ignores scale — the vectors differ in magnitude, not direction.
9. Explanation A (real geometry change): classes became more linearly separable overall but less pure locally — each class stretched into an elongated region whose nearest neighbours often belong to other classes. Explanation B (a distance artefact): a few high-variance nuisance directions grew and now dominate the distances kNN uses, while the linear probe simply learns to ignore them. Discriminate by re-running kNN after whitening or PCA-truncating the features: if kNN recovers, it was B; if not, check A with a clustering evaluation (NMI) or t-SNE.
10. kNN, clustering eval (pure geometry) → linear probe (linear separability) → heads (mild nonlinearity) → few-shot finetune → full finetune (initialization).
11. kNN tests nearest-neighbor purity: MAE's low kNN but high linear score means local neighborhoods are semantically noisier while classes are still nearly linearly separable — geometry organized at large scale, ragged at small scale (He et al., 2022; Oquab et al., 2024).

12. Disentangledness ↔ linear probe (linear readout needs separable factors; misled if factors are linearly entangled but clustered). Hierarchy ↔ dense/patch probes or attention maps (misled when global probes mask missing local structure). Transferability ↔ cross-dataset probes (Misra and van der Maaten, 2020) (misled by in-benchmark-only evaluation).
13. Open-ended; a good battery: frozen patch-level probes for depth, kNN/retrieval for objects, clustering on terrain descriptors, few-shot finetune for traversability, plus direct inspection of attention maps — with a fixed probe recipe and held-out terrain for leakage control.
14. Objectives with explicit normalization (contrastive losses on ℓ_2-normalized outputs (Chen et al., 2020a; He et al., 2020)) pin features to the sphere; reconstruction losses do not, so magnitudes can encode confidence and Euclidean/cosine diverge again.

Concept Map — how this chapter connects

Backward. The notation ($\boldsymbol{x}$, $\boldsymbol{z}$, sim, $\mathcal{L}$) comes from the Notation table in the front matter; Chapter 2 supplies the machinery used here in passing — kNN, distances, k-means, DBSCAN, PCA and t-SNE — which this chapter needed only as evaluation tools.

Forward. The three families of Section 1.5 are the spine of Part II: reconstruction (Chapters 4–7, starting from autoencoders (Vincent et al., 2008; Kingma and Welling, 2014) and masked modeling (He et al., 2022)), pretext prediction and invariance-vs-covariance (Chapter 8, RotNet (Gidaris et al., 2018), PIRL (Misra and van der Maaten, 2020)), joint-embedding alignment (Chapters 9–11: CPC (van den Oord et al., 2018b), SimCLR (Chen et al., 2020a), MoCo (He et al., 2020), BYOL (Grill et al., 2020), SimSiam (Chen and He, 2021), Barlow Twins (Zbontar et al., 2021), DINO and successors (Caron et al., 2021; Oquab et al., 2024; Siméoni et al., 2025), BERT (Devlin et al., 2019)). The evaluation taxonomy returns as a chapter-length checklist in Chapter 16, and the better-than-chance thesis becomes the deep clustering loop of Chapter 3 (Caron et al., 2018).

Lateral. The historynote's threads — denoising (Vincent et al., 2008), context prediction in language, coherence across views (Becker and Hinton, 1992) — each reappear as a full chapter; this chapter is the map they all fit into.

CHAPTER 2

The Mathematical and Machine Learning Toolkit

> The essence of many self-supervised learning methods: anything better than chance can be exploited as a learning signal.
>
> Lecture 02

In this chapter

2.1 How to Read This Chapter 35
2.2 Neural Networks in Fifteen Minutes 36 • The multi-layer perceptron • Convolutional networks • Transformers and the vision transformer • Embeddings
2.3 Loss Functions ..... 42 • Softmax and cross-entropy • Mean squared error • A preview of the contrastive family • Distances as pseudo-labels: self-distillation
2.4 Similarity and Distance . 46 • Euclidean, dot product, cosine • Life on the hypersphere • The concentration of distances
2.5 Optimization 52 • Stochastic gradient descent and momentum • Exponential moving averages of the weights • Stop-gradient: supervising without being supervised • Temperature
2.6 Information Theory 58 • Entropy and its relatives • Jensen's inequality • Mutual information
2.7 Classical Retrieval and Clustering Toolkit 62 • Nearest neighbors • Fast search: KD-trees • Fast search: locality-sensitive hashing • Clustering without labels • k-means • DBSCAN and OPTICS: clustering by density • Dimensionality reduction: PCA and t-SNE • From clustering to pseudo-labels
2.8 Summary and Outlook 79

Chapter roadmap

Before you start. Nothing beyond Chapter 1: the idea that a network learns a representation, and that representations are judged by how well they transfer. Every tool below is built from first principles.

The idea in one sentence. This chapter is a toolbox, not a story — it collects, in one place and with numbers you can check by hand, every mathematical object the rest of the book will pick up and use without re-explaining it.

What we cover.

- **Networks that make embeddings** (Section 2.2): the MLP, the CNN, and attention — three different ways to turn raw pixels or tokens into a vector.
- **Losses** (Section 2.3): softmax and cross-entropy above all, plus MSE — how "make this close to that" becomes a number gradient descent can shrink.
- **Similarity and geometry** (Section 2.4): cosine vs. Euclidean distance, and why almost every embedding in this book ends up living on a sphere.
- **Optimization** (Section 2.5): SGD and momentum move the weights; exponential moving averages and stop-gradient decide what a self-supervised loss is even allowed to chase.
- **Information theory** (Section 2.6): entropy, cross-entropy, KL, Jensen's inequality, mutual information — the vocabulary for "how much does this representation actually tell you."
- **Classical retrieval and clustering** (Section 2.7): kNN, KD-trees, LSH, k-means, DBSCAN — what you can already do with an embedding, no training required.

Read this chapter out of order. It is written as a reference, not a narrative, and nothing past Section 2.2 depends on having read the sections before it. If a later chapter sends you here for one term — "what is temperature again?" — jump straight to that subsection and come back for the rest whenever you need it. Four entries on the list above are worth flagging before you meet them, because they read like footnotes the first time through and are anything but: *temperature* (Section 2.5.4), the *EMA teacher* (Section 2.5.2), *stop-gradient* (Section 2.5.3), and *hypersphere normalization* (Section 2.4.2). Nothing about them looks dramatic here — a knob that rescales some numbers, a slow-moving copy of the weights, a gradient switched off on one wire, a vector divided by its own length. In Chapters 10 and 11 these four stop being footnotes: assembled together, they do not merely *help* DINO and its relatives, they *are* the method. If you remember one sentence from this roadmap, make it this one.

Afterwards you will be able to read any equation in Chapters 3–13 and name every symbol in it on sight; explain in plain words why a self-supervised loss needs a stop-gradient or an EMA teacher rather than just training both sides freely; compute a softmax, a cross-entropy, a cosine similarity, and one round of k-means by hand; and, for any retrieval or clustering claim you meet later, say which of kNN, a KD-tree, LSH, k-means, or DBSCAN is doing the work and what it silently assumes about the data.

2.1 How to Read This Chapter

A self-supervised learning paper is a compression of many ideas into a few equations. When DINO (Chapter 11) matches two softmax distributions with a cross-entropy, normalizes its features onto a hypersphere, evaluates them with a k-nearest-neighbor classifier, and updates

its teacher by an exponential moving average, it silently invokes six different tools. When a VAE (Chapter 4) writes down a lower bound on a log-likelihood, it invokes Jensen's inequality and the Kullback–Leibler divergence. When DeepCluster (Chapter 3) alternates k-means with backpropagation, it invokes the entire classical clustering toolbox. This chapter develops exactly those tools — and nothing else.

The guiding principle is *just-in-time prerequisites*. Each section is written to be read before the chapter that needs it and only goes as deep as that need demands: this is a toolkit, not a survey. Topics not exercised by the course — batch normalization internals, second-order optimization, kernel methods — are deliberately omitted. Experienced readers may skip ahead, consulting the cross-references scattered through the rest of the book whenever a tool is invoked.

Section N references elsewhere are to lectures of the source course when they help map concepts back to the slide deck; chapter references are to this book.

2.2 Neural Networks in Fifteen Minutes

Every method in this book is a neural network that maps a raw input $\boldsymbol{x} \in \mathbb{R}^{C\times H\times W}$ (or a token sequence, or a sensor reading) to a representation, and the entire field can be read as a history of three ways to organize that mapping: the multi-layer perceptron (MLP), the convolutional network (CNN), and the vision transformer (ViT). We need the mechanics of each, because later chapters put them to very different uses — CNNs are clustered in Chapter 3, ViTs are masked in Chapter 5, and MLPs appear throughout as projection and prediction heads.

2.2.1 The multi-layer perceptron

In plain words

One MLP layer does exactly three things to a vector, in this order. **Mix**: multiply by a matrix W, so every output number becomes a weighted combination of *every* input number. **Shift**: add a bias $\boldsymbol{b}$, so the layer can move its output away from zero even when the input is zero. **Bend**: apply a nonlinearity to each coordinate — ReLU just means "keep positive numbers, zero out negative ones." Mix and shift alone are a straight line, stretched and moved; stack a hundred straight lines and the result is still one straight line, because the composition of affine maps is itself affine. The bend is the only step that lets the network draw a *curve*, and it is the only reason depth buys you anything beyond a single wide layer.

An MLP is a stack of layers, each doing "mix, shift, bend". One layer:

$$\boldsymbol{h} = \sigma(W\,\boldsymbol{u} + \boldsymbol{b}),$$

where $\boldsymbol{u}$ is the input vector, the weight matrix W does the mixing, the bias $\boldsymbol{b}$ does the shifting, and σ bends each number separately. The usual bend is the ReLU, $\max(0, a)$; the sigmoid $1/(1+e^{-a})$, which squashes into $(0,1)$, is used where a probability is needed. Stacking layers gives $f_{\boldsymbol{\theta}}(\boldsymbol{u}) = \sigma(W_L \cdots \sigma(W_1 \boldsymbol{u} + \boldsymbol{b}_1) \cdots)$, where $\boldsymbol{\theta}$ means "all the weights and biases".

Two facts worth knowing:

- **Without the bends, depth is pointless**: a stack of linear maps is just one linear map.
- **With them, a network can approximate essentially any continuous function**, and deep networks do it with far fewer units than wide shallow ones.

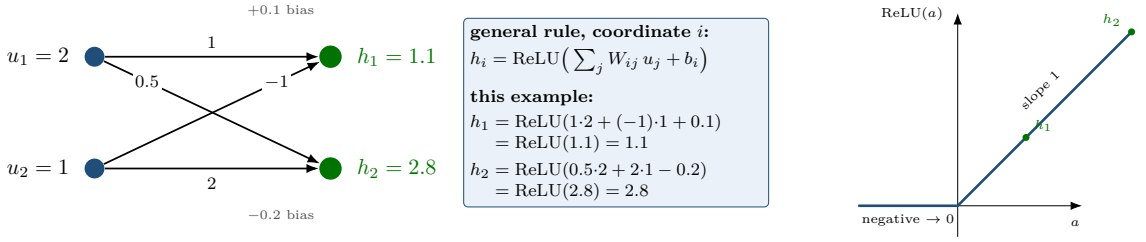
Worked Example — A tiny forward pass

Let $W = \begin{pmatrix} 1 & -1 \\ 0.5 & 2 \end{pmatrix}$, $\boldsymbol{b} = \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix}$, $\boldsymbol{u} = (2,\ 1)^\top$, and $\sigma = \mathrm{ReLU}$. Then

$$W\boldsymbol{u} + \boldsymbol{b} = \begin{pmatrix} 1 \cdot 2 + (-1) \cdot 1 + 0.1 \\ 0.5 \cdot 2 + 2 \cdot 1 - 0.2 \end{pmatrix} = \begin{pmatrix} 1.1 \\ 2.8 \end{pmatrix}, \qquad \boldsymbol{h} = \mathrm{ReLU} \begin{pmatrix} 1.1 \\ 2.8 \end{pmatrix} = \begin{pmatrix} 1.1 \\ 2.8 \end{pmatrix}.$$

Every neuron is one dot product plus a shift; everything else in deep learning is repetition and composition of this operation.

Figure 2.1 draws the worked example above as a small circuit: two input neurons, four weighted wires, two output neurons, and the ReLU that bends the result.



(a) neurons and wires **(b) what the wires compute** **(c) the bend itself**

Figure 2.1. One MLP layer, worked in full. **(a)** Two input neurons feed two output neurons through four weighted wires (the entries of W) plus a per-output bias; this is the mix-and-shift half. **(b)** The same computation spelled out arithmetically — every wire is one multiplication, every neuron is one addition. **(c)** The bend: ReLU passes positive sums through unchanged and clips negative sums to zero, which is what turns a straight-line mixer into something that can build a curve.

How to read it: Trace $u_1 = 2$ along its two wires in (a); each wire multiplies by its own weight, and each output neuron adds up everything that arrives, then applies panel (c)'s kink. Both outputs here land on the positive side of the kink, so ReLU changes nothing — flip the sign of the bias in panel (b) and watch panel (c) explain why an output could be clipped to 0 instead.

Training computes the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ of some loss by the chain rule, layer by layer from output back to input — *backpropagation*, the algorithm whose modern resurgence traces to the error-backpropagation lineage surveyed in LeCun et al. (1998). In this book we never write out these gradients by hand; what matters is that every module we introduce — softmax losses, similarity measures, even stop-gradient masks — is defined so that this chain rule can run through it.

2.2.2 Convolutional networks

In plain words

Why not just feed an image into an MLP? Because an MLP has no notion of "next to." A 256×256 colour image is $256 \times 256 \times 3 \approx 200{,}000$ numbers, and a fully connected first layer would need its own weight from *every one* of them to every neuron — and worse, it would have to learn "there might be an edge in the top-left corner" and "there might be an edge in the bottom-right corner" as two completely unrelated facts about the world, each needing its own weights and its own training examples to discover. A CNN fixes both problems with one design decision: look at only a small patch at a time (*locality*), and reuse the *same* small filter at every position (*weight sharing*). An edge detector learned in the top-left corner is, for free, an edge detector everywhere else in the image too.

Two ideas make a convolutional network suited to images:

- **Locality.** Each output looks only at a small window of the input (its *receptive field*), because nearby pixels are what matter for edges and textures.
- **Weight sharing.** The *same* small filter $\boldsymbol{w} \in \mathbb{R}^{k\times k}$ is used at every position:

$$(I * \boldsymbol{w})_{u,v} \;=\; \sum_{i=1}^{k}\sum_{j=1}^{k} I_{u+i,\,v+j}\, w_{i,j},$$

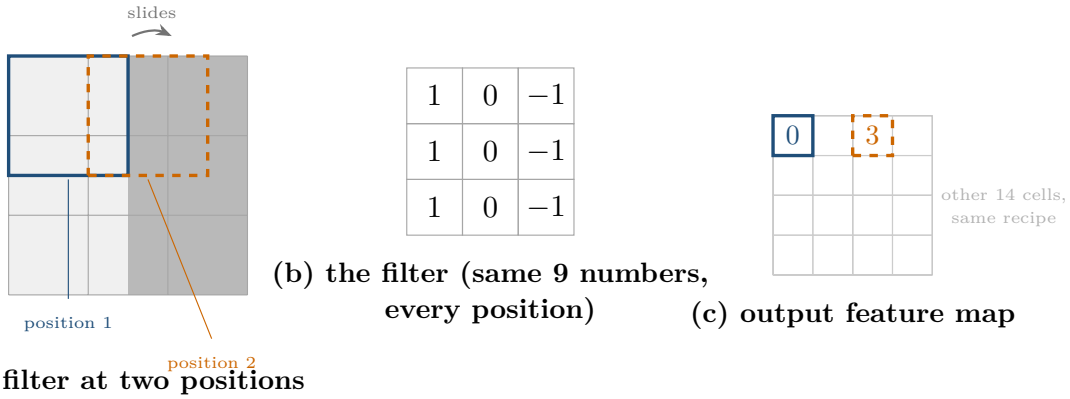
  followed by a bias and a non-linearity.

Weight sharing has two payoffs. Shifting the input shifts the output by the same amount, so a cat is detected wherever it is. And the number of weights depends only on the filter size, not on the image size.

Stack such layers with downsampling in between and each layer sees a larger part of the image: early layers find edges and textures, later layers combine information from across the whole picture. LeNet (LeCun et al., 1998) showed this works for digits; ResNets, whose skip connections $\boldsymbol{h} \mapsto \boldsymbol{h} + f(\boldsymbol{h})$ let networks be hundreds of layers deep, are the backbones used in Chapters 3 and 8. (A plain fully connected layer, by contrast, has no notion of "nearby" at all: it would treat a cat photo and the same pixels shuffled at random as equally sensible.)

For this book one property matters above all: run an image through a CNN, drop the final classification layer, and the penultimate activations form a *spatial feature map* — one descriptor vector per location. That map is the raw material of nearest-neighbor retrieval (Section 2.7), of clustering hybrids (Chapter 3), and of the feature-correspondence methods (STEGO, Chapter 3; DIFT, Chapter 7).

Figure 2.2 makes weight sharing concrete: the same 3×3 filter slides to two different positions on a tiny image, and only the *content* under the filter changes what comes out.



(a) filter at two positions **(b) the filter (same 9 numbers, every position)** **(c) output feature map**

Figure 2.2. Weight sharing, worked on a 6×6 toy image whose left three columns are bright (1) and right three are dark (0), with the vertical-edge filter $\begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$. **(a)** The filter at *position 1* sits entirely inside the bright region; at *position 2* it straddles the bright/dark boundary. Sliding it means nothing about the 9 numbers in the filter changes — only which 9 pixels they are multiplied against. **(b)** The filter itself: the same 9 numbers at every position in the image, which is exactly what "weight sharing" means and why it cuts the parameter count from one weight per pixel pair down to just the 9 filter weights. **(c)** The output feature map: position 1's patch is flat, so $1{\cdot}1 + 0{\cdot}1 + (-1){\cdot}1 = 0$ summed over 3 rows is 0 — the filter reports "no edge here." Position 2's patch straddles the edge, $1{\cdot}1 + 0{\cdot}0 + (-1){\cdot}0 = 1$ summed over 3 rows is 3 — the filter fires. The other 14 output cells are computed the same way and omitted only for space.

How to read it: Cover panel (b) and try to predict whether position 1 or position 2 gives the bigger output before reading panel (c) — the filter is a vertical-edge detector, so it should fire where the image actually changes, not where it is flat. That prediction, made correctly without doing the arithmetic, is the entire intuition for what a convolution filter does.

2.2.3 Transformers and the vision transformer

In plain words

Every token asks a question, advertises an answer, and carries a payload. The **query** is what a token is currently looking for. The **key** is what a token broadcasts about itself, for other tokens' queries to be compared against. The **value** is the actual content a token hands over once it is chosen. Attention compares one token's query to every token's key (a dot product: large when they point the same way), turns those scores into weights

with a softmax, and mixes the *values* by those weights. The contrast with a CNN's filter is exact: a convolution's weights are fixed once training ends and are identical at every position; attention's weights are recomputed from scratch on every forward pass, from the actual content in front of it. A filter cannot decide, mid-inference, to ignore its neighbour; a token can.

A transformer layer replaces fixed local connectivity with *attention*, a content-dependent routing of information between tokens. Given a set of token vectors arranged as rows of $X \in \mathbb{R}^{n \times d}$, self-attention projects them three ways — queries $Q = XW^Q$, keys $K = XW^K$, values $V = XW^V$ with learned matrices $W^Q, W^K \in \mathbb{R}^{d \times d_k}$, $W^V \in \mathbb{R}^{d \times d_v}$ — and computes

$$\text{Attn}(X) \;=\; \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \tag{2.1}$$

where the softmax (Equation (2.2)) is applied to each row separately. So each output token is a weighted average of the value vectors, with weights set by how well its query matches each key. The division by $\sqrt{d_k}$ keeps the scores from growing too large as vectors get longer.

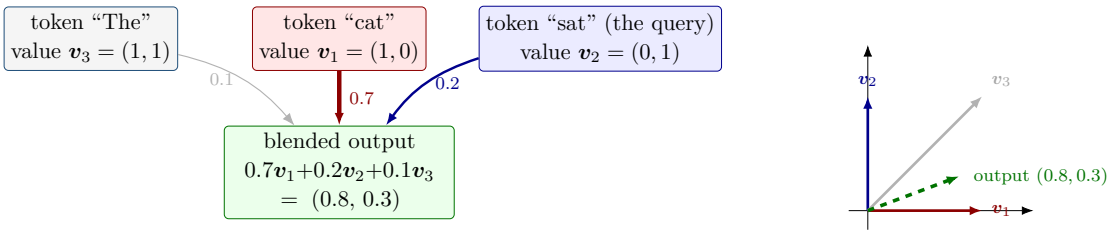
Two consequences. The whole thing is differentiable, so it trains with ordinary backpropagation. And every token can look at every other token, so even the first layer sees the whole input. A full *transformer block* wraps attention with a skip connection, layer normalisation and a small MLP per token; Vaswani et al. (2017) showed that stacking such blocks is enough for state-of-the-art sequence models.

Figure 2.3 follows one query token through the whole computation with concrete numbers, from raw attention weights to a single blended output vector.

The vision transformer (ViT). The ViT (Dosovitskiy et al., 2021) applies attention to images with very few changes:

1. Cut the image into 16×16 pixel patches.
2. Turn each patch into a vector (a *token*) with one linear layer.
3. Add a learned *position* vector to each token. Attention by itself ignores order, so without this the network could not tell the top-left patch from the bottom-right.
4. Put one extra learned token, `[CLS]`, at the front.
5. Run an ordinary transformer.

You get two kinds of output: the `[CLS]` token's output describes the *whole image*, and each patch's output describes *that spot*. Both are used later: MAE (Chapter 5) trains ViTs by hiding most patches, DINO (Chapter 11) trains them so their attention picks out objects, and STEGO (Chapter 3) uses the per-patch outputs.



(a) one query, three weights **(b) the blend, drawn**

Figure 2.3. Attention on one query token, worked in full. **(a)** The query token "sat" compares itself against three tokens' keys (not drawn — only the resulting weights are shown) and gets back weights $0.7, 0.2, 0.1$; the output is the values mixed by exactly those weights. **(b)** The same computation drawn as vectors: the output (dashed) is a single point inside the triangle with corners $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$, pulled strongly toward $\boldsymbol{v}_1$ because it carries the biggest weight. No fixed filter could produce this — the weights $(0.7, 0.2, 0.1)$ were computed from the content of "sat," "cat," and "The" at this specific forward pass, and would be different for a different sentence.

How to read it: Panel (b) is the whole idea in one picture: attention output is always a point inside *the shape spanned by the value vectors, never outside it, and which point depends entirely on the weights in panel (a). Push the* 0.7 *toward* 1 *and the output slides onto* $\boldsymbol{v}_1$ *exactly; spread the weights evenly and it slides toward the centroid of all three.*

2.2.4 Embeddings

All three architectures above end at the same conceptual place: a vector that summarizes an input.

Key Concept — Embedding

An *embedding* is a learned map $f_{\boldsymbol{\theta}} : \boldsymbol{x} \mapsto \boldsymbol{z} \in \mathbb{R}^{d_z}$ that replaces a raw, high-dimensional, weakly-structured input with a vector in which *semantic relationships become geometric relationships*: inputs that should be treated alike lie close together, and the directions and distances in $\mathbb{R}^{d_z}$ carry meaning. Self-supervised learning is, in large part, the enterprise of obtaining good embeddings without labels.

Two rules about embeddings come up again and again:

1. **The distance is part of the design.** Whether "close" means a small distance or a small angle depends on how the embedding was trained (Section 2.4).
2. **The loss decides what the embedding keeps.** An embedding is not a compressed photo of the input. It keeps whatever the training objective found useful and drops the rest. Section 2.6 makes this precise, and Chapter 5 (MAE) shows that choosing what to *hide* during training is as much a design decision as choosing what to rebuild.

Key takeaway — the three architectures, in one line each

An **MLP** mixes everything with everything — maximum freedom, no assumptions, far too many parameters for an image. A **CNN** assumes that what matters is local and that it matters the same way everywhere — which is true of images and is why CNNs work. A **transformer** assumes nothing about *which* parts should interact and decides at run time from the content — more expensive, more flexible, and the reason ViTs dominate the second half of this book.

2.3 Loss Functions

A loss function converts a vague intention ("make similar things close") into a scalar that backpropagation can minimize. Five losses cover essentially the entire course.

2.3.1 Softmax and cross-entropy

In plain words

Softmax turns a list of arbitrary scores into something that behaves like a probability, in two steps. First, **exponentiate** every score: this makes every number positive (a negative score no longer breaks anything) and it exaggerates differences, since e^x grows fast. Second, **divide** each exponentiated score by their sum: this forces the numbers to add up to exactly one, so they can be read as "how much probability mass goes to each option." Cross-entropy then charges you for how much probability you put on the *correct* answer, using $-\log(\cdot)$ as the price list: hand over probability 1.0 on the right answer and you pay nothing; hand over 0.5 and you pay about 0.69; hand over a stingy 0.01 and you pay 4.6. The price list is steep on purpose — it is cheap to be a little unsure and ruinous to be confidently wrong.

Given logits $\boldsymbol{s} = (s_1, \dots, s_K) \in \mathbb{R}^K$, the softmax turns them into a probability distribution over K classes,

$$\text{softmax}_i(\boldsymbol{s}) \;=\; \frac{e^{s_i}}{\sum_{j=1}^{K} e^{s_j}}, \tag{2.2}$$

where e^{s_i} makes each term positive and the denominator normalizes the vector to sum to one. For a target class y, the cross-entropy loss is the negative log-probability of the correct class,

$$\mathcal{L}_{\text{CE}}(y, \boldsymbol{s}) \;=\; -\log \text{softmax}_y(\boldsymbol{s}), \tag{2.3}$$

which equals the information-theoretic cross-entropy between the one-hot target and the model distribution (Section 2.6 justifies the name). Its gradient has the memorably simple form $\partial \mathcal{L}_{\text{CE}} / \partial s_i = \text{softmax}_i(\boldsymbol{s}) - \mathbb{1}[i{=}y]$: prediction minus target.

Worked Example — Numerical softmax and cross-entropy

Let $\boldsymbol{s} = (2.0, 1.0, 0.1)$. Exponentiating gives $(e^{2.0}, e^{1.0}, e^{0.1}) \approx (7.39, 2.72, 1.11)$ with sum ≈ 11.21, so $\text{softmax}(\boldsymbol{s}) \approx (0.659, 0.242, 0.099)$. If the target is class 1, $\mathcal{L}_{\text{CE}} = -\log 0.659 \approx 0.417$; if the target were class 3, $\mathcal{L}_{\text{CE}} = -\log 0.099 \approx 2.31$. The loss is small exactly when the model concentrates mass on the right class. Figure 2.4 draws every bar in this calculation, for both targets.

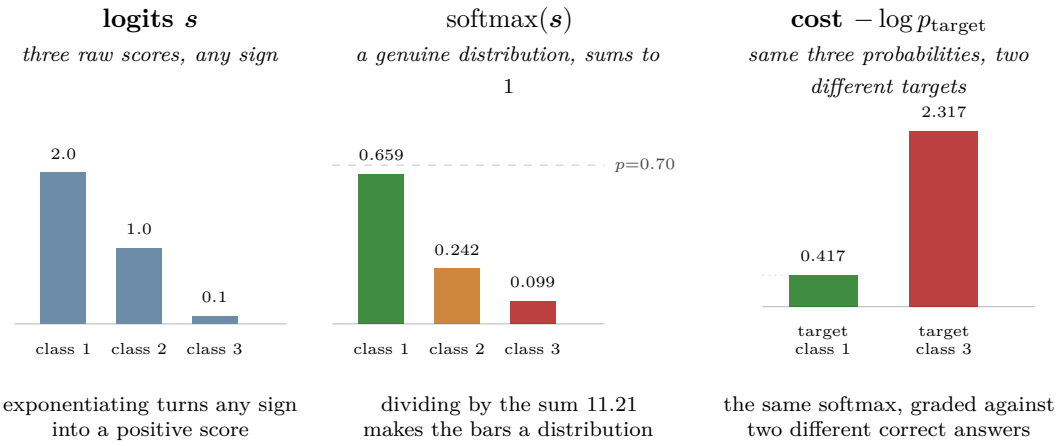
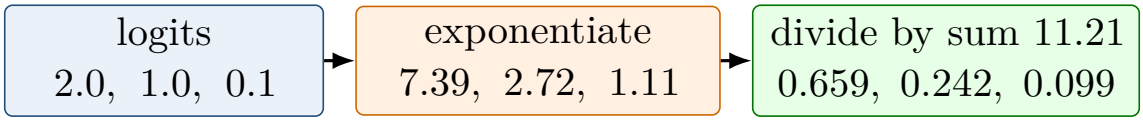


Figure 2.4. The numerical softmax and cross-entropy example, drawn as bars. **Left:** the three raw logits. **Middle:** after exponentiating and normalizing, the same three numbers as a probability distribution — note they now sum to 1, unlike the logits. **Right:** the cost $-\log p_{\text{target}}$ actually paid, for the same predicted distribution, under two different targets: when the target is the class the model favoured (class 1, p=0.659) the cost is a cheap 0.417; when the target is the class the model buried (class 3, p=0.099) the cost jumps to 2.317, over five times as much.

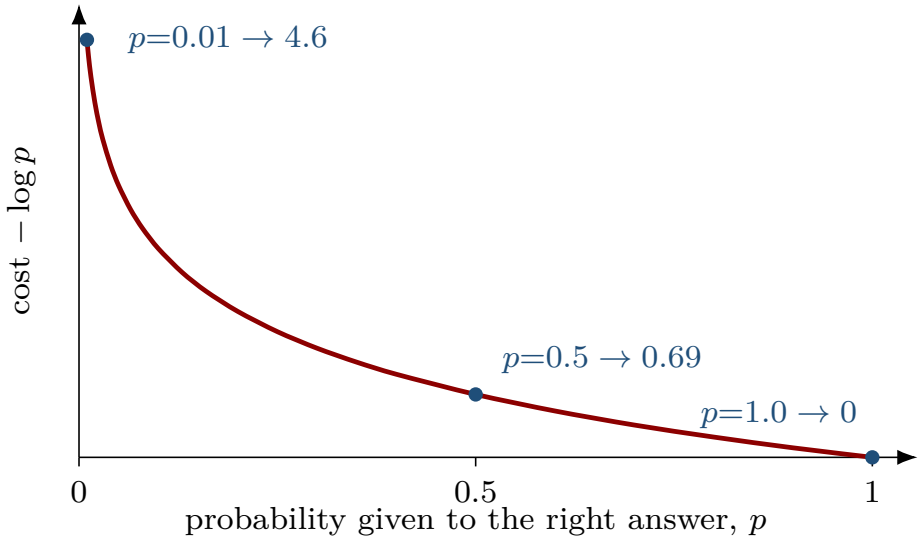
How to read it: Compare the height of the two right-hand bars. Both come from exactly the same softmax output — nothing about the model changed between them. Only the target changed, and $-\log p$ punishes a low probability steeply: the class-3 bar is more than five times taller for a probability that is only about a sixth as large.

Figure 2.5 draws both steps: turning the three logits above into probabilities, and the price list that cross-entropy charges for whatever probability you handed the correct class.

Cross-entropy is the supervised workhorse — and the self-supervised one too, because every method in this book manufactures targets and then treats them exactly like labels: DeepCluster trains on cluster IDs (Caron et al., 2018), RotNet on rotation indices (Gidaris et al., 2018), DINO on a teacher's softened predictions (Caron et al., 2021). The loss is identical; only the provenance of y changes.



(a) exponentiate, then divide



(b) cross-entropy's price list, $-\log p$

Figure 2.5. From logits to a cost. **(a)** The worked example above: exponentiate to make every score positive, then divide by the sum so the three numbers become a genuine probability distribution. **(b)** The cross-entropy cost of whatever probability you assigned the correct class — steep near $p = 0$, essentially flat near $p = 1$. Confident-and-correct is nearly free; confident-and-wrong is expensive, and unboundedly so as $p \to 0$.

How to read it: Panel (b) is why training pushes probabilities toward the extremes rather than settling for "pretty sure": the cost curve is convex and blows up near zero, so a gradient step always prefers shaving a little more off a merely-medium probability over leaving it alone.

2.3.2 Mean squared error

Mean squared error. For numeric targets the default loss is the average squared difference:

$$\mathcal{L}_{\text{MSE}}(\boldsymbol{x}, \hat{\boldsymbol{x}}) \;=\; \frac{1}{d}\sum_{i=1}^{d}(x_i - \hat{x}_i)^2 \;=\; \frac{1}{d}\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2,$$

where $\hat{\boldsymbol{x}}$ is the prediction and d the number of outputs.

Why squared error, rather than something else? It is what you get from a probabilistic assumption: if you model the target as "the prediction plus Gaussian noise", then making the data as likely as possible is exactly the same as minimising squared error. The book uses this twice: autoencoders (Chapter 4) get their reconstruction loss from it, and flow matching (Chapter 7) is the same Gaussian regression applied to a velocity.

2.3.3 A preview of the contrastive family

The triplet loss. The first widely used embedding loss of the deep-learning era (Schroff et al., 2015) works on three examples at a time: an *anchor* $\boldsymbol{z}_a$, a *positive* $\boldsymbol{z}_p$ (same class) and a *negative* $\boldsymbol{z}_n$ (different class):

$$\mathcal{L}_{\text{triplet}} \;=\; \big[\,\|\boldsymbol{z}_a - \boldsymbol{z}_p\|^2 - \|\boldsymbol{z}_a - \boldsymbol{z}_n\|^2 + \alpha\,\big]_+.$$

In words: the positive should be closer to the anchor than the negative is, by at least a margin α. Once it is, the loss is zero ($[\,\cdot\,]_+ = \max(0, \cdot)$ clips it) and nothing more is asked.

Chapter 9 develops two upgrades that lead to the losses used today:

1. **One positive against many negatives at once**, using a softmax instead of distances. This leads through the N-pair loss (Sohn, 2016) to InfoNCE (van den Oord et al., 2018b):

$$\mathcal{L}_{\text{NCE}} \;=\; -\log \frac{e^{\text{sim}(\boldsymbol{q},\boldsymbol{k}_+)/\tau}}{\sum_{j=0}^{K} e^{\text{sim}(\boldsymbol{q},\boldsymbol{k}_j)/\tau}},$$

   which is simply the cross-entropy of Eq. (2.3), with similarities as the scores.
2. **No negatives at all**, preventing collapse instead with stop-gradients, predictors and moving-average teachers (Section 2.5.2; Chapters 10 and 11).

The one-line summary: *contrastive learning is cross-entropy applied to similarities.* That is why this chapter treats softmax, temperature and cosine similarity as one toolkit.

2.3.4 Distances as pseudo-labels: self-distillation

When the targets are not classes but the outputs of another (frozen or slowly-updated) network, the same two losses reappear in distributional form: match predicted probabilities to target probabilities with cross-entropy between distributions (or, equivalently, minimize

a KL divergence, Section 2.6), or match features with MSE. Knowledge distillation (Hinton et al., 2015) and DINO-style self-distillation (Chapter 11) live here. Nothing new is needed — only the discipline to ask, for every self-supervised loss in this book: *what plays the role of* $\boldsymbol{s}$*, and what plays the role of* y*?*

Key takeaway — every loss in this book is one of two things

Either you are comparing a predicted *distribution* against a target distribution — that is cross-entropy, and it does not care whether the target came from a human, a clustering algorithm or another network — or you are comparing a predicted *vector* against a target vector, and that is squared error. When you meet a new self-supervised loss, the useful question is not "what kind of loss is this?" but *what plays the role of the prediction, and what plays the role of the target?*

2.4 Similarity and Distance

Everything downstream of an embedding — retrieval, clustering, contrastive losses — is a statement about geometry, and the geometry is set by a choice of similarity. This section fixes the three candidates and, more importantly, the one fact that resolves most practical confusion: modern SSL features are normalized to live on the unit hypersphere, where the three candidates collapse into a single one.

Before any of the candidates comes a prior question that is easy to forget in a book full of trained networks: *what carries the semantics, the direction of the embedding or its length?* A reusable picture — drawn from document retrieval — pins it down. Represent each of two books by its vector of word counts. Book A is a ten-page essay on space exploration; Book B is a thousand-page encyclopedia that contains a major section on space. Judged by Euclidean distance on raw counts, the two are wildly different, because Book B's vector is roughly a hundred times longer on every relevant coordinate: the distance is dominated by the *portion size*, not the content. Judged by the angle between the two vectors — the *recipe* of which words appear in which proportions — they point in nearly the same direction. Most embedding semantics in this book are recipes, not portions: two crops of the same dog share a direction, up to nuisance factors (scale, brightness, compression strength) that mostly rescale the vector. That is the design judgment underlying everything that follows.

2.4.1 Euclidean, dot product, cosine

In plain words

Cosine and Euclidean distance do not have to agree, and when they disagree, only one of them is measuring what you actually care about. Take a vector $\boldsymbol{u}$, a shrunk copy $\boldsymbol{v}$ of it that points the exact same direction, and a third vector $\boldsymbol{w}$ that points somewhere genuinely different. Cosine only ever looks at direction, so it insists $\boldsymbol{v}$ is a perfect match

for $\boldsymbol{u}$ — correctly, since they point the same way. Euclidean distance looks at raw position, and if $\boldsymbol{v}$ was shrunk enough, its *position* may have moved farther from $\boldsymbol{u}$ than $\boldsymbol{w}$'s did, even though $\boldsymbol{w}$ points somewhere completely different. Which verdict is "right" is not a property of the vectors; it is a property of what their *length* is supposed to mean in your problem. Figure 2.6 makes this concrete with numbers, and shows the fix used throughout this book: rescale every vector to length one and the disagreement disappears entirely.

Three ways to compare two vectors $\boldsymbol{u}, \boldsymbol{v}$:

Euclidean distance — straight-line distance, affected by both direction and length:

$$d_2(\boldsymbol{u}, \boldsymbol{v}) \;=\; \|\boldsymbol{u} - \boldsymbol{v}\| \;=\; \sqrt{\textstyle\sum_i (u_i - v_i)^2}.$$

Dot product — $\boldsymbol{u}^\top \boldsymbol{v} = \|\boldsymbol{u}\|\,\|\boldsymbol{v}\| \cos\theta$, where θ is the angle between them. It mixes angle and length: double the length of $\boldsymbol{u}$ and every similarity involving $\boldsymbol{u}$ doubles, though no direction changed.

Cosine similarity — the dot product with the lengths divided out, leaving only the angle:

$$\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v}) \;=\; \frac{\boldsymbol{u}^\top \boldsymbol{v}}{\|\boldsymbol{u}\|\,\|\boldsymbol{v}\|} \;=\; \cos\theta \;\in\; [-1, 1].$$

It is 1 for vectors pointing the same way, 0 for perpendicular ones, -1 for opposite ones, and does not change if you stretch either vector.

Which one is right? That depends on whether the *length* of an embedding means anything — and that is decided by how it was trained.

Common Confusion

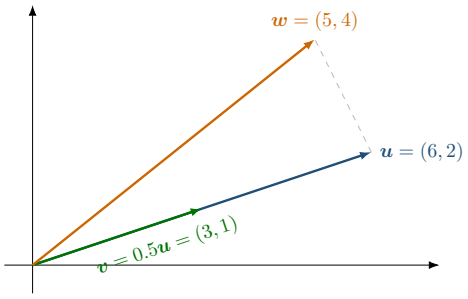
"Cosine distance" is not cosine similarity. Lecture slides sometimes label the formula $\boldsymbol{u}^\top \boldsymbol{v}/(\|\boldsymbol{u}\|\|\boldsymbol{v}\|)$ as "Cosine Distance" (Lecture 02, slide 4 does). It is cosine *similarity*: bounded in $[-1, 1]$, maximal when the vectors point in the same direction. The distance is $d_{\cos}(\boldsymbol{u}, \boldsymbol{v}) = 1 - \mathrm{sim}(\boldsymbol{u}, \boldsymbol{v})$, which vanishes at alignment. Mixing the two flips the direction of every ranking you compute; when a pipeline "maximizes cosine distance" something has been mislabeled.

The strength of cosine, and the weakness of Euclidean. Cosine ignores length. That is exactly right when length reflects something irrelevant — the book-length example above, or, in images, brightness, which makes every raw activation bigger.

Euclidean distance is at the mercy of units. Suppose feature z_1 ranges from 0 to 10,000 (a raw pixel count) and feature z_2 from 0 to 1 (a ratio). In $\|\boldsymbol{u} - \boldsymbol{v}\|^2 = \sum_i (u_i - v_i)^2$ the first term is millions of times bigger than the second, so the distance effectively ignores z_2 entirely. Multiply any one feature by S and its influence on every squared distance grows

by S^2. Cosine divides magnitudes out, and counting metrics such as Hamming (how many positions differ) and Jaccard (how much two sets overlap) never look at magnitudes at all.

Practical rule: never use Euclidean distance on features whose scales you have not normalised — and if the training loss worked with angles, use cosine.

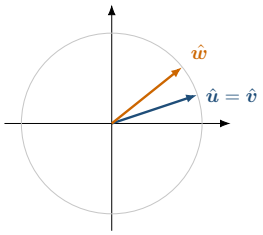


(a) who is closer to u?

	vs. v	vs. w
cosine sim	1.00	0.938
squared dist. d^2	10	5

cosine picks v (same direction); Euclidean picks w (nearer in raw position) — **opposite verdicts.**

(b) the two metrics disagree



(c) normalized: they agree

Figure 2.6. Cosine and Euclidean distance can rank the same three points oppositely, and normalization is the fix. **(a)** v is a half-length copy of u — same direction, smaller magnitude — while w points somewhere genuinely different; the dashed segments are the actual Euclidean gaps. **(b)** The verdicts: cosine says v is more similar to u (it never left u's direction), Euclidean says w is closer (it never moved as far from u's position). Both cannot be "the" answer, and nothing about u, v, w themselves says which one is — it depends on what you built the vectors to mean. **(c)** Divide every vector by its own length: $\hat{u}$ and $\hat{v}$ land on the *same* point of the unit circle (they always pointed the same way), $\hat{w}$ lands elsewhere, and now cosine and Euclidean distance rank them identically, by the identity $d^2 = 2(1 - \text{sim})$ proved just below.

How to read it: Look at panel (a) first: the dashed line from u to w really is shorter than the one from u to v, even though v is the vector that never changed direction. Panel (c) does not move w at all — it removes the very idea of length that made v look far away in (a).

2.4.2 Life on the hypersphere

Why the choice usually stops mattering. Most losses in this book make every feature vector have length 1 — some explicitly, by dividing by the length ($z \mapsto z/\|z\|$) (Chen et al., 2020a; He et al., 2020); some implicitly, because their softmax only responds to directions (DINO). Once both lengths are 1:

$$\|u - v\|^2 \;=\; \|u\|^2 + \|v\|^2 - 2\,u^\top v \;=\; 2(1 - \text{sim}(u, v)).$$

Distance goes down exactly when cosine similarity goes up. So ranking neighbours by Euclidean distance, by dot product or by cosine gives *exactly the same order*. When Lecture 02 (slides 4–5) says "use cosine, because the features lie on a sphere", this equation is the reason. Figure 2.6(c) shows it: on the unit circle the two measures can no longer disagree.

Key Concept — Normalization

L_2-normalization projects an embedding onto the unit hypersphere: $\boldsymbol{z} \leftarrow \boldsymbol{z}/\|\boldsymbol{z}\|$. After normalization, Euclidean distance, dot product, and cosine similarity induce the same neighbor ordering, the embedding's norm carries no information, and one temperature τ uniformly controls the sharpness of every softmax built on these features (Section 2.5.4).

Common Confusion

Cosine vs. Euclidean. Beginners treat the choice of metric as a matter of taste. It is not: it is a claim about the embedding. *If* features are unit-normalized — explicitly, or because the loss only reads directions — the two metrics are the same ranking in disguise, and arguing about them is wasted effort. *If* features are unnormalized (raw CNN activations, classical descriptors), they genuinely disagree: Euclidean distance is dominated by the largest coordinates and by overall activation magnitude, which often encodes nuisance factors such as brightness or contrast rather than content. The lecture's practical rule follows immediately: check how the network was trained; if the loss optimizes an angle, retrieve with cosine; DINO descriptors, for instance, are compared by cosine similarity (Lecture 02, slide 5), and Chapter 11 explains why the training objective makes that the right choice.

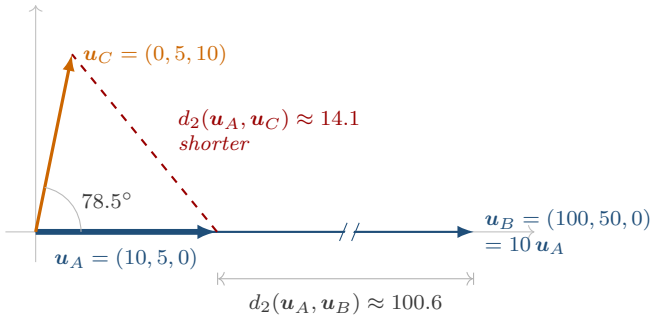
Worked Example — Cosine vs. Euclidean on unit-circle points

Take three points on the unit circle: $\boldsymbol{u} = (1, 0)$, $\boldsymbol{v} = (0, 1)$, $\boldsymbol{w} = (\cos 60°, \sin 60°) = (0.5, 0.866)$. Cosine similarities: $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v}) = 0$ ($\theta = 90°$) and $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{w}) = 0.5$ ($\theta = 60°$), so $\boldsymbol{w}$ ranks closer to $\boldsymbol{u}$. Squared Euclidean: $\|\boldsymbol{u} - \boldsymbol{v}\|^2 = 2(1 - 0) = 2$ and $\|\boldsymbol{u} - \boldsymbol{w}\|^2 = 2(1 - 0.5) = 1$: same ordering, as the identity promises, and the Euclidean gap is a rescaling of the cosine gap. Contrast with the unnormalized twin of $\boldsymbol{v}$: scaling it to $\boldsymbol{v}' = 2\boldsymbol{v}$ leaves cosine unchanged, $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v}') = 0$, but inflates every dot product with $\boldsymbol{v}'$ twofold and moves Euclidean distances: an unnormalized embedding conflates "how confident" with "how similar." Figure 2.8 draws both pairs on the circle, with the identity itself showing up as a right triangle.

Worked Example — Recipe vs. portion: the music-taste example

Represent listening habits as a vector over genres [Rock, Pop, Jazz]: $\boldsymbol{u}_A = (10, 5, 0)$, $\boldsymbol{u}_B = (100, 50, 0)$, $\boldsymbol{u}_C = (0, 5, 10)$. Euclidean distance declares $d_2(\boldsymbol{u}_A, \boldsymbol{u}_B) = \sqrt{90^2 + 45^2} \approx 100.6$ and $d_2(\boldsymbol{u}_A, \boldsymbol{u}_C) = \sqrt{10^2 + 10^2} \approx 14.1$: user A is said to be far closer to C — mostly jazz, a genre A never plays — than to B, whose tastes are *identical in proportion* ($\boldsymbol{u}_B = 10\,\boldsymbol{u}_A$) and differ only in volume. Cosine similarity reverses the silly verdict exactly where intuition says it should: $\mathrm{sim}(\boldsymbol{u}_A, \boldsymbol{u}_B) = 1$ (zero angle; same recipe) and $\mathrm{sim}(\boldsymbol{u}_A, \boldsymbol{u}_C) = 25/(\sqrt{125} \cdot \sqrt{125}) = 0.2$. Whenever the semantics live in proportions and the magnitude is bookkeeping, cosine — or equivalently the unit-sphere geometry of

Section 2.4.2 — is the only honest metric. Figure 2.7 draws all three listeners as vectors from the origin.



drawn in the plane of u_A and u_C; u_B is shortened to fit (break marks)

Same direction (u_A, u_B): sim = 1 **Different direction (u_A, u_C):** sim = 0.2

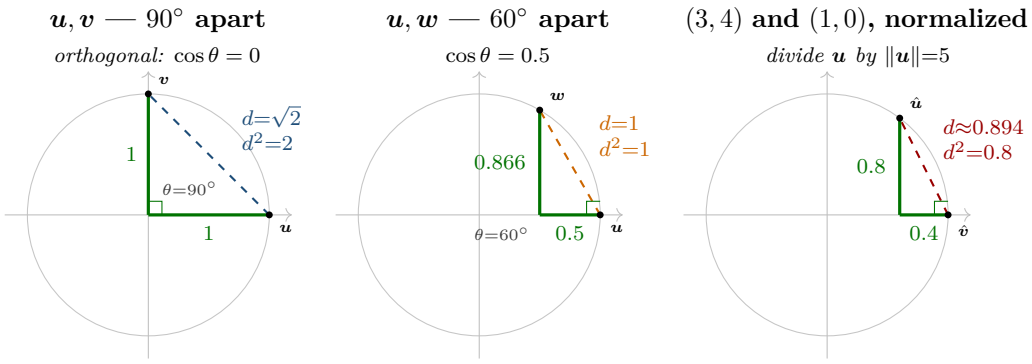
Figure 2.7. Recipe versus portion, drawn. u_A and u_B are exact scalar multiples — same direction, ten times the length — so cosine correctly declares them identical (sim = 1) while Euclidean distance is fooled by the length gap into calling them the most dissimilar pair in the picture ($d_2 \approx 100.6$, dashed gray). u_C points somewhere genuinely different from u_A ($\theta \approx 78.5°$, sim = 0.2), yet happens to sit closer to u_A in raw position ($d_2 \approx 14.1$, dashed red) purely because u_B's length is so much larger. Only the direction encodes taste; the length is just how much the listener happened to play.

How to read it: Look at u_A and u_B first: they are the same ray from the origin, just walked further out — "same recipe, bigger portion." Then notice that this is exactly the pair Euclidean distance rates as most *dissimilar, because it can only see how far apart the tips are, never that one tip is a multiple of the other.*

Worked Example — Verifying $d^2 = 2(1 - \text{sim})$ numerically

Let $u = (3, 4)^\top$ and $v = (1, 0)^\top$. Raw quantities: $u^\top v = 3$, $\|u\| = 5$, $\|v\| = 1$, so $\text{sim}(u, v) = 3/5 = 0.6$ while $d_2(u, v) = \sqrt{(3-1)^2 + (4-0)^2} = \sqrt{20} \approx 4.47$: the two metrics disagree wildly, as they may on unnormalized features. Normalizing gives $\hat{u} = (0.6, 0.8)$, $\hat{v} = (1, 0)$, hence $d_2^2(\hat{u}, \hat{v}) = (0.6 - 1)^2 + (0.8 - 0)^2 = 0.16 + 0.64 = 0.8$ — exactly $2(1 - 0.6)$, as promised. The third panel of Figure 2.8 draws this pair after normalizing.

Why does normalization help learning at all? Empirically, forcing all features onto a fixed-radius sphere removes a trivial degree of freedom (grow all logits by growing the norm) and makes the temperature τ a meaningful global scale; spreading mass over a sphere also acts as a weak whitening, discouraging the collapse where every input maps to a single point. Chapter 15 discusses how far this intuition is actually supported.



legs $1, 1$: the right angle sits at the origin itself

legs $1-\cos\theta$, $\sin\theta$: dropping a perpendicular from $\boldsymbol{w}$

same construction, after $\hat{\boldsymbol{u}} = \boldsymbol{u}/5$: legs $0.4, 0.8$

Figure 2.8. The identity $d^2 = 2(1 - \text{sim})$ made visible as a right triangle, for both worked examples above. In each panel the green legs are $1 - \cos\theta$ (horizontal, from dropping a perpendicular from the second point onto the first) and $\sin\theta$ (vertical); Pythagoras gives $d^2 = (1 - \cos\theta)^2 + \sin^2\theta = 2 - 2\cos\theta$, exactly the dashed hypotenuse squared. **Left, middle:** the unit-circle example, $\boldsymbol{u}$ against $\boldsymbol{v}$ (90°) and against $\boldsymbol{w}$ (60°) — the closer angle gives the shorter hypotenuse, so cosine and Euclidean agree on which point is nearer. **Right:** the numerical-verification example, $\boldsymbol{u} = (3, 4)$ and $\boldsymbol{v} = (1, 0)$, normalized onto the same circle before the identical triangle is drawn; the identity does not care where the original vectors came from, only that they end up on the circle.

How to read it: In every panel, follow the two green legs to the dashed hypotenuse: their squares always sum to the printed d^2. That triangle is *the proof of $d^2 = 2(1 - \text{sim})$ — not an illustration of it, the actual Pythagorean argument, specialized to a chord of the unit circle.*

Normalisation also blocks a cheat. Suppose a loss rewards two views of the same image for being *close* in ordinary distance. The network can win trivially: output the zero vector for every input. Every pair is then at distance 0 — perfect score, useless features.

Forcing features onto the unit sphere removes that cheat: the zero vector is not on the sphere. A different cheat remains — putting *every* image at the same point on the sphere, which is the collapse Chapter 10 fights — but at least shrinking the vectors is no longer a shortcut to zero loss.

2.4.3 The concentration of distances

Why "nearest" stops meaning much in high dimensions. The lecture warns (Lecture 02, slides 4–5) that nearest-neighbour search becomes unreliable when vectors have many dimensions. The reason is called *concentration of distances*.

Here is the intuition. A squared distance is a sum over all d coordinates. With hundreds of coordinates, that sum behaves like an average of many random terms — and averages of many random terms all come out close to the same value. So as d grows, *every* point ends up

at roughly the same distance from the query. Formally, for the nearest and farthest points at distances $d_{\min}$ and $d_{\max}$,

$$\lim_{d\to\infty} \frac{d_{\max} - d_{\min}}{d_{\min}} \;=\; 0,$$

under mild assumptions: the gap between "nearest" and "farthest" shrinks to nothing compared with the distances themselves.

What that means in practice: the "nearest" neighbour becomes a lottery. Nudge the query slightly and a different point wins. Small noise spread across hundreds of irrelevant coordinates outweighs the signal in the few meaningful ones. (On an exam, the expected answer to "why is nearest neighbour unreliable in high dimensions?" is exactly: *because distances concentrate*.)

Two consequences for this book. It is another reason to compare *directions* (cosine) rather than raw positions: directions can stay informative when raw distances have all become alike. And it explains why the search structures of Sections 2.7.2 and 2.7.3 perform much worse in high dimensions than their textbook costs suggest.

Key takeaway — normalise, then stop worrying about the metric

> Cosine similarity compares directions; Euclidean distance compares positions. They disagree whenever length carries information. Almost every method in this book normalises its features to unit length first — and once you do that the two measures become the same thing, related exactly by $\|\boldsymbol{u} - \boldsymbol{v}\|_2^2 = 2 - 2\,\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v})$. The question stops mattering precisely because everyone removed the thing it depended on.

2.5 Optimization

Self-supervised losses are non-convex and trained on millions of examples; the optimizer is not a detail. Four ideas suffice for the whole book.

2.5.1 Stochastic gradient descent and momentum

Stochastic gradient descent. Computing the exact gradient of the loss over all N training examples at every step would be far too slow. SGD uses a small random *minibatch* of B examples instead:

$$\boldsymbol{\theta}_{t+1} \;=\; \boldsymbol{\theta}_t \;-\; \eta\, \boldsymbol{g}_t, \qquad \boldsymbol{g}_t = \frac{1}{B} \textstyle\sum_{i\in\mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \ell_i(\boldsymbol{\theta}_t),$$

where η is the learning rate (the step size). The minibatch gradient is a noisy estimate of the true one, correct on average; its variance shrinks like $1/B$ as the batch grows (so the typical error shrinks like $1/\sqrt{B}$).

A common confusion: when a method "needs large batches" — SimCLR, with batches in the thousands, is the famous case (Chapter 9) — the reason is usually the *loss*, not the

optimiser. SimCLR's loss uses the other images in the batch as negatives, so a bigger batch means more negatives. Less gradient noise is only a side benefit.

Momentum damps the zig-zag of noisy gradients by accumulating an exponential average of past gradients into a velocity:

$$\boldsymbol{g}_t \leftarrow \beta\, \boldsymbol{g}_{t-1} + \nabla_{\boldsymbol{\theta}} \ell, \qquad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \boldsymbol{g}_t,$$

with $\beta \approx 0.9$ typical. Adam additionally normalizes each coordinate by a running estimate of its squared gradient scale, making the effective step roughly scale-free per parameter; Adam and its weight-decayed variant AdamW are the default for the transformer backbones of Chapters 5 and 11, while SGD+momentum remains standard for CNN backbones.

2.5.2 Exponential moving averages of the weights

In plain words

An EMA is a running average that leans heavily on its own past. At every step, the new shadow value is mostly (say 90%) the old shadow value, plus a small dose (the remaining 10%) of whatever the live network just became. One noisy update barely moves it; a trend sustained over many updates eventually drags it along, just slowly and smoothly. Two numbers in this section look alike and are *not* the same thing — do not let the shared digits below fool you. The decay m is a dial you choose: how much of the past survives each step, typically very close to 1. A weight's *starting value* is just whatever number happens to be there before training begins; it has nothing to do with m and could have been anything.

A second use of exponential averaging — this time of *parameters rather than gradients* — is one of the quiet workhorses of SSL.

Key Concept — Exponential moving average of weights

Given a network $f_{\boldsymbol{\theta}}$ trained by gradient descent, an EMA maintains a shadow copy $\boldsymbol{\theta}_T$ that chases the live weights:

$$\boldsymbol{\theta}_T \leftarrow m\, \boldsymbol{\theta}_T + (1 - m)\, \boldsymbol{\theta}, \tag{2.4}$$

with decay $m \in [0.9, 0.9999]$ applied after every update. The shadow is a low-pass filter over the training trajectory: it lags, but it averages away gradient noise, and evaluating with $\boldsymbol{\theta}_T$ is often slightly better than with $\boldsymbol{\theta}$.

Worked Example — One step of EMA

A weight takes values $1.00, 1.04, 0.98$ over three updates, with decay $m = 0.9$ and shadow starting at $\boldsymbol{\theta}_T = 0.9$. **These two 0.9s are unrelated numbers that happen to share**

a decimal: m is the dial that sets how much of the past survives each step, while $\boldsymbol{\theta}_T = 0.9$ is just this example's arbitrary starting point — the update rule would run identically if the shadow had started at 3.7 instead. Applying Eq. (2.4) three times, keeping four significant figures:

$$\boldsymbol{\theta}_T: \quad 0.9 \xrightarrow{0.9(0.9)+0.1(1.00)} 0.910 \xrightarrow{0.9(0.910)+0.1(1.04)} 0.923 \xrightarrow{0.9(0.923)+0.1(0.98)} 0.9287.$$

The shadow moves roughly ten times slower than the live weight and suppresses the oscillation between 1.00, 1.04, and 0.98 — it never visits any of those three values, only their smoothed trend.

Figure 2.9 extends the same idea from three updates to fourteen: a jagged live ("student") trajectory and its EMA ("teacher") shadow, plotted on the same axes.

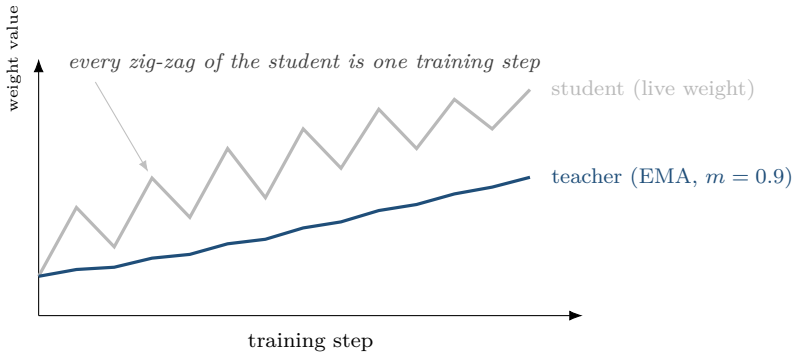


Figure 2.9. An exponential moving average is a low-pass filter, drawn over fourteen steps instead of three. The student (gray) swings up and down every step; the teacher (blue), computed from the same Eq. (2.4) with $m = 0.9$, never sees any individual swing — it only ever moves toward the student's *recent average*, so it traces a smooth curve that lags visibly behind and never matches the student's instantaneous value.

How to read it: Pick any single downward zig-zag in the gray curve and check that the blue curve barely reacts to it — that non-reaction is *the noise-averaging Section 2.5.2 keeps promising, made visible. The permanent gap between the two curves at the right edge is the price: the teacher is always a few steps out of date.*

Where the EMA shows up. Twice, in two roles:

- **MoCo** (He et al., 2020): the EMA copy is a *momentum encoder* that produces the negative examples. Because it changes slowly, negatives made at different times remain comparable (Chapter 9).
- **DINO** (Caron et al., 2021): the EMA copy is the *teacher*. The student learns to match it, even though the teacher is nothing but the student's own recent past (Chapter 11).

Why this stops the network collapsing to a constant is subtle; roughly, the teacher is a slowly moving target the student keeps chasing, so neither settles. Chapters 10 and 15 look at the explanations.

Two ways to think about it:

- **As a smoothing filter.** The step-to-step wobble of SGD averages out; the long-term trend passes through. The teacher is always a slightly older, smoother version of the student.
- **As "taking turns", made continuous.** DeepCluster alternates in discrete rounds: freeze the features and cluster, then train (Section 2.7.5). The EMA teacher replaces those rounds with a continuous delay that plays the same stabilising role.

Learning note from study sessions

The collapse-prevention mechanism that finally made this click in study sessions was hearing the EMA teacher described as a "brake" on Chapter 3's pseudo-labeling loop. DeepCluster's known danger — the network reinforces whatever the current clustering says, right or wrong — is softened wherever the model chases a moving average of its own recent past rather than a hard refit of its own current output. Same instinct, same failure suppressed, different decade of machinery.

When you meet "prediction without collapse is impossible" claims later in the book, hold on to this physical picture: a lag in the target prevents gradient descent from dragging both sides of the matching equation to the same degenerate point simultaneously.

2.5.3 Stop-gradient: supervising without being supervised

In plain words

Imagine two copies of a network looking at two crops of the same photo, and a loss that rewards them for agreeing. If both copies are free to change, the cheapest way to satisfy that loss is not to learn anything about photos: both copies can just collapse to outputting the same constant, always, regardless of the input, and agreement is perfect. Stop-gradient is the patch: freeze one copy's output as a fixed target for this step — compute it normally, but tell backpropagation "do not touch whatever produced this number." Now only the other copy is allowed to move, so the only way left to reduce the loss is to actually make that copy's prediction resemble the (fixed, meaningful) target. The shortcut of dragging both sides to a mutual constant is no longer available to either side, because one side cannot move at all.

Key Concept — Stop-gradient

The operator $\mathrm{sg}(\cdot)$ is the identity in the forward pass and zero in the backward pass: $\mathrm{sg}(x) = x$, $\partial\ \mathrm{sg}(x)/\partial x = 0$. Applying it to one branch of a loss makes that branch a *fixed target*: gradients reshape the other branch to match it, but never move the target itself.

Stop-gradient converts "A predicts B" from a symmetric relation — which backpropagation could satisfy by dragging A and B together into a single collapsed point — into an asymmetric teacher–student relation. SimSiam (Chen and He, 2021) is the cleanest demonstration: remove the sg and the model collapses instantly; keep it and the model learns rival-quality features with no negatives at all (Chapter 10). The EMA teacher of DINO plays the same role through architecture rather than an explicit operator. Throughout the book, when you see sg, read it as: *this side of the equation is the label.*

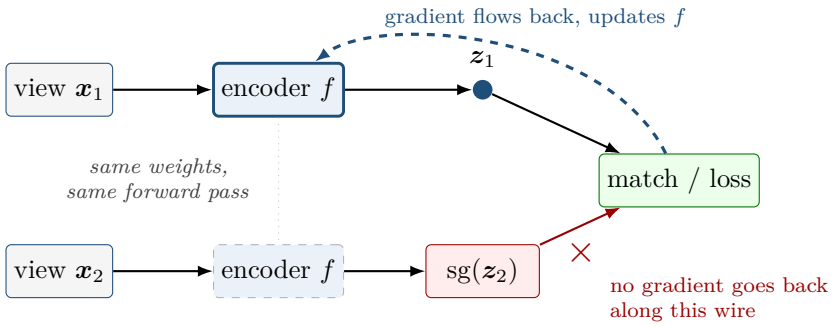


Figure 2.10. Stop-gradient, drawn as a blocked wire. Both branches run the *identical* forward computation (dotted line: shared weights f). The top branch's output $\boldsymbol{z}_1$ feeds the loss normally, and the loss's gradient flows all the way back (dashed blue arc) to update f. The bottom branch's output is wrapped in $\mathrm{sg}(\cdot)$ before it reaches the loss: the forward value $\boldsymbol{z}_2$ is used exactly as computed, but no gradient arrow leaves the loss along that wire (red $\times$) — this step's pull on f comes entirely through the top branch.

How to read it: Count the dashed arrows: there is exactly one, and it points at e_1, not e_2. That asymmetry is the whole mechanism. Erase the sg *box and a second dashed arrow would appear from* `match` *back to e_2 as well — and the cheapest way left to satisfy the loss becomes collapsing both branches to the same constant output, since two free variables being pulled together can always meet in the middle.*

2.5.4 Temperature

In plain words

Temperature is one knob that squeezes or stretches a softmax without ever changing which option wins. Divide every score by a *small* τ (say 0.25) before the softmax, and the biggest score gets amplified relative to the rest, so the distribution becomes almost all-or-nothing on the top choice. Divide by a *large* τ (say 5), and the scores get squashed toward each other, so the distribution spreads out toward treating every option as nearly

equally likely. What never changes, at any τ, is the *ranking*: whichever score was biggest stays biggest. Temperature controls confidence, not opinion.

Dividing the scores by a temperature $\tau > 0$ before the softmax, $\text{softmax}_i(\boldsymbol{s}/\tau) = e^{s_i/\tau} / \sum_j e^{s_j/\tau}$, reshapes the distribution but never changes the ranking:

- $\tau \to 0$: all probability goes to the top score — a hard choice;
- $\tau \to \infty$: everything becomes equally likely.

The same knob is turned for opposite reasons in different chapters:

- **Contrastive losses (Chapter 9):** a small τ makes the loss focus on the *hardest* negatives — the ones most similar to the positive.
- **Distillation (Chapter 12):** a *large* τ softens the teacher's output so the student can see how likely the teacher found the *wrong* classes — the "dark knowledge" of Hinton et al. (2015).
- **DINO (Chapter 11):** a *small* teacher τ makes the teacher confident, giving the student a sharp target.

Figure 2.11 shows one set of four scores pushed through three different temperatures, with every probability computed exactly (verified numerically, not sketched).

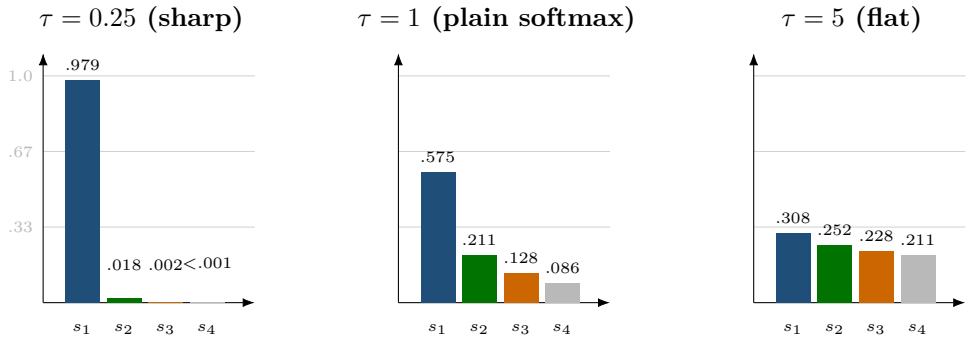


Figure 2.11. The same four scores $\boldsymbol{s} = (2.0, 1.0, 0.5, 0.1)$ pushed through $\text{softmax}(\boldsymbol{s}/\tau)$ at three temperatures (probabilities computed exactly, not sketched). At $\tau = 0.25$ nearly all the mass sits on s_1; at $\tau = 1$ the plain softmax already favors s_1 but leaves real mass on the rest; at $\tau = 5$ the four bars are almost level. In every panel the bars are ordered $s_1 > s_2 > s_3 > s_4$, left to right, exactly as the raw scores were — temperature never reorders them, it only redistributes how much probability separates first place from last.

How to read it: Read the three panels as one flip-book: the tallest bar shrinks from .979 to .575 to .308 while the shortest grows from under .001 to .086 to .211 — confidence draining out of first place and into the rest as τ grows, with the ranking frozen throughout.

Key takeaway — the four tools that look boring and are not

Most of this chapter is standard machinery. Four items are not, because later they stop being implementation details and become the method itself: **temperature** (Chapters 9 and 11 turn it *down* for sharp targets, Chapter 12 turns it *up* to expose dark knowledge), the **EMA teacher** (Chapters 10 and 11), the **stop-gradient** (Chapter 10 shows that removing it destroys training completely), and **hypersphere normalisation** (without it, half the losses in Chapters 9–11 are not even well posed).

2.6 Information Theory

Three quantities — entropy, cross-entropy, Kullback–Leibler divergence — and one derived quantity, mutual information, recur constantly: the VAE's objective is a rearranged KL (Chapter 4), InfoNCE is a lower bound on mutual information (Chapter 9), and the "what should a representation discard?" question is an information question (Chapters 5 and 15).

2.6.1 Entropy and its relatives

In plain words

Entropy measures how surprised you should expect to be. If you already know the answer for certain, there is nothing left to learn — zero surprise, zero entropy. If every outcome is equally likely, you are maximally in the dark, and entropy is as large as it can get for that many outcomes. Cross-entropy asks a related but different question: if you trust distribution q but the world actually follows p, how surprised are you *on average*? If q happens to equal p, cross-entropy equals entropy — you pay exactly for your irreducible uncertainty about p, no more. If q is wrong, you pay extra, and that extra is exactly the KL divergence: the surcharge for betting on the wrong distribution.

Entropy. For a random variable X with distribution p,

$$H(X) \;=\; -\sum_{x} p(x)\,\log p(x)$$

measures *how surprised you expect to be* by an outcome. It is largest when every outcome is equally likely, and zero when one outcome is certain. (Equivalently: the average number of bits needed to write down an outcome with the best possible code.)

Cross-entropy. If outcomes really follow p but you bet according to a model q, your expected surprise is

$$H(p, q) \;=\; -\sum_{x} p(x)\,\log q(x).$$

The classification loss Eq. (2.3) is exactly this, with p putting all its weight on the correct class.

KL divergence. The *extra* surprise caused by betting on q instead of the truth p is the *Kullback–Leibler divergence*

$$D_{\mathrm{KL}}(p \,\|\, q) \;=\; \sum_x p(x)\, \log \frac{p(x)}{q(x)} \;=\; H(p,q) - H(p), \tag{2.5}$$

which satisfies $D_{\mathrm{KL}}(p\|q) \geq 0$ with equality iff $p = q$ (Gibbs' inequality), is *asymmetric* — $D_{\mathrm{KL}}(p\|q) \neq D_{\mathrm{KL}}(q\|p)$ in general, which is why the direction of the KL in a VAE matters (Chapter 4) — and is not a metric. Minimizing $H(p,q)$ over q with p fixed is the same as minimizing $D_{\mathrm{KL}}(p\|q)$, since $H(p)$ is constant; classification training is therefore "matching distributions" in the strict sense of Eq. (2.5).

Figure 2.12 draws three distributions over four outcomes and their entropies, all computed exactly from $H(X) = -\sum_x p(x) \log p(x)$ in nats (natural log, this book's convention throughout).

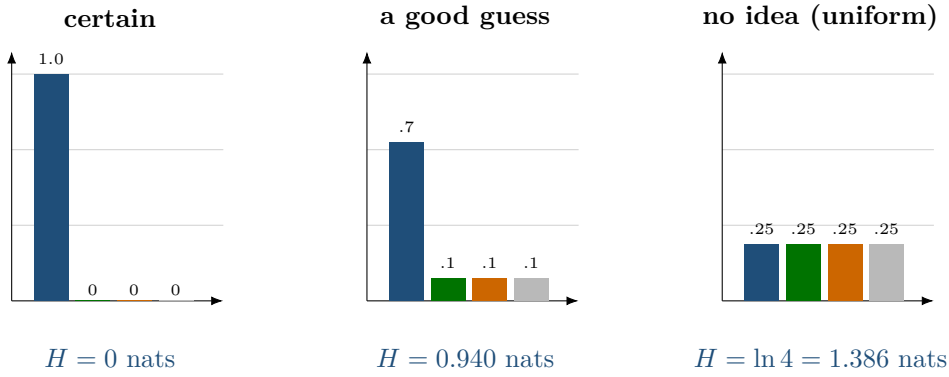


Figure 2.12. Three distributions over four outcomes and their entropies, all measuring the same thing — expected surprise — in three different arrangements of belief. *Certain* commits everything to one outcome and costs nothing to code, on average, once you already know the answer. *A good guess* spreads a little probability onto the alternatives and costs a little. *No idea* spreads evenly and pays the maximum for four outcomes, $\ln 4$. If the world actually follows the middle distribution but you predict with the right-hand (uniform) one, cross-entropy is $H(p,q) = \ln 4 = 1.386$ nats — your own entropy (0.940) plus a Kullback–Leibler surcharge of 0.446 nats for guessing the wrong shape, exactly as Eq. (2.5) predicts.

How to read it: The three panels are the same information-theoretic quantity under three beliefs, not three different quantities — watch the bars flatten left to right and the entropy climb in lock step. KL is not shown as a fourth panel because it is not a property of one distribution: it is the gap between the cost you paid (a cross-entropy) and the cost you had to pay (an entropy), which only appears when you compare two of these panels against each other.

2.6.2 Jensen's inequality

Key Concept — Jensen's inequality

If g is a convex function and X a random variable, then $g(\mathbb{E}[X]) \leq \mathbb{E}[g(X)]$; for strictly convex g, equality requires X almost surely constant. Applied to the convex function $-\log$:

$$-\log \mathbb{E}[X] \;\leq\; \mathbb{E}[-\log X], \qquad \text{i.e.} \qquad \log \mathbb{E}[X] \;\geq\; \mathbb{E}[\log X]. \tag{2.6}$$

Swapping the order of a log and an expectation turns an average into a lower bound.

Worked Example — Jensen on a two-outcome distribution

Let $X = 1$ or $X = 3$ with probability $1/2$ each, and $g = -\log$ (convex). Then

$$-\log \mathbb{E}[X] = -\log 2 \approx -0.693, \qquad \mathbb{E}[-\log X] = \tfrac{1}{2}(-\log 1) + \tfrac{1}{2}(-\log 3) = \tfrac{1}{2}\log 3 \approx 0.549,$$

and indeed $-0.693 \leq 0.549$. Flipping signs, $\log \mathbb{E}[X] = 0.693 \geq \mathbb{E}[\log X] = 0.549$: the expectation of the log strictly *underestimates* the log of the expectation whenever X is random, by a gap of $0.693 - 0.549 \approx 0.144$ nats here. The VAE (Chapter 4) gets its tractable objective — the ELBO — from exactly this swap: $\log p(\boldsymbol{x}) = \log \mathbb{E}[\cdot] \geq \mathbb{E}[\log \cdot]$, and the slack between the two sides is precisely the KL divergence from the approximate posterior to the true posterior. Figure 2.13 draws this gap directly: the curve, the two outcomes on it, the chord between them, and the vertical distance that is the whole content of the inequality.

2.6.3 Mutual information

Key Concept — Mutual information

The *mutual information* between random variables X and Z is the reduction in uncertainty about one upon observing the other:

$$I(X;Z) \;=\; H(X) - H(X \mid Z) \;=\; D_{\mathrm{KL}}\big(p(x,z) \,\|\, p(x)\,p(z)\big) \;\geq\; 0, \tag{2.7}$$

where $p(x,z)$ is the joint distribution and $p(x)p(z)$ the product of marginals. $I(X;Z) = 0$ iff X and Z are independent; unlike correlation, it detects arbitrary statistical dependence, not just linear.

Mutual information: what a representation should keep. The mutual information $I(A;B)$ measures how much knowing A tells you about B. It gives the cleanest description of the goal of representation learning: keep what is useful, throw away the rest.

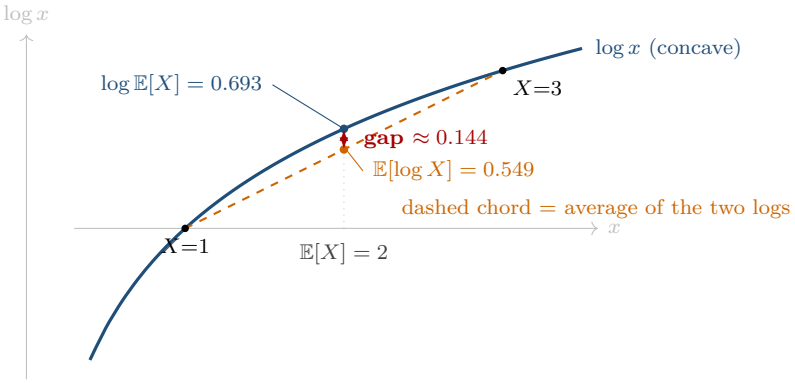


Figure 2.13. Jensen's inequality for $g = -\log$ (equivalently, concavity of log), on the two-outcome example above. The blue curve is $\log x$; the two black dots are the outcomes $X = 1$ and $X = 3$, each weighted $1/2$. The orange dashed chord connects them — averaging the *outputs* $\log 1$ and $\log 3$ gives the chord's height at $x = \mathbb{E}[X] = 2$, which is $\mathbb{E}[\log X] = 0.549$. But the curve itself, evaluated at the *averaged input* $\mathbb{E}[X] = 2$, sits higher: $\log \mathbb{E}[X] = 0.693$. Concavity — the curve bows above every chord it spans — forces this ordering for any non-degenerate distribution.

How to read it: The red double arrow is the entire inequality: chord below, curve above, at the same $x = \mathbb{E}[X] = 2$. Shrink the gap between $X = 1$ and $X = 3$ toward a single point and the chord rises to meet the curve — Jensen is equality exactly when X stops being random, which is why the ELBO gap in Chapter 4 vanishes only when the approximate posterior is exact.

- The *information bottleneck* (Tishby et al., 2000) writes that goal as an objective, $\min I(\boldsymbol{x}; \boldsymbol{z}) - \beta I(\boldsymbol{z}; \text{task})$: compress the input, but keep what predicts the task. Chapter 4 uses this to explain autoencoders.
- InfoNCE (Chapter 9) can be read as a *lower bound* on the information shared by two augmented views: $I(\boldsymbol{z}_1; \boldsymbol{z}_2) \geq \log(K+1) - \mathcal{L}_{\text{NCE}}$ with K negatives (van den Oord et al., 2018b).

A caution Chapter 15 develops: that bound is loose, and pushing up a bound is not the same as pushing up the quantity itself (Poole et al., 2019).

Key takeaway — surprise, in three arrangements

Entropy $H(p)$ is how surprised you expect to be when the world runs on p and you know it. **Cross-entropy** $H(p, q)$ is how surprised you expect to be when the world runs on p but you have been betting on q. **KL divergence** is the difference between them — exactly the extra cost of betting wrong, zero only when $q = p$. That is why "minimise the KL to the target" is simply the formal way of saying "make these two distributions agree".

2.7 Classical Retrieval and Clustering Toolkit

This section collects Lecture 02's classical methods, and every one is used later:

- **k-nearest neighbours** — the standard no-training test of DINO's features;
- **k-means** — produces DeepCluster's pseudo-labels;
- **DBSCAN** — the model for "clusters without choosing k", and an exam favourite;
- **PCA** — usually applied before clustering;
- **t-SNE** — how embeddings get plotted.

One question runs through all of them: *given only distances between embeddings, what structure can we find — neighbours, groups, classes — and what does each method quietly assume?* Search finds *local* structure; clustering finds *global* structure. And the key idea of Chapter 3 is that once either is even slightly better than chance, it can be fed back to the network as training targets.

2.7.1 Nearest neighbors

In plain words

> Finding a nearest neighbor the honest way is brute force: to know which stored point is closest to your query, measure the distance to *every* stored point, and keep the smallest. That is correct and also wasteful — most of those measurements were never going to win. A KD-tree (Section 2.7.2) is a shortcut for exactly this waste: pre-sort the points once into a tree of nested boxes, so that at query time, whole boxes can be ruled out with one cheap comparison against a splitting line, instead of one expensive distance computation per point inside them. Figure 2.14 draws both searches on the same data, so you can see directly how many comparisons each one actually makes.

k-nearest neighbours. To classify a new point, find the k closest labelled training points and take a majority vote (Lecture 02, slide 3).

- With $k = 1$, the boundary between classes is jagged and copies any mislabelled points.
- Larger k smooths the boundary, but near the edge of a small class its points can be outvoted by a larger neighbour.

Is kNN supervised or unsupervised? This is a quiz question, and the answer is *both halves*:

- **finding** the k nearest points uses no labels — unsupervised;
- **voting** among them uses their labels — supervised.

It is also *non-parametric*: nothing is trained. "Training" means storing the data; prediction is the search. And unlike k-means, there are no centres anywhere — a common mix-up.

Why kNN is the book's favourite evaluation. Three reasons.

1. **Nothing is trained.** No weights are fitted to make the classes separable, so a good score can only come from the features themselves.
2. **It uses exactly the distance the features were built for** — cosine for normalised features, Euclidean otherwise — and nothing else.
3. **Almost nothing to tune**: just k and the distance. Linear probes have learning rates, weight decay and training length, and different papers set them differently; kNN numbers are easier to compare.

In SSL papers you will see it as a "kNN monitor": freeze the network, compute features for the labelled training set, and classify each test image by a vote among its k nearest training features. The labels are only consulted at the vote. So if this accuracy rises during training, it means the unsupervised objective — which never saw a label — is organising the space by class on its own. Chapter 11's DINO results rest on exactly this argument.

2.7.2 Fast search: KD-trees

The problem. Finding the nearest neighbour by brute force means measuring the distance to all N points — fine for a thousand points, hopeless for a billion.

Building a KD-tree. A KD-tree (Bentley, 1975) sorts the points into boxes in advance:

1. Split all the points into two halves at the median of one coordinate (say x).
2. Split each half at the median of the next coordinate (y).
3. Keep splitting, cycling through coordinates, until each box holds a handful of points.

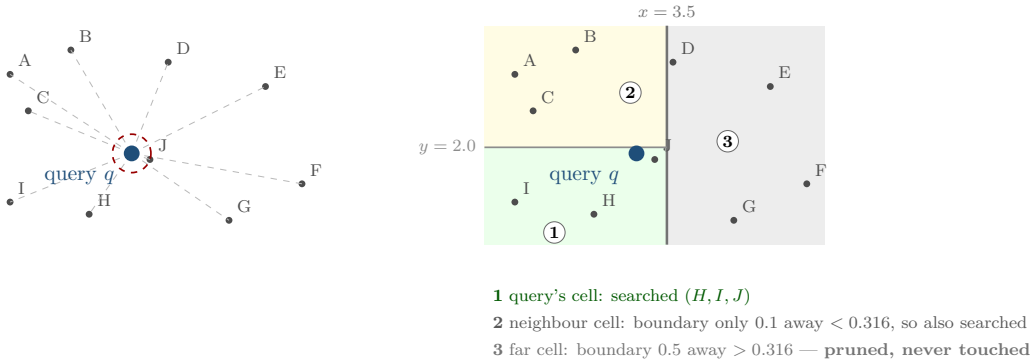
Building it costs $O(N \log N)$, once.

Searching it. To find the nearest neighbour of a query q:

1. Walk down the tree to the box q falls into, and note the best point there. Call its distance B.
2. Walk back up. At each split, ask: *could the other side contain something closer than B?* The other side is at least $|q_d - v|$ away, where the split was at value v on coordinate d.
3. If $|q_d - v| \geq B$, the whole other side is too far — skip it entirely. Otherwise, search it too.

In low dimensions most of the tree gets skipped, and a query costs about $O(\log N)$ instead of $O(N)$. (The slide's example inserts the point $(15, 80)$ by comparing x at the root, then y, until it reaches an empty spot below $(5, 25)$.)

Figure 2.14 puts brute force and a KD-tree side by side on the same ten points, so the difference in *how much work is done* is visible rather than asserted.



(a) brute force: 10 of 10 points measured **(b) KD-tree: 6 of 10 points measured**

Figure 2.14. The same query, the same ten points, two search strategies. **(a)** Brute force draws a dashed line to every point (all ten measured) before the dashed circle — radius 0.316, the true nearest neighbor J — can be trusted. **(b)** A KD-tree first splits on $x = 3.5$, then splits the left half on $y = 2.0$. The query lands in the bottom-left cell $\{H, I, J\}$, which is searched in full; because the query is only 0.1 from the y=2.0 boundary — closer than the best distance found so far — the neighbouring top-left cell $\{A, B, C\}$ must be checked too. The right cell $\{D, E, F, G\}$ is never opened at all: its boundary is 0.5 away, farther than the best distance already in hand, so nothing inside it could possibly win.

How to read it: Count the dashed lines in (a) against the shaded cells in (b): brute force pays for all ten measurements up front, while the KD-tree pays for two cheap plane comparisons (at the thick lines) and only then measures the six points those comparisons could not rule out. Both return the exact same answer, J at distance 0.316 — the tree does not approximate anything here, it only skips arithmetic that could not have changed the result.

Worked Example — A KD-tree query prunes half the space

Points $A = (2, 3)$, $B = (6, 1)$, $C = (4, 5)$; split the x-axis at $x = 4$ (points A, B left, C right). Query $q = (5, 5)$: descend right ($5 > 4$), reach C, best distance $\|q - C\| = 1$. On unwinding, check the split plane: $|q_x - 4| = 1$, which is *not smaller* than the best distance 1, so the left subtree cannot contain a strictly closer point and is discarded — no distance to A or B is ever computed. (Check: $\|q - A\| = \sqrt{13}$, $\|q - B\| = \sqrt{17}$, both indeed larger.)

Worked Example — An exact KD-tree query: pruning and backtracking

The lecture's worked trace (slide 9) asks for the nearest neighbor of $q = (52, 52)$ in a tree over the nine points $(51, 75), (25, 40), (10, 30), (1, 10), (50, 50), (35, 90), (70, 70), (55, 1), (60, 80)$, built so that the root splits on x at $(51, 75)$, the right child splits on y at $(70, 70)$, and so on. *Descent* (comparing at each level to the split coordinate): at $(51, 75)$, $52 \geq 51$, go right, $d(q, \text{root}) \approx 23.02$ — set `best` $\leftarrow 23.02$. At $(70, 70)$, $52 < 70$ on the y-split, go left, $d \approx 25.45$ (no update). At $(55, 1)$, $52 < 55$ on the x-split, go left to the empty child; $d \approx 51.08$ (no update). *Unwinding* now begins at $(55, 1)$: $|q_x - 55| = 3 < 23.02$, so the

sphere crosses this split; the unvisited right subtree of $(55, 1)$ must be checked (it is empty). Up to $(70, 70)$: $|q_y - 70| = 18 < 23.02$; check the other child $(60, 80)$, $d \approx 29.12$ (no update). Up to the root $(51, 75)$: $|q_x - 51| = 1 < 23.02$; the *whole left subtree* must be descended. Inside it $(25, 40)$, $(35, 90)$, $(10, 30)$ are visited in turn, and $(50, 50)$ is found at $\sqrt{4+4} \approx 2.83$; `best` shrinks to 2.83. From now on every further backtracking check passes trivially — plane distances all exceed 2.83 — and the rest of the tree is pruned in bulk. The search still had to visit nearly all nine points before the first tight candidate appeared: the point of the example, immediately below, is that the early wide sphere made nearly every prune fail, just as it will in high dimension. Figure 2.15 draws the splits, the query, and both best-distance circles — the wide one right after the root, and the tight final one — so the backtracking checks can be read off as "does this circle still cross that line."

The catch: high dimensions. A KD-tree only saves time if it can skip most of the tree. It skips a branch when the query's current-best circle does not reach across that branch's splitting line. But in high dimensions (Section 2.4.3) the best distance B stays large — every point is roughly equally far — so the circle reaches across almost every splitting line. Almost nothing gets skipped, the search visits nearly the whole tree, and it ends up *slower* than simply checking every point, after all the effort of building the tree.

The exercise of operating a small schematic tree by hand — deciding, given a query, a split value, and a current best distance, whether a subtree may be pruned — is the canonical exam calculation for this material, and the worked example already walked the longest case. Below it we look at why that machinery collapses at scale.

Common Confusion

KD-trees in high dimension. Three traps about KD-trees round out the picture. They do *not* require labels: they are purely spatial indexes over coordinates. They are *not* the answer to high-dimensional search: beyond roughly $d \approx 20$ they decay to linear or worse scans. And they can be *slower than brute force*: the expected $O(\log N)$ query assumes most subtrees prune, and in high d the memory and constant factors of the traversal dominate. The defining statement, "KD-trees' efficiency degrades as dimensionality increases," is the single most-examined fact in this area.

What practice shows. The lecture's slide from `ann-benchmarks.com` (Lecture 02, slide 12) plots accuracy (recall) against speed (queries per second). The exact tree methods sit at the bottom: to find the true nearest neighbour every time, they manage only tens of queries per second. Modern approximate methods — HNSW, the FAISS family, ScaNN, Annoy — find it more than 95% of the time while answering *a thousand to ten thousand times* more queries per second.

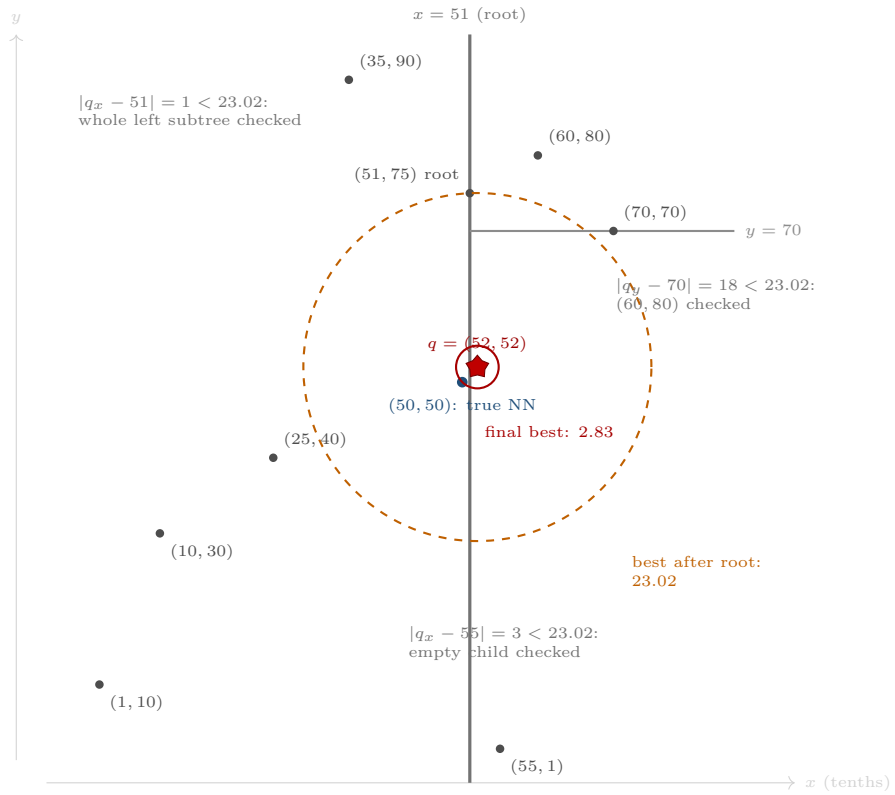


Figure 2.15. The nine-point KD-tree trace of the worked example, drawn to scale (axis units are tens: 5.2 means $x = 52$). The two solid/thick lines are the only splits the trace specifies: the root divides on $x = 51$, and its right child $(70, 70)$ divides the right half on $y = 70$. The dashed orange circle is the search radius right after the root is visited, 23.02; the solid red circle is the final radius, 2.83, once the true nearest neighbor $(50, 50)$ is found deep in the left subtree. Every annotated check compares the gap to a split line against the *current* best distance — and because the wide orange circle still reaches across every split line in the tree, every one of those checks forces a backtrack.

How to read it: Look at the orange circle first: it is centred on q with radius 23.02*, and it visibly crosses* both *solid lines, even the one* 18 *units away. That is why backtracking cannot be skipped here — the only cell it does* not *reach is the empty child of* $(55, 1)$*, and even that one has to be checked before the search can be sure it is empty. Only once the tiny red circle exists (after* $(50, 50)$ *is found) does anything actually get pruned, and by then almost every point has already been visited.*

The lesson the whole field has accepted: *in high dimensions, being exact costs too much to be worth it.* Settle for "almost always right" and gain orders of magnitude in speed — which is what the hashing and graph methods next do.

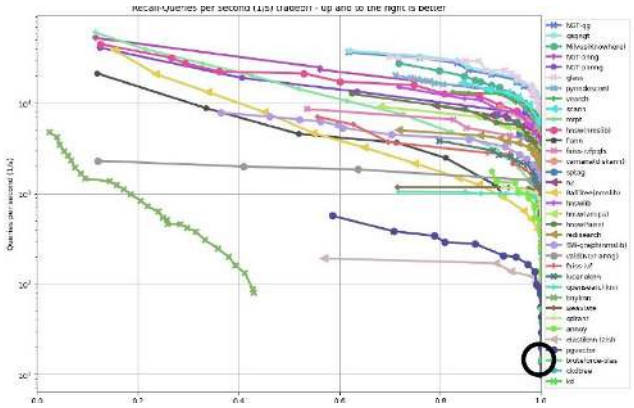


Figure 2.16. The empirical version of the last two paragraphs: recall (how often the true nearest neighbor is actually returned) against queries per second, log scale, for dozens of real nearest-neighbor libraries on the same benchmark. The circled cluster at bottom right — `bruteforce`, `kd`, `ckdtree` — are the *exact* methods: to push recall to 1.0 their throughput collapses to tens of queries per second. Every curve above and to the left of them is an approximate method trading a sliver of recall for three to four orders of magnitude more speed.

How to read it: Find the circled point, then look straight up: entire families of modern indexes sit two to four decades higher on the same recall column. That vertical gap is the whole argument for approximate search in one picture — exactness is not free, and in high dimension it is barely affordable at all.

Source: Lecture 02, slide 12.

2.7.3 Fast search: locality-sensitive hashing

In plain words

An LSH bit is a coin flip that is biased toward agreeing for nearby points. Draw a random line through the origin; a point gets a "+" if it falls on one side, a "−" if it falls on the other. Two points that are close together are usually on the same side of any given random line, so their bits usually match; two points that point in very different directions are roughly a coin flip either way. Stack a few such lines together and read off the sequence of bits as a *bucket label* — now only points sharing your query's exact bucket ever get an exact distance computed, and everyone else is skipped for free. The price of the speed is honesty about what got skipped: occasionally two points that really are close sit on opposite sides of one of the lines, land in different buckets, and the search misses each other. Nearby points *usually* collide. Not always.

LSH (Indyk and Motwani, 1998; Gionis et al., 1999) trades exactness for speed.

The promise. A hash is *locality-sensitive* if close points usually get the same hash and far points usually do not (Lecture 02, slide 10):

$$d(\boldsymbol{a},\boldsymbol{b}) \le r \;\Rightarrow\; p[h(\boldsymbol{a}) = h(\boldsymbol{b})] \ge p_1, \qquad d(\boldsymbol{a},\boldsymbol{b}) \ge c\,r \;\Rightarrow\; p[h(\boldsymbol{a}) = h(\boldsymbol{b})] \le p_2,$$

with $p_1 > p_2$.

The hash for cosine distance. Draw a random direction $\boldsymbol{r}$ and record which side of it each vector lies on: $h(\boldsymbol{u}) = \text{sign}(\boldsymbol{r}^\top \boldsymbol{u})$. Two vectors at angle θ land on the same side with probability $1 - \theta/\pi$ — always for identical vectors, never for opposite ones.

Sharpening it. One bit is a weak test. Join k bits into a code; two vectors now share the whole code with probability $(1 - \theta/\pi)^k$, which is still high for small angles but drops fast for large ones.

Searching (slide 11). Build L tables, each with its own random directions. Hash the query into every table, gather everything that shares its code in any table, and measure exact distances only for that small set. Two dials: more bits per code make each bucket smaller (faster, but more misses); more tables win the misses back.

Why it matters here. Modern search libraries (HNSW graphs, Faiss) use more sophisticated versions of the same bargain: accept approximate answers in exchange for enormous speed. The DINOv3 data engine (Chapter 11) compares millions of images by cosine similarity — exactly this setting.

Worked Example — Band-and-table arithmetic

Take random-hyperplane hashing, where a single bit collides with probability $1 - \theta/\pi$ for angular separation θ. Concatenating k independent bits into one signature makes two points collide only if *all* k bits agree, so the per-table collision probability becomes $p(\theta)^k = (1 - \theta/\pi)^k$: for $\theta = 60°$ ($p = 2/3$) and $k = 10$ bits this is about 0.017 — a single table misses such pairs constantly. That per-table failure is precisely what the L independent tables fix: a near neighbor is lost only if it collides in *no* table, an event of probability $\left(1 - p(\theta)^k\right)^L$, which for the same numbers and $L = 100$ tables drops to roughly 0.18, and to about 0.02 at $L = 230$. The (k, L) knobs thus play symmetric, separable roles: k discriminates (bigger k kills far-pair collisions, shrinking buckets and false positives), while L recalls (bigger L rescues the near pairs that larger k just threw away) — at the price of storing L copies of the index. Figure 2.17 plots both halves of this arithmetic.

The subtlety behind the slide's formal definition is worth isolating, because it separates LSH from every hash function one meets elsewhere. A cryptographic hash is engineered to

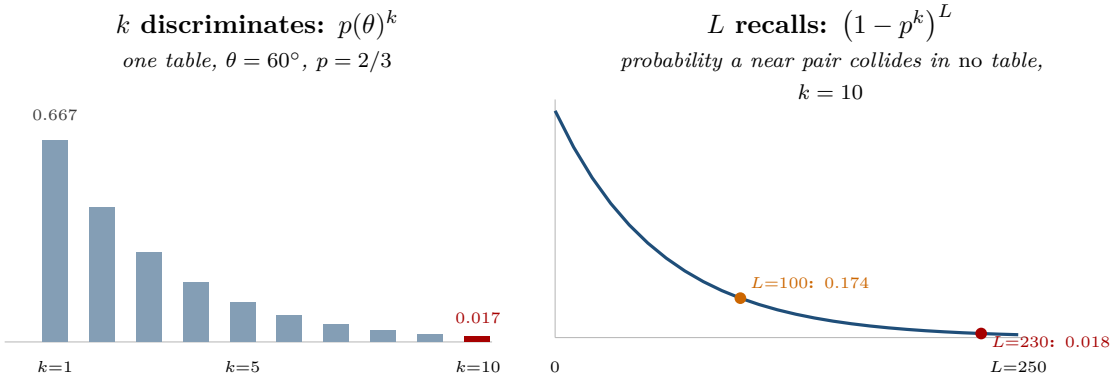


Figure 2.17. The band-and-table arithmetic, both knobs. **Left:** the per-table collision probability $p(\theta)^k$ for a pair 60° apart, as k grows from 1 to 10 — concatenating bits is what makes a table *selective*, at the cost of nearly killing true collisions too. **Right:** the probability that L independent tables *all* miss the same near pair, $(1-p^k)^L$, computed exactly and plotted for $k = 10$; the two marked points are the numbers in the worked example.

*How to read it: Read the two panels as cause and effect: the left panel's tiny bar (*0.017*) is a single table's collision chance after ten discriminating bits — good for precision, bad for recall. The right panel shows the price of getting that recall back: it takes on the order of a* hundred *tables just to get the miss rate near* 0.17*, and hundreds more to push it much lower. Selectivity and recall pull in opposite directions, and* (k, L) *is the two-dial control for trading one against the other.*

destroy locality: flip one input bit and the output changes beyond recognition. A locality-sensitive hash is engineered to *preserve* locality, probabilistically and on average — with no guarantee for any individual pair, only the guarantee that closeness makes collision more likely. That is why LSH is an *approximate* nearest-neighbor method: it promises to return a point within radius $c \cdot r$, with probability controlled by (k, L), and never promises the exact winner.

2.7.4 Clustering without labels

What clustering is. Clustering groups points so that points in the same group are more alike than points in different groups — with no labels. It is "classification without being told the classes".

No method wins everywhere. Methods differ in what they assume: do you know how many clusters there are? how much memory and time do you have? are there outliers? (Lecture 02, slides 13–14). The two families this book needs — *centroid-based* (k-means) and *density-based* (DBSCAN, OPTICS) — fail on exactly opposite kinds of data.

Lecture 02's slide 13 shows this with a grid of toy datasets. Two of them tell the whole story:

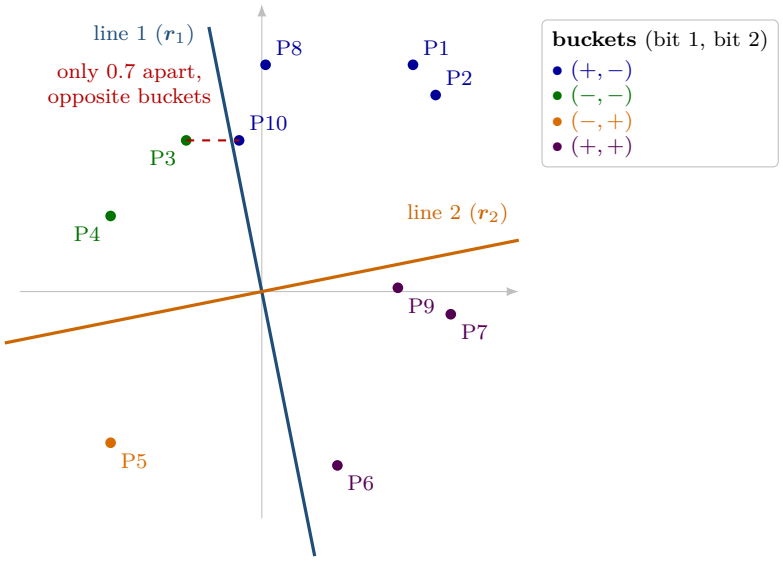


Figure 2.18. Two random lines through the origin cut the plane into four buckets; ten points are colored by their two-bit code (which side of line 1, which side of line 2). Points that are angularly close mostly share a color — the three blue points in the upper right, or the two green points on the left. The dashed pair, P_3 and P_{10}, is the honest exception: they are only 0.7 apart in raw distance, closer to each other than many same-color pairs are, but line 1 happens to pass between them, so they hash to different buckets and a single-table query for one would miss the other entirely.

How to read it: Trace line 1 with your finger: everything strictly to its right is "+" on bit one, everything to its left is "−," regardless of how far a point is from the line. P_3 and P_{10} are both close to the line itself, which is exactly why a small nudge — or, equivalently, the line landing a little differently for a different random draw — moves one of them across it. This is the whole tradeoff: a second, independently drawn line (or table) is what catches the pairs this particular line happened to split.

- **Two interleaved crescents.** k-means can only draw straight boundaries between centres, so with $K = 2$ it cuts both crescents in half. DBSCAN follows the crowd along each crescent and returns both intact.
- **Uniform noise, no structure at all.** k-means happily reports as many "clusters" as you asked for. DBSCAN does not invent borders: depending on ε it reports the whole square as a single cluster (as in the figure below) or as noise — either way, no fake structure.

Keep that grid in mind for everything below.

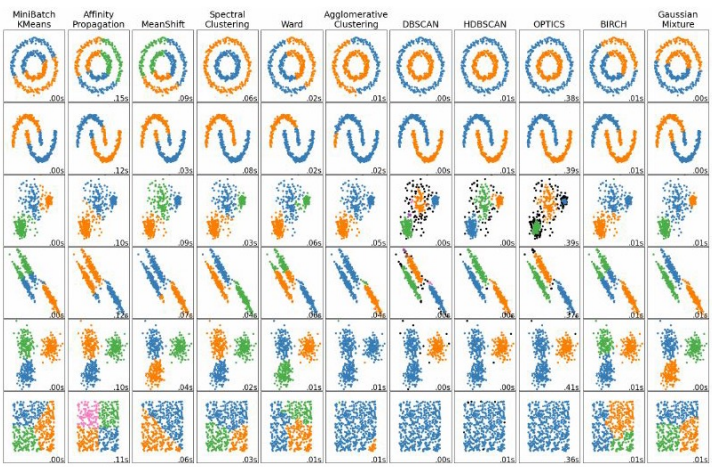


Figure 2.19. Eleven clustering algorithms against six synthetic datasets, drawn from an actual run rather than redrawn by hand — this grid is real algorithm output, timed and all, which is exactly why it belongs here instead of a hand sketch. Read column by column for one algorithm's behavior across datasets, or row by row for one dataset's behavior across algorithms. **MiniBatch KMeans** is column 1; **DBSCAN** is column 7. The two rows this chapter's prose singles out are the rings and the two-moons (rows 1–2): k-means can only ever cut the plane with straight Voronoi boundaries, so it has no way to trace a ring or a crescent, while density-based DBSCAN follows the curve of the data itself. The bottom row is pure uniform noise — no cluster structure at all by construction — and is the honesty test: an algorithm that reports confident-looking groups there is reporting an artifact of its own assumptions, not a property of the data.

How to read it: Compare column 1 (MiniBatch KMeans) against column 7 (DBSCAN) down the rings and two-moons rows first — that contrast is the one the surrounding paragraph describes in words. Then check both columns against the bottom, uniform-noise row: it is the one dataset in this grid built to contain nothing, so it is the fairest test of which algorithms know when to say so.

Source: Lecture 02, slide 13.

2.7.5 k-means

In plain words

k-means alternates two simple moves until nothing changes. **Assign**: give every point to whichever centroid is currently nearest — a plain distance comparison, nothing more. **Update**: move every centroid to the average position of the points that just chose it. That is the whole algorithm. Repeat those two moves and the centroids walk toward the middle of their own clusters, even starting from a bad guess — Figure 2.20 runs this to convergence on eight points, with every assignment checked by computer rather than by eye.

k-means partitions N points $\{\boldsymbol{x}_i\} \subset \mathbb{R}^d$ into K clusters $C_1, \ldots, C_K$ by minimizing the within-cluster sum of squares (WCSS),

$$\min_{C_1,\ldots,C_K} \sum_{j=1}^{K} \sum_{\boldsymbol{x}\in C_j} \|\boldsymbol{x} - \boldsymbol{\mu}_j\|^2, \qquad \boldsymbol{\mu}_j = \frac{1}{|C_j|} \sum_{\boldsymbol{x}\in C_j} \boldsymbol{x}, \tag{2.8}$$

where $\boldsymbol{\mu}_j$ is the centre of cluster j.

What k-means assumes. This objective is secretly a probability model: it is what you get from a mixture of round Gaussian blobs, all the same size, with each point assigned wholly to one blob. So k-means' preferences are not accidents of the algorithm — it *assumes* clusters are round and similar in size. It fails exactly where that is false: elongated blobs, crescents, clusters of very different spread.

How it is solved: Lloyd's algorithm (Lloyd, 1982). Finding the true best clustering is computationally hopeless, so everyone uses this simple loop:

1. Place k starting centres.
2. **Assign:** give each point to its nearest centre.
3. **Update:** move each centre to the average of its points.
4. Repeat 2–3 until nothing changes.

Each step can only lower the total squared distance, and it cannot go below zero, so the loop always stops. But it stops at a *local* best, which depends on where you started.

Key Concept — k-means as alternating minimization

The loop is best read as coordinate descent on the joint objective

$$J(\{r_{ij}\}, \{\boldsymbol{\mu}_j\}) \;=\; \sum_{i=1}^{N}\sum_{j=1}^{K} r_{ij}\,\|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2, \qquad r_{ij} \in \{0,1\}, \quad \textstyle\sum_j r_{ij} = 1,$$

where r_{ij} indicates the assignment of $\boldsymbol{x}_i$ to cluster j. The algorithm alternates two exact minimizations. *Assignments, centroids fixed*: J is minimized pointwise by $r_{ij} = \mathbb{1}[\,j = \arg\min_{j'} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{j'}\|^2\,]$ — assign each point to its nearest centroid. *Centroids, assignments fixed*: setting $\partial J/\partial \boldsymbol{\mu}_k = -2\sum_i r_{ik}(\boldsymbol{x}_i - \boldsymbol{\mu}_k) = 0$ gives $\boldsymbol{\mu}_k = (\sum_i r_{ik}\boldsymbol{x}_i)/(\sum_i r_{ik})$, the arithmetic mean of the currently-assigned points. Both half-steps can only decrease J, and $J \geq 0$, so Lloyd's iteration converges — though only to a local minimum, and which one is decided by the seeds. This coordinate-descent skeleton is why a later chapter can say "DeepCluster alternates k-means with backpropagation" and be talking about exactly the same kind of loop, with the second half-step replaced by SGD on the network weights (Section 2.7.8; Chapter 3).

Worked Example — Two Lloyd steps by hand

Points on the line: $1, 2, 8, 9$, with $K = 2$ and initial centroids $\boldsymbol{\mu}_1 = 1$, $\boldsymbol{\mu}_2 = 9$. *Round 1.* Assign by nearest centroid: $1, 2 \to C_1$; $8, 9 \to C_2$. Update: $\boldsymbol{\mu}_1 = 1.5$, $\boldsymbol{\mu}_2 = 8.5$. *Round 2.* The midpoint between centroids is 5; distances of 2 are 0.5 vs. 6.5, of 8 are 6.5 vs. 0.5 — no assignment changes, so the algorithm has converged with WCSS $= (0.5^2 + 0.5^2) \times 2 = 1$. Had we instead initialized $\boldsymbol{\mu}_1 = 2, \boldsymbol{\mu}_2 = 8$, round 1 would give $1, 2 \to C_1$, $8, 9 \to C_2$ anyway. Now start badly: $\boldsymbol{\mu}_1 = 0$, $\boldsymbol{\mu}_2 = 20$. Every point is nearer 0 than 20, so all four join C_1 and C_2 is *empty*. The update moves $\boldsymbol{\mu}_1$ to 5; $\boldsymbol{\mu}_2$ has no points, so it stays at 20 — and on the next round every point is still nearer 5. Lloyd's algorithm is stuck for good with one cluster and WCSS $= 4^2 + 3^2 + 3^2 + 4^2 = 50$, against 1 for the right answer. Initialization is the whole game (and empty clusters are a real failure mode, which is why practical code re-seeds them).

Figure 2.20 runs the same loop in two dimensions, on eight points and a deliberately bad initial guess, so that both the assignment step and the centroid movement are visible.

Historical Note

"k-means" is a name with three births. Steinhaus posed the partition problem in the continuous setting in 1957 (Steinhaus, 1956); MacQueen named and analyzed "k-means" in 1967 (MacQueen, 1967); and Lloyd had circulated the alternating assignment/update algorithm as a 1957 Bell Labs technical report on pulse-code modulation, published only in 1982 (Lloyd, 1982). Textbooks divide credit differently; the algorithm itself is indifferent.

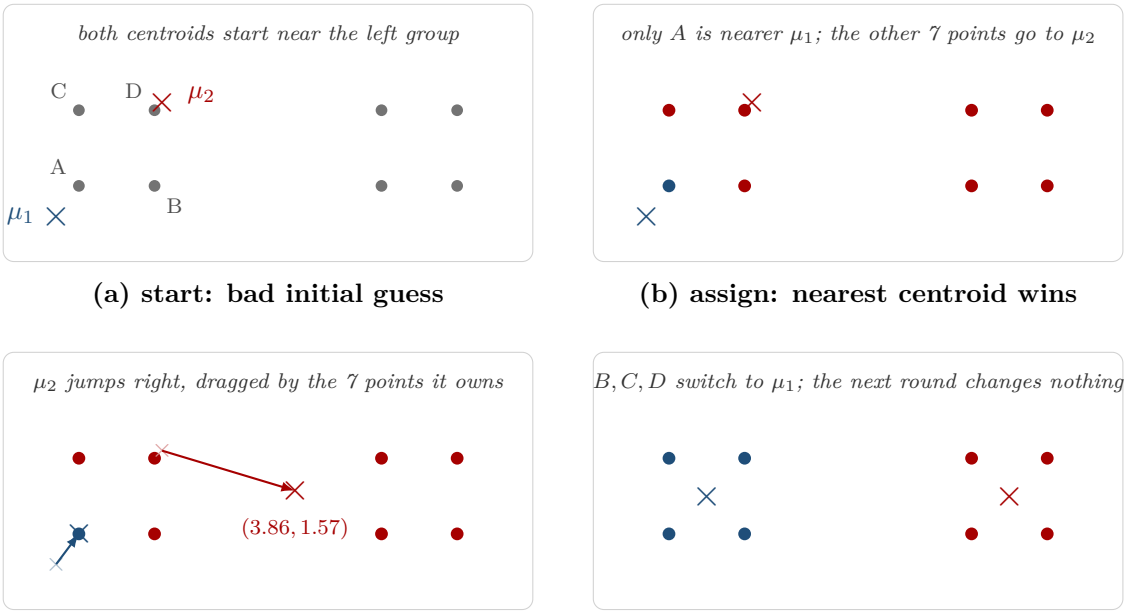
Plain k-means has three practical weak spots, each with a standard fix.

1. Where to start: k-means++. Random starting centres can land two in one cluster and none in another, and Lloyd's algorithm then gets stuck. k-means++ (Arthur and Vassilvitskii, 2007) places the starting centres one at a time, and prefers points that are *far* from every centre chosen so far:

1. Pick the first centre uniformly at random.
2. For every point $\boldsymbol{x}$, measure $D(\boldsymbol{x})$, its distance to the nearest centre chosen so far.
3. Pick the next centre at random with probability proportional to $D(\boldsymbol{x})^2$ — far points are much more likely.
4. Repeat until there are K centres.

Points in an already-covered cluster have small D, so they are rarely picked again; isolated groups have large D, so they get a centre early. This alone guarantees a result within a factor $O(\log K)$ of the best possible, on average, and removes most bad runs (Lecture 02, slide 16).

2. Too much data: mini-batch k-means. Each step of Lloyd's algorithm looks at every point. Mini-batch k-means (Sculley, 2010) looks at only a small random batch, and nudges



(a) start: bad initial guess **(b) assign: nearest centroid wins**

(c) update: centroids → cluster means **(d) repeat → converged**

Figure 2.20. One full run of Lloyd's algorithm on eight points, starting from a deliberately poor guess (every assignment below verified by computer, not by eye). **(a)** Two initial centroids, both dropped near the left group. **(b)** Assignment by nearest centroid puts only point A with μ_1; every other point, including the entire right-hand group, is nearer to μ_2 at this stage. **(c)** The update step recomputes each centroid as the mean of its assigned points: μ_1 barely moves (it owned one point, itself), while μ_2 is dragged far to the right by the seven points now pulling on it. **(d)** Reassigning with the new centroids flips B, C, D onto μ_1 — the left group is now assembled correctly — and one more update step places both centroids at the true cluster means, $(1.5, 1.5)$ and $(5.5, 1.5)$; a further round changes no assignment, so the algorithm has converged.

How to read it: Panel (b) is the surprising one: a bad start does not mean a bad outcome, it means extra rounds. Watch μ_2 in particular — it starts inside the left group in (a), gets dragged rightward by the majority vote in (c), and only settles into the right group's true center after the reassignment in (d). Nothing here was hand-placed to look convincing; every distance comparison was checked numerically.

each centre that batch touches towards its points: $\boldsymbol{\mu} \leftarrow (1-\eta)\,\boldsymbol{\mu} + \eta\,\boldsymbol{x}$ with $\eta = 1/v[\boldsymbol{\mu}]$, where $v[\boldsymbol{\mu}]$ counts how many points that centre has ever absorbed. A new centre moves a lot; a centre backed by thousands of points barely moves. It is a running average, and it is what makes k-means usable at web scale (Lecture 02, slide 17).

3. How many clusters: choosing K. The total squared distance (WCSS) *always* goes down as K goes up — with one centre per point it reaches zero — so it cannot tell you the right K. People use the "elbow" of that curve, information criteria, or simply whatever works best downstream. Chapter 3's DeepCluster just picks a large K and lives with it.

Common Confusion

"The clusters look wrong, so k-means must have failed." Two different things can go wrong, and they have different fixes.

K too large. Ask for 20 clusters on data with 3 natural groups and k-means will not complain: it cuts each real group into several pieces, every centre finds points, and the output looks tidy while the grouping is wrong.

Bad features. If each cluster contains a mix of object categories, the fault is usually not k-means or K at all. The feature space itself has not separated the categories, and no way of cutting up a jumbled space can unjumble it. Fix the representation, not the clustering. (This is what Quiz Q6 of the lecture is after.)

Failure Modes

k-means. (i) *K unknown*: the objective rewards more clusters, so an over-large K silently splits natural groups and an under-large K merges them; there is no principled fix inside the method. (ii) *Seed sensitivity*: different initializations reach different local optima (Worked Example 2.7.5); always use k-means++ and restarts. (iii) *Spherical-Gaussian assumption*: because the assignment uses Euclidean distance to a centroid, k-means implicitly models clusters as equal-variance balls — it dissects elongated, anisotropic, or non-convex clusters (the classic two-moons failure) regardless of the true structure. (iv) *Outliers*: every point must be assigned, so a handful of outliers can drag centroids off the data mass. (v) *Empty clusters*: in high dimensions a centroid can lose all points; this becomes a real collapse mode when k-means is put inside a learning loop (DeepCluster, Chapter 3). Non-spherical data is precisely where DBSCAN enters.

2.7.6 DBSCAN and OPTICS: clustering by density

In plain words

DBSCAN never asks how many clusters you want; it asks how crowded a neighbourhood has to be to count as "inside" something. Stand at any point and look within a small radius ε. If enough other points are close enough — at least `minPts` of them — you are standing in a crowd: a *core* point. If you are not yourself in a crowd but you are right next to someone who is, you get to join their cluster as a *border* point. Otherwise you are standing alone, and DBSCAN calls it exactly that: *noise*, not a cluster of size one. Clusters grow by chaining crowded points together, so their shape can bend and curve however the data actually does — nothing forces it into a round blob the way a centroid does, and nothing forces every point to belong somewhere the way k-means does.

The idea. DBSCAN (Ester et al., 1996) defines a cluster as a *crowded region*, separated from other crowded regions by empty space. There are no centres and no K.

The lecture's picture (Lecture 02, slide 18) is a music festival:

- a **core point** is someone with at least `minPts` people within arm's reach ε — they are *in* a crowd;
- a **border point** is someone within arm's reach of a core point, but not surrounded themselves — on the *edge* of a crowd;
- **noise** is someone standing alone in an empty field.

The rule, precisely. Let $N_\varepsilon(p) = \{q : d(p,q) \le \varepsilon\}$ be the points within ε of p. Then p is a core point if $|N_\varepsilon(p)| \ge$ `minPts`. (This book counts p itself; the lecture's quiz does not, and Exercise 5 shows the difference matters.)

How clusters form. Start at any core point and spread to every core point within ε, then to theirs, and so on — like a flood fill. Border points join the first cluster that reaches them. Whatever is never reached is noise.

What you get. Because clusters grow along chains of crowded neighbourhoods rather than around a centre, they can be any shape — crescents, rings, long thin strands. DBSCAN finds how many there are by itself, and it labels outliers as noise instead of dragging a cluster centre towards them.

Worked Example — DBSCAN by hand on six points

Take the 2D dataset $A(0,0)$, $B(1,0)$, $C(0,1)$, $D(5,5)$, $E(6,5)$, $F(10,10)$, with $\varepsilon = 1.5$ and `minPts` $= 3$. Pairwise gaps: in the first group, $d(A,B) = 1$, $d(A,C) = 1$, $d(B,C) = \sqrt{2} \approx 1.41$, all $\le \varepsilon$ and pairwise — so A, B, C each see 3 points within ε and are *core*, forming one cluster $\{A, B, C\}$. In the second group, $d(D,E) = 1 \le \varepsilon$, but neither D nor E has any third point within ε: both fail the core-point test and are marked *noise*. F is far from everything: noise. Result: one cluster, three noise points. No mention of K anywhere — and two points that k-means would have merged into a spurious "cluster" are honestly reported as not belonging anywhere.

Where DBSCAN wins and where it fails.

- **Wins: curved clusters.** On the two crescents, each crescent is one continuous crowd, so DBSCAN follows it round the bend and labels it as a single cluster — exactly what k-means cannot do.
- **Fails: clusters of different density.** DBSCAN uses one radius ε everywhere. Make it small enough to separate the dense clusters, and the sparse clusters dissolve into noise. Make it big enough to keep the sparse clusters, and neighbouring dense clusters merge. (Quiz questions Q10/Q11 test this, and it is what OPTICS, below, fixes.)

- **Honest: pure noise.** On a featureless rectangle of random points, k-means still draws borders; DBSCAN draws none — it reports one undivided cluster or, with a small ε, only noise.

Failure Modes

DBSCAN. (i) *Varying density*: a single global ε cannot describe a dataset whose clusters have different densities — tightening ε to separate a sparse pair of clusters shreds the dense one into noise; this is the method's characteristic failure. (ii) *Parameter sensitivity*: the clustering changes qualitatively as ε crosses the dataset's natural neighbor distances; in practice one reads the sorted k-distance plot (distance to the k-th neighbor for each point) and places ε at its elbow. (iii) *Memory*: the neighborhood computation is $O(N^2)$ in the worst case unless index structures apply (Lecture 02, slide 19). (iv) *High dimensions*: as with KD-trees, distance contrasts wash out, so ε becomes hard to set on raw deep features — reduce dimension first.

OPTICS (Ankerst et al., 1999) addresses the varying-density failure by giving up the single threshold: instead of one clustering it computes an *ordering* of the points together with, for each point, its *reachability distance* to the already-processed set. Valleys in the reachability plot are clusters at *all* density scales simultaneously; choosing an ε slices the plot, so DBSCAN is recoverable as a special case. OPTICS is usually slower than one DBSCAN run but replaces parameter searching with inspection, and its memory profile suits large sets (Lecture 02, slide 21). When even OPTICS is too heavy, the standard recipe is to subsample or pre-cluster to exemplars and run the density analysis on those. HDBSCAN, a modern hierarchical successor, is not on the slides; we note only that the research line continues.

Where does this leave SSL? The lecture's summary table — k-means needs K and a meaningful mean but minibatches gracefully; DBSCAN needs only a neighborhood notion and an ε, tolerates outliers, but costs quadratic memory (Lecture 02, slides 19–20) — reads today as a menu for hybrid method design: pick the clustering whose failure modes you can tolerate inside a training loop, run it on PCA-reduced features, and feed the assignments back as targets. Chapter 3 develops exactly this, from DeepCluster to the data engines of Chapter 11, where hierarchical k-means is used to *balance* billion-scale training pools.

2.7.7 Dimensionality reduction: PCA and t-SNE

Deep features are high-dimensional (hundreds to thousands of coordinates), and everything in Section 2.7 degrades with dimension. Two reductions cover the course's needs, with sharply different purposes. They are not competitors: one is a pipeline tool and the other is a visualization, and confusing their jobs is a reliable source of misread figures.

PCA. Principal component analysis (Jolliffe, 2002) finds the linear subspace capturing maximal variance: center the data matrix $X \in \mathbb{R}^{N \times d}$ (subtract the column means), form the covariance $\Sigma = \frac{1}{N} X^\top X$, and take the eigenvectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m$ of its m largest eigenvalues — or,

equivalently, the top-m right singular vectors of X. Projecting $\boldsymbol{z} \mapsto X\boldsymbol{v}_j$-style onto this basis decorrelates the features and preserves, in Euclidean geometry, as much of the data's spread as any m-dimensional linear map can. PCA's role in this book is industrial, not glamorous: DeepCluster reduces CNN features with PCA *before* k-means (Caron et al., 2018), because clustering raw 4096-dimensional features is both slower and geometrically noisier (Lecture 02, slides 21 and 24). Two of its properties are doing the work, and both trace back to the earlier geometry sections. First, decorrelation: the projected coordinates are orthogonal, so the Euclidean distances that k-means consumes are not double-counting directions in which the features were redundant. Second, dimension control: distance concentration (Section 2.4.3) makes neighbor gaps in 4096 dimensions nearly meaningless, while a 256-dimensional whitened space keeps them marginally readable. It works because it is exactly the operation k-means implicitly assumes: it hunts Euclidean variance. Skip this step in a clustering hybrid and the pipeline degrades — distances wash out, centroids latch onto noise directions, and the pseudo-labels of the next section lose their structure. (Lecture 02's slide 21 recommends the same move for DBSCAN when memory or ε-setting becomes intractable.)

t-SNE. t-SNE (van der Maaten and Hinton, 2008) is for *display*, not for pipelines. From the $N \times N$ matrix of pairwise distances in the high-dimensional space it builds affinities $p_{ij} \propto \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/2\sigma_i^2)$ (symmetrized, with σ_i set per-point to fix an effective neighborhood size), initializes a 2D map with affinities q_{ij} under a heavy-tailed Student-t kernel, and moves the map points to minimize $D_{\mathrm{KL}}(P\|Q)$. The heavy tail is the trick: it lets moderate high-dimensional distances map to large visual separations, unfolding the embedding locally without demanding global metric fidelity. The interpretation rules follow directly from the construction — and are the most common misuse in SSL papers:

Common Confusion

Reading t-SNE plots. A t-SNE plot faithfully shows *who is whose neighbor* but not *how far apart clusters are* and not *how large clusters are*: inter-cluster distances and relative cluster sizes are artifacts of the KL objective and the perplexity knob, not properties of the embedding. Running t-SNE twice can place the same two clusters adjacently or across the figure; neither means anything. Use t-SNE to inspect local neighborhood structure (Chapter 13 uses exactly this reading for world-model features), never to claim "well-separated clusters" — that quantitative claim belongs to the clustering metrics of Chapter 3. PCA, by contrast, *does* preserve global linear geometry, but too weakly: in 2D it often hides nonlinear cluster structure that t-SNE exposes.

2.7.8 From clustering to pseudo-labels

Everything in this section so far treats the embedding space as fixed — we retrieve in it, we cluster it, we plot it. The lecture's pivotal move, and the hinge between this chapter and Chapter 3, is to *feed the cluster structure back into the network that produced the features*

(Lecture 02, slides 22–24). One round of k-means on frozen features produces, for every image, a cluster index $y_i \in \{1, \dots, K\}$; those indices are called *pseudo-labels*, and a classifier head can be trained on them by ordinary cross-entropy exactly as if a human had supplied them. Train, re-extract features, re-cluster, repeat.

Why does this bootstrapping work at all? The slide's boxed answer is this chapter's epigraph: *anything better than chance can be used as a learning signal.*

And a randomly initialised network is already slightly better than chance. Its local, shared filters naturally group similar textures and edges. The DeepCluster authors measured it: a small classifier (an MLP) trained on a *randomly weighted* AlexNet's last convolutional features gets about 12% on ImageNet, where guessing gets 0.1% (Caron et al., 2018). That weak signal is enough to start the loop:

$$\text{better features} \to \text{cleaner clusters} \to \text{cleaner pseudo-labels} \to \text{better features} \to \cdots$$

Two ways the cluster-and-train loop fails. Chapter 3 studies them in detail; they deserve names now, because they follow the book all the way to Chapter 11.

- **Collapse.** Nothing stops the network from mapping every image to the same vector. Then every image lands in one cluster, every pseudo-label is identical, and the task is trivially "solved". Working systems add explicit guards: cluster-balancing terms, resetting empty clusters, or negative examples.
- **Confirming its own mistakes.** The training step treats the current clusters as the truth. If two categories were wrongly merged, the network is trained to be *more* confident in that merge, and the next clustering inherits the error instead of fixing it.

The joint-embedding methods of Chapters 9–11 can be read as machinery for keeping this same bootstrapping loop alive while avoiding both failures.

2.8 Summary and Outlook

The toolkit is complete. Here it is, grouped by job:

- **Turn an input into a vector:** MLPs, CNNs, ViTs.
- **Turn a target into a gradient:** cross-entropy, MSE, KL — whether the target came from a person or was manufactured.
- **Compare two vectors:** cosine similarity on normalised features (where it agrees with Euclidean distance).
- **Update the weights:** SGD with momentum, or Adam.
- **Make stable targets out of the network's own past:** EMA copies and stop-gradients.

- **Describe what is being optimised:** entropy, KL, Jensen's inequality, mutual information.
- **Use the vectors:** kNN, KD-trees and LSH to search; k-means, DBSCAN and OPTICS to cluster; PCA and t-SNE to look.

Every method in Parts II and III is a new combination of these pieces. When a later chapter introduces something that looks new, ask first: *which piece was swapped?*

Check Your Understanding

Level 1 (recall).

1. Write the softmax of logits $\boldsymbol{s} = (1, 1, 1)$. What happens to the cross-entropy of *any* target class?
2. State Jensen's inequality for $-\log$ in both directions.
3. What two quantities does DBSCAN require instead of a cluster count K?

Level 2 (compute).

1. Compute $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v})$ and $\|\boldsymbol{u} - \boldsymbol{v}\|$ for $\boldsymbol{u} = (3, 4)$, $\boldsymbol{v} = (4, 3)$. Repeat after normalizing both to unit norm, and verify $d^2 = 2(1 - \mathrm{sim})$ only in the second case.
2. For the points of Worked Example 2.7.6, recompute the DBSCAN labeling with $\varepsilon = 1.5$ but `minPts` $= 3$ where the count is taken over *other* points only (the convention of Lecture 02). What changes relative to the worked example?
3. A teacher's EMA has $m = 0.99$. Starting from $\boldsymbol{\theta}_T = \boldsymbol{\theta}_0$, after how many student updates does the teacher's weight toward the initial value fall below 1% of its original contribution?

Level 3 (reason).

1. Show that minimizing cross-entropy $H(p, q)$ over q with p fixed is equivalent to minimizing $D_{\mathrm{KL}}(p\|q)$, and explain why this justifies calling classifier training "distribution matching."
2. Explain why a KD-tree query can prune a subtree using only the distance to the splitting plane, and why this argument weakens in high dimensions.
3. InfoNCE with logits sim/τ: describe in words the effect of $\tau \to 0$ on which negatives dominate the gradient.

Level 4 (connect).

1. DINO trains a student to match an EMA teacher through a temperature-scaled cross-entropy on normalized features. Identify, for each ingredient, which failure it prevents or which quantity it controls, using this chapter's concepts.
2. Worked Example 2.6.2 left a gap of about 0.14 nats. For the VAE, this gap has a meaning. What is it, and when does it vanish? (Chapter 4 checks your answer.)

3. k-means is sometimes called "EM for a mixture of equal spherical Gaussians with hard assignments." Using the spherical-assumption failure mode, construct a 2D dataset where k-means fails but a full Gaussian mixture would not.

Level 5 (extend).

1. The bound $I(\boldsymbol{z}_1; \boldsymbol{z}_2) \geq \log(K+1) - \mathcal{L}_{\text{NCE}}$ suggests increasing the negative count K without limit. Give two distinct reasons — one statistical, one practical — why this reasoning breaks in real training.
2. Prove from Eq. (2.5) that $D_{\text{KL}}(p\|q) \geq 0$, using Jensen's inequality. *Hint: apply Eq. (2.6) to $X = q(x)/p(x)$ under p.*
3. t-SNE minimizes $D_{\text{KL}}(P\|Q)$ rather than $D_{\text{KL}}(Q\|P)$. Using the asymmetry of the KL, argue why the chosen direction penalizes *splitting true neighbors apart* in the map more than it penalizes crowding non-neighbors together, and connect this to the reading rules in the Common Confusion box.

Solution sketches.

1. $\text{softmax}(\boldsymbol{s}) = (1/3, 1/3, 1/3)$; $\mathcal{L}_{\text{CE}} = \log 3 \approx 1.099$ regardless of the target — a uniform prediction pays the worst-case flat rate, the entropy of the uniform distribution over three classes.
2. $-\log \mathbb{E}[X] \leq \mathbb{E}[-\log X]$, equivalently $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$, since $-\log$ is convex.
3. A neighborhood radius ε and a density threshold `minPts`.
4. $\text{sim} = 24/25 = 0.96$; $\|\boldsymbol{u} - \boldsymbol{v}\| = \sqrt{2}$. After normalization $\boldsymbol{u}/\|\boldsymbol{u}\| = (0.6, 0.8)$, $\boldsymbol{v}/\|\boldsymbol{v}\| = (0.8, 0.6)$: cosine unchanged (0.96), $d^2 = 0.08 = 2(1 - 0.96)$ as the hypersphere identity requires; before normalization $\sqrt{2} \neq \sqrt{0.08}$.
5. Now A, B, C each see only 2 *other* points within ε, so no core points exist and all six points are noise — the entire clustering evaporates. (Excluding the point itself with `minPts`=3 is the same as `minPts`=4 counting it. Going the other way, `minPts`=2 counting the point itself would make D, E core too, giving clusters $\{A, B, C\}$ and $\{D, E\}$.) The threshold, not the geometry, decided whether the pair counted.
6. The initial weights enter $\boldsymbol{\theta}_T$ with coefficient m^t; solve $0.99^t < 0.01$: $t > \log 0.01 / \log 0.99 \approx 458$ updates.
7. $H(p, q) = H(p) + D_{\text{KL}}(p\|q)$ from Eq. (2.5); $H(p)$ is constant in q, so gradients coincide. Classifier training moves the model distribution toward the target distribution in KL.
8. Every point on the far side of a split plane is at least plane-distance away, so a best-so-far point closer than the plane excludes the whole subtree. In high dimension the query ball reaches most planes of most nodes (distances concentrate), so almost no prune survives — the tree degenerates toward a linear scan.
9. As $\tau \to 0$ the softmax concentrates on the largest logit — the *hardest* (most similar) negative — so the gradient is dominated by pushing the anchor away from its nearest negatives alone.

10. EMA teacher: prevents the moving-target chase from diverging and (empirically) the collapse to constant outputs; the teacher receives no gradient (stop-gradient), which makes the student–teacher relation one-way; *centering* the teacher's outputs stops one dimension from taking over; temperature on the teacher sharpens targets (entropy minimization), which stops the outputs going uniform; normalization fixes the logit scale so temperature is meaningful (Sections 2.4.2, 2.5.2, 2.5.4; full account in Chapter 11).
11. The gap is the KL from the approximate posterior $q(\boldsymbol{z} \mid \boldsymbol{x})$ to the true posterior $p(\boldsymbol{z} \mid \boldsymbol{x})$; it vanishes iff the approximate family contains the true posterior.
12. Two elongated parallel Gaussians whose principal axes are not aligned with the segment joining their means (a thin diagonal bar next to a round blob at equal nearest-centroid distance): k-means cuts both along the Voronoi bisector of the centroids, merging part of the bar into the blob; a full mixture fits covariances and recovers both.
13. Statistical: the bound needs negatives sampled to cover the data manifold — with finite K the estimator is biased and loose (Poole et al., 2019), and tightening at fixed mutual information saturates at $\log(K+1)$. Practical: false negatives — random "negatives" include same-class samples that the loss then actively separates (Chapter 9).
14. $D_{\mathrm{KL}}(p\|q) = \mathbb{E}_{x\sim p}[-\log(q(x)/p(x))] \geq -\log \mathbb{E}_p[q(x)/p(x)] = -\log \sum_x q(x) = 0$, applying Eq. (2.6) with the convex $-\log$.
15. The $p_{ij}\log(p_{ij}/q_{ij})$ form weights each term by the *data-space* affinity: large p_{ij} with small q_{ij} (true neighbors mapped far apart) costs heavily, while small p_{ij} with large q_{ij} (non-neighbors mapped together) costs little. Hence strong local-neighborhood fidelity and weak control of crowding, global distances, and cluster sizes — exactly the reading rules stated.

Concept Map — how this chapter connects

Backward. Chapter 1 motivated why representations matter and how they are evaluated; this chapter supplied all machinery those protocols (kNN, linear probes in Section 2.7.1's spirit, clustering) actually run on.

Forward. Chapter 3 puts the classical toolkit to work: DeepCluster alternates k-means (Section 2.7.5) on PCA-reduced features with cross-entropy training. Chapters 4–5 use MSE-as-Gaussian-likelihood, KL, Jensen, and the information bottleneck. Chapters 9–11 are built from cosine similarity on the hypersphere, temperature, InfoNCE as KL/MI, stop-gradient, and the EMA teacher. The LSH and retrieval view returns at scale in the DINOv3 data engine (Chapter 11), and t-SNE reappears as the diagnostic lens on world-model embeddings (Chapter 13). Part III (Chapters 15–16) then interrogates what, among these tools, is actually responsible for SSL working.

Part II

The Course: Lecture by Lecture

CHAPTER 3

Clustering and Classical SSL Hybrids

> The essence of many SSL methods: anything better than chance can be exploited as a learning signal.
>
> Lecture 02, slide 23

In this chapter

3.1 Nearest Neighbors: From Metric to Search ..... 86 • Why retrieval is the first SSL skill • KD-trees: exact search, geometrically • Locality-sensitive hashing: probabilistic search • The modern ANN landscape
3.2 Clustering as a Signal Generator .. 95 • The clustering zoo on one slide • k-means in action • Seeding matters: k-means++ • Scaling up: k-means in mini-batches • DBSCAN: density instead of centroids
3.3 DeepCluster: Closing the Loop . 108 • DeepCluster: cluster your own features, then train on the cluster numbers • A round trip through one epoch • Twins of EM: the alternation, written down
3.4 STEGO: Distilling Feature Correspondences into Segmentation 117 • The whole idea first, as a story (almost no maths) • STEGO: sharpen a correspondence that is already there • The recipe • From the story to the formulas: why each piece of maths is needed • The CRF detour — and why STEGO's loss is one • Both hybrids, one thesis
3.5 Chapter summary 134
3.6 Check Your Understanding 136

Chapter roadmap

Before you start. You need three things from Chapter 2: what a *distance* between two vectors is (Euclidean and cosine), what k-means does (put every point in the group whose average it is closest to), and what *cross-entropy* is (the loss you use when a network must pick one class out of K). You also need to know what a convolutional network is, at the level of "it slides small filters over an image." Everything else is built here.

The idea in one sentence. Any procedure that sorts your data better than random guessing is already a training signal — because if you train a network to reproduce that sorting, the network's features get better, and better features make the next sorting better still.

What we cover.

- **Finding neighbours** (Section 3.1). Before you can group points you must be able to ask "who is near me?" cheaply. KD-trees answer it exactly in low dimension; hashing answers it probabilistically at any dimension; the modern approximate systems are how anyone does this at web scale.
- **Grouping points** (Section 3.2). k-means and DBSCAN are the two workhorses. They fail in opposite situations, which is why you should know both.
- **Closing the loop, image level** (Section 3.3). DeepCluster clusters the network's own features, calls the cluster numbers "labels," and trains on them. Then it does it again.
- **Closing the loop, pixel level** (Section 3.4). STEGO takes a frozen backbone whose features already match up across images, and sharpens that matching into a segmentation — with no labels at any stage.

Afterwards you will be able to explain why a randomly initialised network is *not* a random feature extractor; describe the DeepCluster loop, its three ways of collapsing, and the patch for each; read the STEGO loss as a classical energy function; and say, for any new "train on your own output" method, where the non-circularity comes from.

Here is the whole chapter in one move. Suppose you have a pile of unlabelled images and some crude, non-learned routine — clustering, say — that sorts them into groups. The groups are bad. But they are not *random*: they are better than chance. That is enough. Call the group numbers "labels," train the network to predict them, and the network is forced to find whatever visual evidence actually distinguishes one group from another. Those improved features then feed a better clustering, which supplies a better label set, and the cycle turns. Nobody outside ever supplied a fact. The lecture boxes this thought on slide 23 and the rest of the chapter is two different ways of cashing it in.

Lecture 02 — "Clustering" — is the hinge of the course. Its first two acts rehearse the classical, non-learning machinery: nearest-neighbour search, then clustering. Its third act closes that machinery into a loop with backpropagation. The toolkit itself — distances, KD-trees, LSH, k-means and its refinements, DBSCAN, OPTICS, PCA — is developed with worked numbers in Chapter 2, and this chapter does not re-teach it.

Instead we do three things a slide deck cannot stop to do. First, we make the retrieval machinery *visible*: a KD-tree partition and its backtracking search, an LSH banding scheme, and a Lloyd iteration, drawn step by step. Second, we bring in the external evidence that anchors both ends of the chapter — the ann-benchmarks landscape of modern approximate-nearest-neighbour (ANN) systems, and the DeepCluster results table whose cells the lecture

Table 3.1. kNN classifier configuration variants. Increasing k or weighting by distance smooths the decision; the metric should match the geometry the feature extractor was trained for.

Configuration	Vote rule	Metric	Typical use
1-NN	single nearest neighbor	cosine	zero-cost sanity check of features
20-NN, uniform	majority of 20	cosine	standard frozen-feature evaluation
20-NN, distance-weighted	weight $\propto$ sim	cosine	same, less tie sensitivity
k-NN	majority of k	Euclidean	small feature spaces, raw pixels

shows but never transcribes. Third, and most important, we close the loop: from DeepCluster's alternation of k-means with cross-entropy training, through CRF energy functions, to STEGO's distillation of feature correspondences into unsupervised segmentations.

3.1 Nearest Neighbors: From Metric to Search

3.1.1 Why retrieval is the first SSL skill

The simplest use of a representation: look up neighbours. Once every image is a vector, the cheapest thing you can do — with no training at all — is find its nearest neighbours (Lecture 02, slides 3–5).

- **With labels:** classify a new image by a vote among its k nearest labelled images. That is the kNN classifier.
- **Without labels:** neighbours alone let you cluster, remove duplicates, and pick out good training data. Chapter 11's data engines do exactly this over hundreds of millions of images.

The slides show dogs turned into sets of vectors $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_N\}$, and comparing two images becomes measuring a distance between vectors: Euclidean distance or cosine similarity (slide 4). The lecture's rule for choosing (slide 5) is short: *use the distance your training loss used.* DINO is trained with cosine-based losses, so compare DINO features with cosine. (For normalised features the two agree anyway — Chapter 2 shows why.) Slide 5 adds the empirical remark behind that: papers often find little difference between the two for high-dimensional descriptors, because such features are usually normalised and spread over a hypersphere; the rule is to check what the training loss optimised. Two notation traps on slide 4: its Euclidean distance $d = \sqrt{(a-b)^2}$ means $\|a-b\|$, and its "cosine distance" $d = a \cdot b/(|a|\,|b|)$ is really the cosine *similarity* (large means close); the matching distance is $1 - \cos$.

kNN also deserves a second look because it is not one algorithm but a small configuration space. Table 3.1 collects the variants that appear across this book's evaluation protocols.

Figure 3.1 shows the vote in action on the lecture's toy dataset, the iris flowers, each reduced to two numbers (sepal length and width). A new flower landing in a coloured region is assigned that region's class. Two features of the picture are worth noticing. The

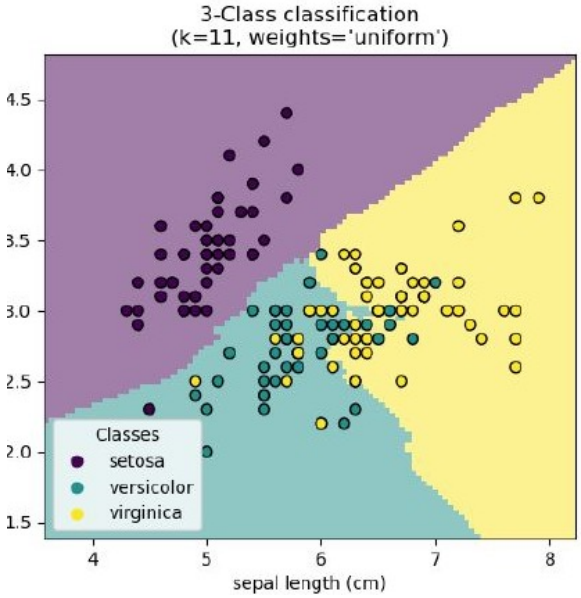


Figure 3.1. kNN deciding on the iris flowers with $k = 11$, uniform votes (Lecture 02, slide 3). Each point is one flower placed by two of its measurements; the shaded background is the region of the plane in which the indicated class would win the vote of the 11 nearest training flowers. Note the jagged, locally-assembled boundaries — the mark of a model made of stored examples — and the small smooth "bay" where the teal and yellow classes meet: many little votes outvoting one straggler.

Source: Lecture 02, slide 3 (plot from the scikit-learn documentation).

boundary between classes is not one smooth stroke: it is a jagged path pieced together from the perpendicular bisectors of neighbouring training points. And a straggler of one class sitting inside another class's territory only deforms the boundary in its own small patch — with $k = 11$ voters, one odd example cannot flip a whole region. Both are the signatures of a rule that memorises its training set rather than summarising it: the trained model *is* the training set.

In plain words — kNN: keep every answer sheet, vote at the last second

kNN does no training at all, and that is exactly why it is useful. You put every labelled example's descriptor on a shelf — in a search index that can answer "who is near me?" quickly — and all the work happens when a new query arrives: find the k shelved examples closest to it and let them vote on a class. Nothing is compressed, averaged, or forgotten, so the rule can trace very wiggly class boundaries. The price is paid at query time and in memory, which is why the index structures of the next two subsections matter.

```
# build phase: store labeled descriptors in a search index
index = build_index(descriptors, labels)   # KD-tree or LSH tables

def knn_classify(query, k):
    nbrs  = index.query(query, k)        # the k closest (x_i, y_i)
    votes = Counter(y_i for (x_i, y_i) in nbrs)  # or weight each by sim(q, x_i)
    return argmax_class(votes)           # majority class of the k
```

Listing 3.1. The kNN classifier in one page. There is no training loop: indexing the labelled descriptors is the whole build phase. The metric (Euclidean or cosine) is supplied by the caller — in this book it is chosen to match the geometry the feature extractor was trained with (slide 5).

Worked Example — One query, three different k's

A query descriptor sits at $q = (2.0, 2.0)$, among six labelled examples (two classes, A and B; distances computed by the Euclidean rule):

example	$(2.3, 1.9)$	$(1.5, 1.2)$	$(2.6, 2.8)$	$(1.0, 2.5)$	$(4.0, 4.0)$	$(4.5, 3.5)$
label	B	A	A	A	B	B
$d(q, \cdot)$	0.32	0.94	1.00	1.12	2.83	2.92

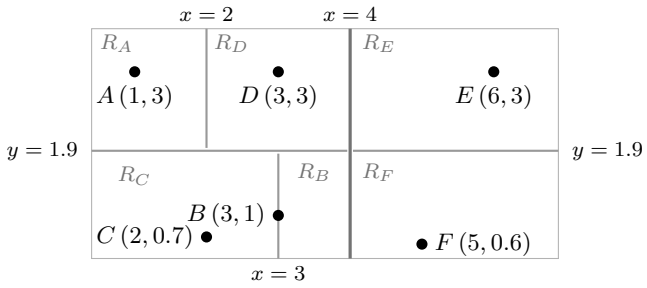
(For instance $d(q, (2.3, 1.9)) = \sqrt{0.3^2 + 0.1^2} = \sqrt{0.10} \approx 0.32$.) Now ask the classifier three times: $k = 1$ trusts the single closest example — the B point at 0.32, which is an odd B sitting deep inside A's territory — and answers B: wrong, fooled by the outlier. $k = 3$ consults B, A, A and answers A by two votes to one. $k = 5$ adds one more A and one more B, and confirms A by three to two. The lesson in one line: a single nearest neighbour can be a fluke; growing k averages the flukes away at the cost of blurring fine boundary detail — which is exactly the trade Table 3.1 lists configurations against.

Common Confusion

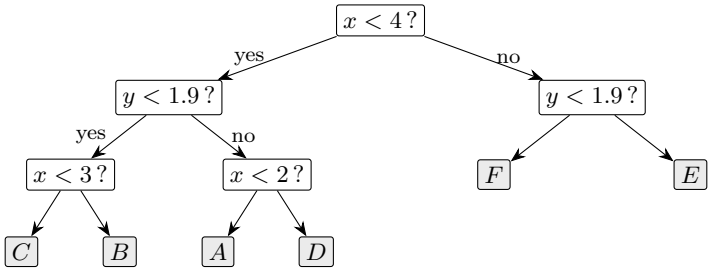
The two k's; and exact vs. approximate. The K of k-means is the number of *clusters* — a property the method imposes on the dataset. The k of kNN is the number of *neighbors consulted for one query* — a smoothing parameter of a classifier. They share a letter and nothing else: there is no notion of a centroid in kNN, and no notion of voting in k-means. A second confusion, *exact versus approximate nearest neighbor*, hides a steeper cliff than the phrase suggests: an ANN index does not return a slightly imperfect neighbor list that degrades gracefully — below recall 1.0, some true neighbors are *absent entirely*, because the retrieval itself is probabilistic (Section 3.1.3). For evaluation and data curation that distinction, not the small distance error, is what matters.

3.1.2 KD-trees: exact search, geometrically

Checking every point costs $O(Nd)$ per query. A KD-tree (Bentley, 1975) sorts the points in advance into a tree of straight cuts (Lecture 02, slides 6–7, after Carl Kingsford, CMU):



(a) the partitioned space: six points, five cuts, six cells



(b) the binary tree over the same splits (left child = test true)

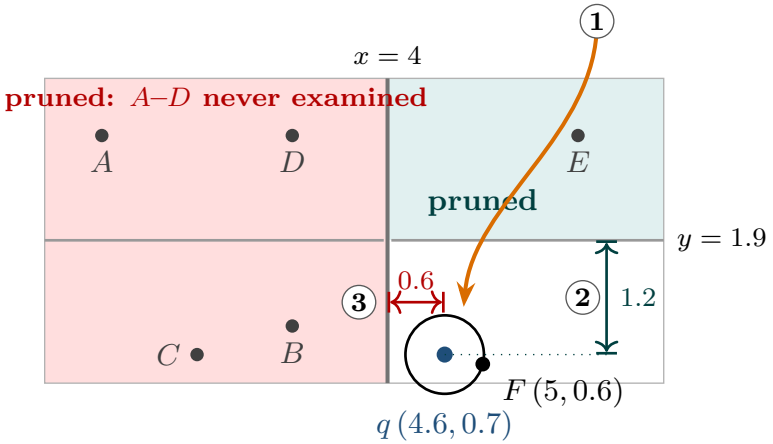
Figure 3.2. A KD-tree over six points. (a) The plane, cut by axis-aligned splits (first $x = 4$, then $y = 1.9$ on each side, and so on); the $R_{\cdot}$ regions are the spatial cells owned by each leaf. (b) The binary tree encoding exactly the same splits: internal nodes carry the split test, the left child is the "test true" branch, and leaves (shaded) hold the points — the two drawings are one data structure seen twice. Insertion of a new point is a descent by these tests followed by appending a leaf (Lecture 02, slides 6–7, illustrations after Carl Kingsford, CMU).

- each branch point cuts space along one coordinate;
- the leaves hold the data points;
- to insert a point such as $(15, 80)$, compare coordinates at each branch and walk down to a leaf.

A query then takes about $O(\log N)$ steps. Figure 3.2 shows a KD-tree over six points — the cut-up plane and the tree are two pictures of the same thing.

The query algorithm (slides 8–9) is descent plus *backtracking*: go down as for insertion, tracking the best candidate seen so far; then unwind the path, and at each node compare the current best distance against the distance from the query to the node's *splitting plane*. If the plane is farther than the best candidate, the entire other subtree is discarded — this is the moment the tree earns its keep. Figure 3.3 makes the moment concrete.

The lecture's own KD-tree, built and searched. Slides 6–9 use two concrete trees; working them by hand is the fastest way to own the algorithm, and both are classic exam material.



1 descend as for insertion: $x < 4$? no, $y < 1.9$? yes $\Rightarrow$ leaf F, best distance $r = d(q, F) \approx 0.41$ (the circle)

2 unwind at $y = 1.9$: gap $1.2 > r \Rightarrow E$'s cell cannot hold anything closer — pruned

3 unwind at $x = 4$: gap $|4.6 - 4| = 0.6 > r \Rightarrow$ the whole left subtree is pruned

Figure 3.3. The backtracking (circle-pruning) test on the tree of Figure 3.2. The query $q = (4.6, 0.7)$ descends right ($x \geq 4$), then down ($y < 1.9$), reaching the leaf $F = (5, 0.6)$ at distance $r \approx 0.41$. Unwinding: E's cell lies beyond the $y = 1.9$ plane, a gap of $1.2 > r$, so it is pruned; at the root, the whole left subtree lies beyond the $x = 4$ plane, a gap of $0.6 > r$, so A–D are never examined. One scalar comparison per node certifies F as the *exact* nearest neighbor after inspecting a single leaf (Lecture 02, slides 8–9; the slides' worked example uses the query $(52, 52)$).

Worked Example — Building by insertion (slides 6–7)

The slide inserts the points *in the given order* (it does not use the median trick of Listing 3.2): $(30, 40)$, $(5, 25)$, $(10, 12)$, $(70, 70)$, $(50, 30)$, $(35, 45)$. The split axis alternates by depth: x at the root, y at depth 1, x at depth 2, y at depth 3.

1. $(30, 40)$ becomes the root (splits on $x = 30$).
2. $(5, 25)$: $5 < 30$, go left; empty, so it becomes the left child (splits on $y = 25$).
3. $(10, 12)$: $10 < 30$ left; at $(5, 25)$ compare y: $12 < 25$, left child of $(5, 25)$ (splits on x).
4. $(70, 70)$: $70 \geq 30$, right child of the root (splits on y).
5. $(50, 30)$: right of the root; at $(70, 70)$: $30 < 70$, left child of $(70, 70)$ (splits on x).
6. $(35, 45)$: right of the root; at $(70, 70)$: $45 < 70$, left; at $(50, 30)$ compare x: $35 < 50$, left child of $(50, 30)$.

Inserting $(15, 80)$ (slide 7): $15 < 30$ left; at $(5, 25)$ compare y: $80 \geq 25$, right — the empty right slot of $(5, 25)$. Two comparisons, and the new point sits next to $(10, 12)$ (Figure 3.4). Note the price of insertion order: nothing keeps the tree balanced, so a sorted insertion order would build a chain of depth N.

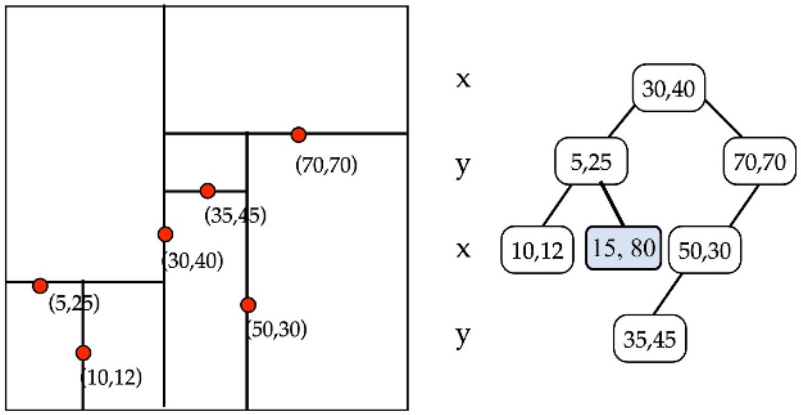


Figure 3.4. The lecture's KD-tree after the six insertions and the new point $(15, 80)$ (shaded). Left: the plane; each point draws its split line through its own cell (vertical for x-splits, horizontal for y-splits). Right: the tree; the letters on the left give the split axis of each depth.

Source: Lecture 02, slide 7 (illustration by Carl Kingsford, CMU).

Worked Example — Searching for the nearest neighbour of $(52, 52)$ (slides 8–9)

The tree of Figure 3.5: root $(51, 75)$ splits on x; its children $(25, 40)$ and $(70, 70)$ split on y; below them $(10, 30)$, $(35, 90)$, $(55, 1)$, $(60, 80)$ split on x; the leaves are $(1, 10)$ and $(50, 50)$. The slide's algorithm computes a distance at *every* visited node, not only at leaves.

step	node	what happens	best so far
1	$(51, 75)$	$d = \sqrt{1^2 + 23^2} = 23.0$; $52 \geq 51$: go right	$(51, 75)$, 23.0
2	$(70, 70)$	$d = \sqrt{18^2 + 18^2} = 25.5$; $52 < 70$: go left	unchanged
3	$(55, 1)$	$d = \sqrt{3^2 + 51^2} = 51.1$; leaf	unchanged
4	unwind at $(70, 70)$	gap to split $\lvert 52 - 70 \rvert = 18 < 23.0$: other side may hold a closer point	
5	$(60, 80)$	$d = \sqrt{8^2 + 28^2} = 29.1$	unchanged
6	unwind at root	gap $\lvert 52 - 51 \rvert = 1 < 23.0$: **must** search the left half	
7	$(25, 40)$	$d = \sqrt{27^2 + 12^2} = 29.5$; $52 \geq 40$: go right	unchanged
8	$(35, 90)$	$d = \sqrt{17^2 + 38^2} = 41.6$; $52 \geq 35$: go right	unchanged
9	$(50, 50)$	$d = \sqrt{2^2 + 2^2} = 2.8$	$(50, 50)$, **2.8**
10	unwind at $(35, 90)$	gap $\lvert 52 - 35 \rvert = 17 > 2.8$: prune (empty anyway)	
11	unwind at $(25, 40)$	gap $\lvert 52 - 40 \rvert = 12 > 2.8$: prune $(10, 30)$ and $(1, 10)$	

Answer: $(50, 50)$ at distance 2.8, after visiting 7 of the 9 nodes. The lesson of this example is the *opposite* of Figure 3.3: the true nearest neighbour lies on the *far* side of the root's split, because the query sits just 1 unit from the line $x = 51$. A search that

only descended (steps 1–3) would answer $(51, 75)$ — eight times too far. Backtracking is not an optimisation; it is what makes the answer exact.

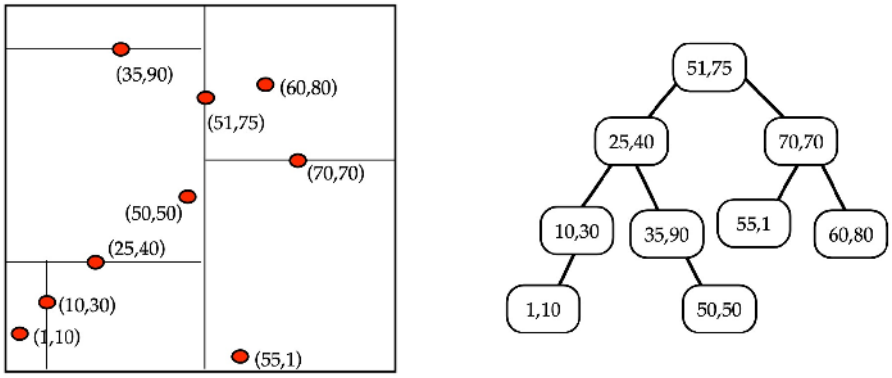


Figure 3.5. The lecture's search example: find the nearest neighbour of $(52, 52)$. The descent goes right at the root, but the answer, $(50, 50)$, sits just left of the root's vertical split line.

Source: Lecture 02, slide 9 (illustration by Carl Kingsford, CMU).

The same two figures, as code: the build is a recursive median split, and the query is one descent plus one scalar test per unwind (Listing 3.2).

```python
def build(points, depth=0):
    if len(points) <= leaf_size: return Leaf(points)
    a    = widest_axis(points, depth)      # split coordinate (slides: x, then y)
    m    = median(points, axis=a)          # the drawn planes x=4, y=1.9, ...
    near = {p for p in points if p[a] <  m}
    far  = {p for p in points if p[a] >= m}
    return Node(a, m, build(near, depth+1), build(far, depth+1))

def nearest(node, q, best=None):
    if is_leaf(node): return closest_so_far(best, node.points, q)
    near, far = order_children(node, q[node.a])   # near = the side q falls on
    best = nearest(near, q, best)                 # descend as insertion would
    if abs(q[node.a] - node.m) < dist(q, best):   # ball crosses the split plane?
        best = nearest(far, q, best)              # only then is far side live
    return best
```

Listing 3.2. KD-tree build and exact nearest-neighbor query. The whole value of the tree lives in one line of `nearest`: visit the far side *only* if the best-so-far ball crosses the split plane — the single comparison Figure 3.3 draws as two dashed gap arrows.

Failure Modes

KD-trees. The prune in Figure 3.3 requires the query ball to *miss* the splitting plane. In d dimensions the ball touches a growing fraction of the exponentially many cells, and distance concentration keeps best distances large, so fewer and fewer subtrees are prunable: beyond

roughly $d \approx 20$ a KD-tree degenerates toward a *linear scan* — precisely the dimension range of even modestly pooled deep features, let alone 384- or 768-dimensional ViT descriptors. Ball trees help marginally; the practical escapes are to reduce dimension first (PCA, Section 3.3) or to abandon exactness (Section 3.1.3).

3.1.3 Locality-sensitive hashing: probabilistic search

Locality-sensitive hashing, in one sentence. LSH (Indyk and Motwani, 1998; Gionis et al., 1999) gives up on *always* finding the nearest neighbour, in exchange for *usually* finding it very fast.

The promise, made precise. A hash function h is *locality-sensitive* if close points usually get the same hash and far points usually do not (slide 10):

$$d(\boldsymbol{a}, \boldsymbol{b}) \le r \;\Rightarrow\; p[h(\boldsymbol{a}) = h(\boldsymbol{b})] \ge p_1, \qquad d(\boldsymbol{a}, \boldsymbol{b}) \ge c\,r \;\Rightarrow\; p[h(\boldsymbol{a}) = h(\boldsymbol{b})] \le p_2,$$

with $p_1 > p_2$. Read it as: within distance r, the chance of sharing a hash is *at least* p_1; beyond $c\,r$, it is *at most* p_2.

The standard hash for cosine distance. Pick a random direction $\boldsymbol{r}$ and record which side of it a vector falls on: $h_{\boldsymbol{r}}(\boldsymbol{u}) = \operatorname{sign}(\boldsymbol{r}^\top \boldsymbol{u})$. Two vectors at angle θ land on the same side with probability $1 - \theta/\pi$: identical vectors always agree, opposite ones never do.

The search (slide 11). Stack k such bits into a short code, and build L separate tables with different random directions. To search, hash the query into every table, collect everything that shares its code in any table, and compute exact distances only for that small collection. Slide 11's third step adds the stopping rule: visit the tables one after another and stop as soon as a point within a chosen *stop margin* of the query has been found, or when no tables are left — a query with a very close match is answered after the first table or two. Figure 3.6 follows one vector through the whole process.

Failure Modes

LSH. (i) *False negatives*: a genuinely close pair can land in different buckets in *every* table, and no post-hoc checking recovers a pair that was never a candidate; raising L only shifts the trade-off. (ii) *Bucket imbalance*: random projections know nothing of the data distribution, so some buckets overflow while most stay empty, and degenerate bit patterns place pathological vectors far from every bucket. (iii) *Tuning*: (k, L) and the thresholds r, c must be set per dataset; the guarantees are probabilistic in the randomness of the hash functions and say nothing about a worst-case query.

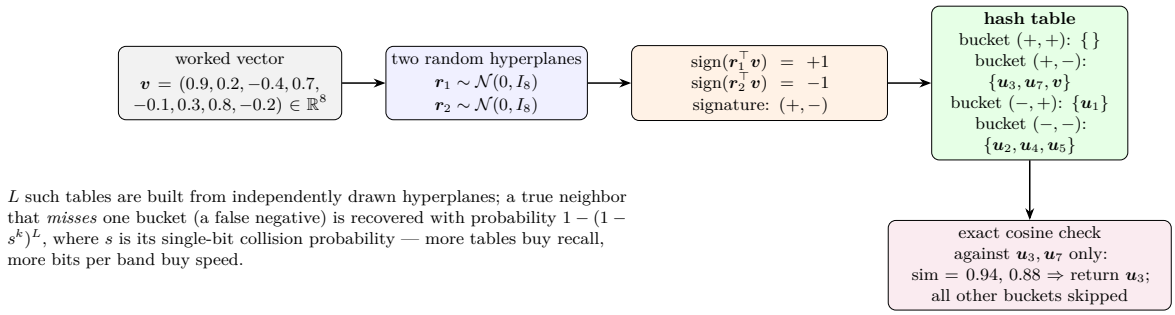


Figure 3.6. LSH banding on one worked 8-dimensional vector. The vector is projected onto $k = 2$ random hyperplanes; the two signs form its band signature and route it to one of 2^k buckets. Exact distances are computed only inside that bucket — here three points survive out of eight. With L independent tables a query fans out to L buckets, trading computation against recall (Lecture 02, slides 10–11).

Table 3.2. The approximate-nearest-neighbor landscape as charted by ann-benchmarks.com (Lecture 02, slide 12). Recall is the fraction of true neighbors returned; QPS is query throughput. Graph methods dominate at high recall; quantization wins on memory; KD-trees are exact but only in low dimension.

Family	Example systems	Core idea and trade-off character
Hashing (LSH)	FALCONN-style	bucket by random projections; light memory, recall limited by L
Graph (HNSW)	hnswlib, NMSLIB	greedy walk on a proximity graph; highest recall/QPS, heavy index
Quantization	Faiss IVF-PQ, ScaNN	compress vectors, search codebooks; tiny memory, GPU-scaled
Tree-based	Annoy, FLANN	random-projection trees; fast builds, moderate recall
Exact	KD-tree, brute force	full or pruned scan; recall = 1, low dimensions only

3.1.4 The modern ANN landscape

The lecture's slide 12 is a screenshot of ann-benchmarks.com, and it makes a single visual point worth internalizing: on recall-versus-throughput axes, practical ANN systems span *orders of magnitude* of queries per second at fixed recall. Table 3.2 summarizes the families that populate those plots. We quote no benchmark numbers — the curves are dataset- and hardware-dependent — but the family-level trade-offs are stable.

The line to remember for the rest of the book is that *everyone accepts approximation*: curated retrieval over the hundred-million-image pools of Chapter 11's data engines runs on indexes of this kind (Faiss-style quantised and graph indexes), comparing millions of vectors by cosine similarity.

Key takeaway — Retrieval is the tax every SSL system pays

Every method in this book eventually has to answer "which other points are near this one?" — to pair images, to curate a dataset, to score a frozen backbone with a k-NN probe. In low dimension you can answer it exactly and cheaply with a KD-tree. Above roughly twenty dimensions you cannot: distances concentrate, the pruning test stops firing, and exact search degenerates into a scan of everything. Deep features are hundreds of dimensions wide, so *everybody* gives up exactness and buys recall with hashing, graphs or quantisation instead. Remember the shape of that trade, not the individual systems: approximate search does not return a slightly-wrong neighbour, it sometimes returns no neighbour at all.

3.2 Clustering as a Signal Generator

Retrieval assumes something to compare against; clustering needs nothing but the points. "Unsupervised classification" (slide 13) groups data so that within-group similarity exceeds between-group similarity — and slide 14's framing is the honest one: there is no perfect clustering algorithm, and the choice is a function of the problem's parameters: *do I know the number of clusters? How complex is the problem? Does all the data fit in RAM? How much time do I have?* The two workhorses answer these four questions differently enough to be complementary.

3.2.1 The clustering zoo on one slide

Slide 13 opens the clustering part with the scikit-learn comparison grid (Figure 3.7): eleven algorithms (columns) run on six toy data sets (rows). The slide shows it without comment; its point is slide 14's sentence — *there is no perfect clustering algorithm* — and it is worth reading row by row, because every failure in it is caused by one assumption.

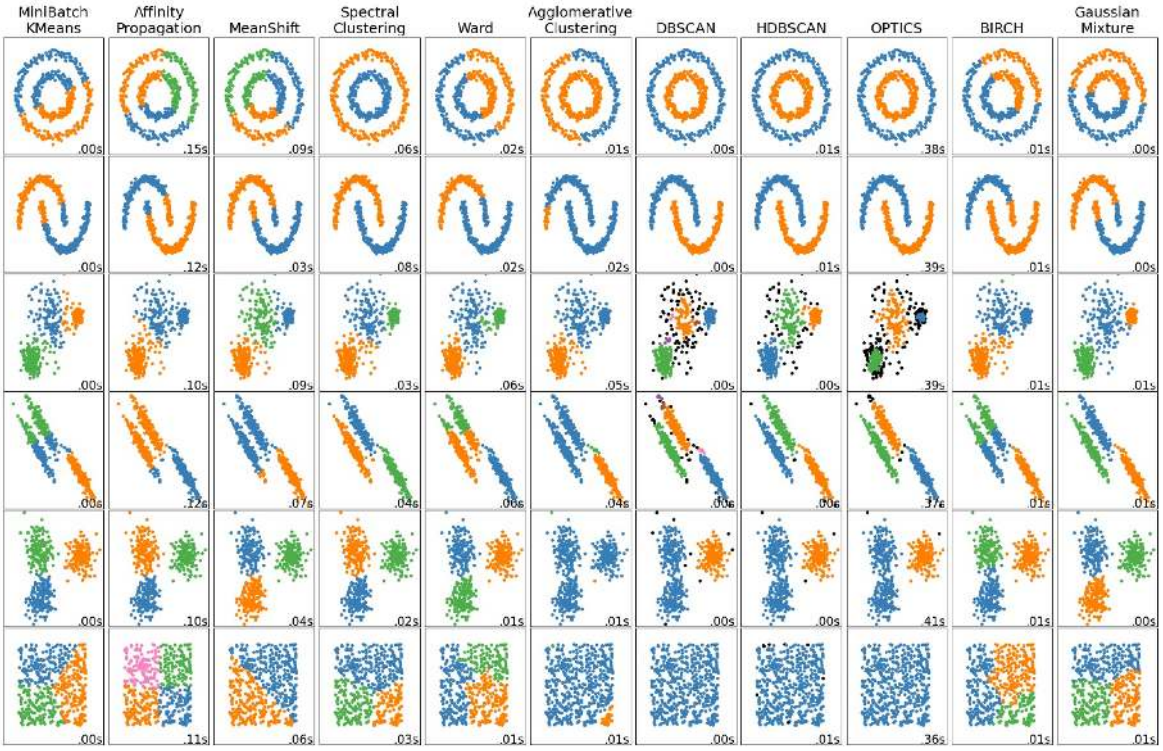
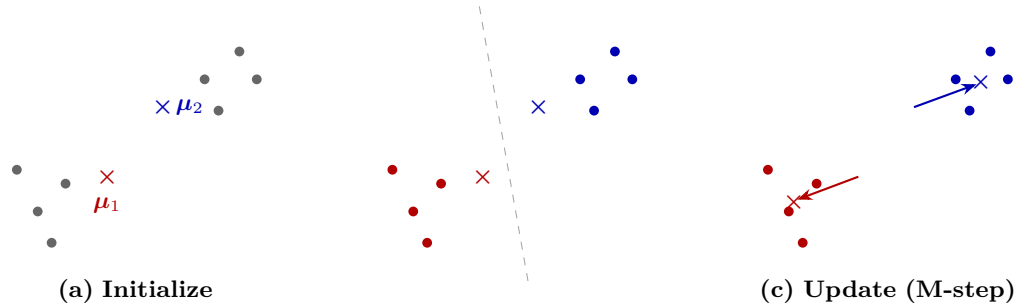


Figure 3.7. Eleven clustering algorithms on six toy data sets. Rows, top to bottom: two concentric circles; two moons; three blobs of very different spread; three stretched (anisotropic) blobs; three well-separated round blobs; uniform noise with no structure. Colours are the found clusters; black points (DBSCAN, HDBSCAN, OPTICS) are marked as noise. The number in each corner is the run time.

Source: Lecture 02, slide 13 (scikit-learn clustering comparison).

How to read it: Pick one column and go down it: which rows does it get right? Then pick one row and go across: which assumption separates the winners from the losers?

Algorithm	**needs K?**	**Idea in one line**	**In the grid**
Mini-batch k-means	yes	nearest centroid, centroids = means (§3.2.4)	round blobs only; cuts circles, moons and even noise
Affinity propagation	no	points "vote" for exemplars by passing messages	too many clusters on noise; slow ($O(N^2)$)
Mean shift	no	move every point uphill on a kernel density; points that reach the same peak form a cluster	round blobs; fails on rings/moons
Spectral clustering	yes	cluster the eigenvectors of a similarity graph	separates circles and moons (connectivity, not distance to a centre)
Ward	yes	agglomerative merging that adds the least variance	behaves like k-means
Agglomerative (average link)	yes	repeatedly merge the two closest clusters (here restricted to a nearest-neighbour graph)	follows chains of neighbours: gets the moons;



(a) Initialize
seeds (e.g. k-means++):
$\boldsymbol{\mu}_1 = (2.2, 1.5)$,
$\boldsymbol{\mu}_2 = (3.0, 2.5)$

(b) Assign (E-step)
color each point by
nearest centroid;
the dashed bisector is
the decision boundary

(c) Update (M-step)
centroids jump to
their clusters' means:
$\boldsymbol{\mu}_1 = (1.28, 1.14)$,
$\boldsymbol{\mu}_2 = (3.96, 2.86)$

Figure 3.8. One Lloyd iteration on eight points (cf. Lecture 02, slide 15). Assignment colors every point by its nearest centroid; the M-step recomputes $\boldsymbol{\mu}_j = \frac{1}{|X_j|} \sum_{\boldsymbol{x} \in X_j} \boldsymbol{x}$; the loop stops when the assignment stops changing. The assignment boundary in (b) is the perpendicular bisector of the centroids — the Voronoi view that makes the spherical geometry baked into the method visible.

Reading the grid in three lessons.

- **Centre-based methods** (k-means, Ward, BIRCH, mean shift) assume clusters are round blobs around a centre. They cut rings and moons in half, and they split structureless noise into K pieces anyway (last row) — they never answer "no structure".
- **Connectivity- and density-based methods** (spectral, graph-based agglomerative, DBSCAN family) follow chains of close points, so they get rings and moons. The density family can also say "noise" (black) and "one cluster" for the last row.
- **Model-based** Gaussian mixtures allow each cluster its own shape and size (a covariance matrix), which wins on stretched and uneven blobs but still assumes each cluster is one ellipse.

The course keeps two of the eleven — k-means and DBSCAN — because they sit at opposite ends of these trade-offs. Slide 14's four questions decide between them: *Do I know K? How complex is the cluster shape? Does the data fit in RAM? How much time do I have?*

3.2.2 k-means in action

Lloyd's algorithm for k-means — alternate assigning each point to its closest centroid $\boldsymbol{\mu}_j$ with resetting $\boldsymbol{\mu}_j = \frac{1}{|X_j|} \sum_{\boldsymbol{x} \in X_j} \boldsymbol{x}$, where X_j is the set of points currently assigned to cluster j and K the (fixed) number of clusters — is derived and worked through by hand in Chapter 2. Figure 3.8 re-stages three phases of one run on eight two-dimensional points, corresponding to the iteration panels of slide 15.

In plain words — k-means: move the depots until nobody switches

Think of the K centroids as delivery depots and the points as customers. Every customer is served by the closest depot (the assign step), then each depot is relocated to the exact centre of its own customers (the update step). Relocating moves the boundaries, so some customers find a different depot closer and switch — which moves the centres again. Round and round, until one full pass produces no switches, and the configuration freezes. Two facts fall straight out of the rules. The boundary between any two groups is the straight line halfway between their centres — the dashed bisector of Figure 3.8(b) — which is why k-means carves round, equally-sized-looking cells and nothing else. And the loop can never climb out of a bad arrangement it has settled into: the seeds decide which frozen states are even reachable, a concern that earns its own treatment in Section 3.2.3.

```
mu = seed_centroids(X, K)            # k-means++ or random

while assignments change:
    # E-step: hand every point to its closest centroid
    for x in X:
        c[x] = argmin_j sqdist(x, mu[j])

    # M-step: move each centroid to the mean of its members
    for j in 1..K:
        X_j   = {x : c[x] == j}
        mu[j] = mean(X_j)            # empty X_j: re-seed it somewhere
```

Listing 3.3. Lloyd's algorithm, the loop that Figure 3.8 draws once. Each round is one E-step (hand every point to its closest centroid) and one M-step (move each centroid to the average of its members); the loop stops when the assignment stops changing. Seeding is done by k-means++ where available (Section 3.2.3); plain random seeding also runs, at the risk measured in the worked example next.

Worked Example — Same four points, two seedings — sixteen times the cost

Four points sit at the corners of a rectangle: $(0,0)$, $(0,2)$, $(8,0)$, $(8,2)$. Run Listing 3.3 twice with $K = 2$.

Good seeds, $(0,0)$ and $(8,0)$ — one in each column. $(0,2)$ joins seed 1 (its distance to seed 2 is $\sqrt{8^2+2^2} \approx 8.2$, versus 2), $(8,2)$ joins seed 2, and the update moves the centroids to the column means $(0,1)$ and $(8,1)$. Next round no point switches: done. Total squared distance to the assigned centroids: $1+1+1+1 = 4$, and the partition found is the true pair of columns.

Bad seeds, $(0,0)$ and $(0,2)$ — both in the left column. Now $(8,0)$ is closer to seed 1 ($8.0 < \sqrt{68} \approx 8.2$) and $(8,2)$ to seed 2, so the split runs along *rows*: $\{(0,0),(8,0)\}$ and $\{(0,2),(8,2)\}$. The means become $(4,0)$ and $(4,2)$, and no point can ever switch back — $(0,0)$ sits 4.0 from its own centroid and 4.5 from the other, similarly all round. Converged,

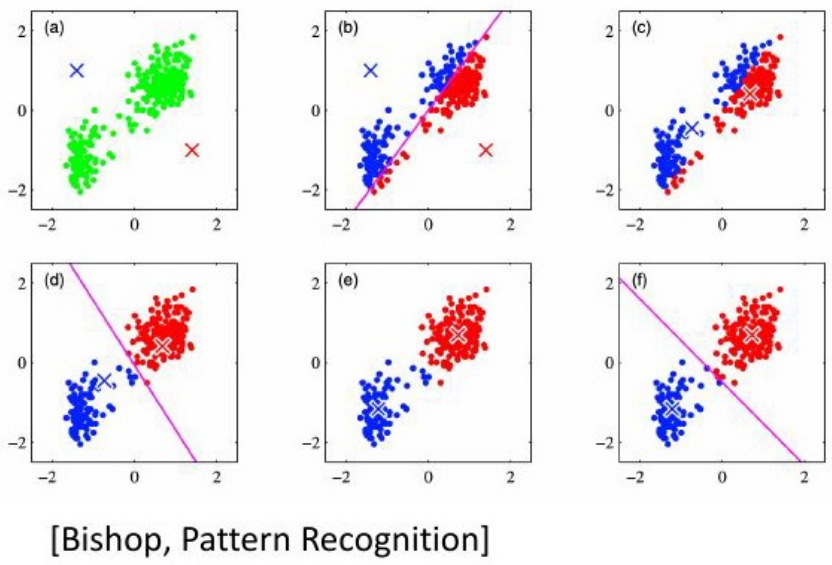


Figure 3.9. The lecture's run of Lloyd's algorithm (Lecture 02, slide 15), tracked end to end. (a) Two seeds dropped in the cloud. (b) First assignment: every point coloured by its closer seed; the magenta line is the halfway boundary. (c) Update: the centroids jump to the means of their coloured sets. (d) Second assignment. (e) Second update. (f) An assignment that changes nothing — the loop has converged. Compare the three sketched phases of Figure 3.8; the bisector drawn here in every assignment panel is the feature our sketch collapsed into a single dashed line.

Source: Lecture 02, slide 15 (panels after Bishop, Pattern Recognition*).*

at total cost $16 + 16 + 16 + 16 = 64$: sixteen times worse, on data whose structure is visible at a glance.

Same points, same algorithm, same K. The seeds alone decided which of the two stable arrangements the loop could reach, and no step inside Lloyd's corrects the choice. That is the failure the next subsection engineers away.

Historical Note

Three births of k-means. Steinhaus posed the partition problem for continuous bodies in a 1956 note on dividing material bodies into parts (Steinhaus, 1956); Lloyd circulated the assign–update alternation as a 1957 Bell Labs report on pulse-code modulation, published only in 1982 (Lloyd, 1982); MacQueen coined and analyzed the name "k-means" in 1967 (MacQueen, 1967). The seeding refinement the slides credit, k-means++, is Arthur and Vassilvitskii's (Arthur and Vassilvitskii, 2007): sample each new seed with probability proportional to its squared distance from the nearest existing seed — spreading seeds pays off with an $O(\log K)$ approximation guarantee (slide 16). The final scaling step on the slides is Sculley's *mini-batch* k-means (Sculley, 2010), which updates centroids

stochastically from small batches with a per-centroid learning rate decaying in the count of points seen (slide 17) — the web-scale variant on which the hybrid pipelines below implicitly rely.

3.2.3 Seeding matters: k-means++

Why seeding matters. In the worked example above, the same data with two different starting points gave costs of 4 and 64 — sixteen times apart. The naive start (slide 16) picks K random data points; with many clusters it will often put two seeds in one blob and none in another.

k-means++ (Arthur and Vassilvitskii, 2007) replaces that with one rule: *choose each new seed with probability proportional to its squared distance from the nearest seed chosen so far.* Far-away points are much more likely, so the seeds spread out. Writing $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_j$ for the seeds chosen so far and $D(\boldsymbol{x})$ for a point's distance to the closest one,

$$p(\text{next seed} = \boldsymbol{x}) \;=\; \frac{D(\boldsymbol{x})^2}{\sum_{\boldsymbol{x}'} D(\boldsymbol{x}')^2}, \qquad D(\boldsymbol{x}) = \min_{\ell \le j} \|\boldsymbol{x} - \boldsymbol{\mu}_\ell\|. \tag{3.1}$$

The first seed alone is drawn uniformly; from then on rule (3.1) draws one seed at a time until K exist — and everything after that is ordinary k-means unchanged.

In plain words — Squaring the distance loads the dice

Read $D(\boldsymbol{x})^2$ as a lottery in which every data point holds tickets, and the number of tickets is the point's squared distance to the nearest seed already on the board. The squaring is the whole trick: a point twice as far away holds *four* times as many tickets, so untouched regions win the draw overwhelmingly and the seeds come out spread over different dense areas. Two things about the rule are easy to miss. The point that was just seeded holds zero tickets — you never re-spend a seed where one already stands. And the whole effort is spent once, up front: each draw is one pass over the data, so seeding K clusters costs about what K ordinary assign steps would cost, and buys a starting position that naive sampling matches only by getting lucky.

Worked Example — One k-means++ draw, start to finish

Four points on a line, 0, 1, 10, 11 — two tight pairs — with $K = 2$. *Step 1a*: the first seed is uniform; suppose it lands on 1. *Step 1b*: measure each point's gap to the nearest (here only) seed: $D = (1, 0, 9, 10)$ for the points in order. Square the gaps into tickets: $D^2 = (1, 0, 81, 100)$, a pool of 182. Normalise:

$$p \;=\; \frac{(1, 0, 81, 100)}{182} \;\approx\; (0.5\%,\ 0\%,\ 44.5\%,\ 54.9\%),$$

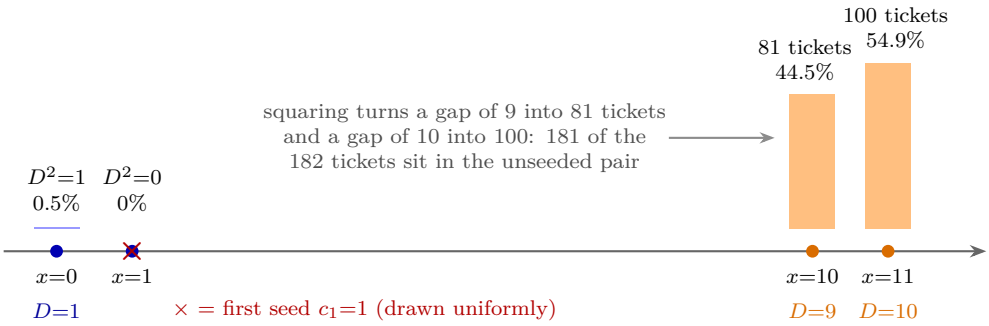


Figure 3.10. k-means++ drawing its second seed on four points (worked through, with the same numbers, in the worked example below). The first seed landed on the blue pair and holds no lottery tickets; tickets grow with the *squared* gap to the nearest seed, so the orange pair takes 99.5% of the draw. Whatever wins it, Lloyd's loop then runs one seed per pair and converges immediately — the opposite of the sixteen-times-the-cost configuration produced by unlucky seeding one subsection earlier.

so the second seed lands in the right-hand pair with probability 99.5%. Compare with the naive uniform lottery: whatever the first seed was, one of the three remaining points shares its pair, so under uniform sampling two seeds collide in one pair with probability $\frac{1}{3}$. *Steps 2–4*: with seeds at 1 and 10 (say), Lloyd's first assignment is $\{0, 1\}$ and $\{10, 11\}$, the update moves the centroids to 0.5 and 10.5, and nothing changes after that — converged in one round at total cost $4 \times 0.25 = 1$. One careful draw bought what the worked example of Section 3.2.2 showed two uniform draws can fail to.

```
def kmeans_pp(X, K):
    seeds = [sample_uniform(X)]              # step 1a: first seed, uniform
    for _ in 1..K-1:                         # step 1b: one weighted draw per seed
        D2 = [min(sqdist(x, s) for s in seeds) for x in X]
        seeds.append(sample(X, weights=D2))  # p(x) ~ D(x)^2
    return seeds                             # step 2-4: hand these to Lloyd
```

Listing 3.4. k-means++ seeding: nothing but a uniform first draw and $K-1$ squared-distance-weighted draws; the list it returns seeds the ordinary Lloyd loop of Listing 3.3 (and it is the default in scikit-learn's and Faiss's k-means, the libraries a DeepCluster-style pipeline actually calls).

Two cautions about k-means++:

- **It only chooses the starting points.** Lloyd's loop still runs afterwards, and the $O(\log K)$ guarantee is an *average* over the random choices. Any single run can still be unlucky, so implementations run several times and keep the best result.
- **It weakens in high dimensions**, for the same reason KD-trees do (Section 3.1.2): when all distances look alike, "far away" stops meaning much.

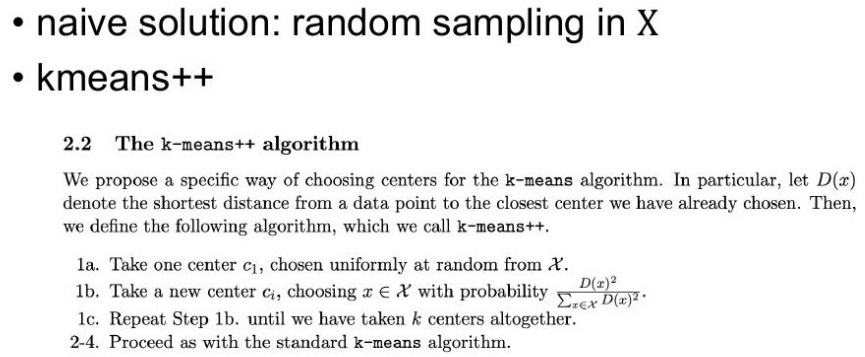


Figure 3.11. The seeding rule as the original paper states it, quoted on Lecture 02, slide 16. Steps 1a–1c produce the K seeds by squared-distance sampling; steps 2–4 are Lloyd's algorithm untouched. Note how small the change is relative to naive uniform seeding — one weighted draw per seed — and how large the prize: the expected clustering cost lands within a factor of order $\log K$ of the best possible arrangement whatever the data (the history note states the guarantee).

Source: Lecture 02, slide 16 (quoting Arthur and Vassilvitskii (2007)).

It still matters here. DeepCluster's per-epoch k-means uses exactly such a default, and with $K = 10{,}000$ clusters, purely random starting points would leave whole regions of feature space without any centre.

3.2.4 Scaling up: k-means in mini-batches

Mini-batch k-means: the speed fix. Good seeding fixes where k-means starts, not how slow each round is. Every round reassigns *all* N points. DeepCluster clusters over a million vectors into 10,000 groups every epoch, so that full pass is the bottleneck.

Sculley's mini-batch k-means (Sculley, 2010) (slide 17, Figure 3.12) looks at only a small random batch M each round. Each point in the batch is sent to its nearest centre, and only that centre is nudged towards the point — by an amount that shrinks as the centre collects more points:

$$v_c \leftarrow v_c + 1, \qquad \eta \;=\; \frac{1}{v_c}, \qquad \boldsymbol{\mu}_c \leftarrow (1-\eta)\,\boldsymbol{\mu}_c + \eta\,\boldsymbol{x}, \tag{3.2}$$

where v_c counts how many points centroid $\boldsymbol{\mu}_c$ has already absorbed.

In plain words — The running mean, one visitor at a time

A centroid in Lloyd's algorithm is the mean of its members; equation (3.2) is the same mean computed online. A running mean weights the v-th arrival by $1/v$: the first visitor to a centroid moves it all the way to themselves, the hundredth moves it by one part in a hundred. Early points shout, late ones whisper, and after enough whispers the centroid sits exactly where the batch mean would have put it — but computed while streaming

Algorithm 1 Mini-batch k-Means.

1: Given: k, mini-batch size b, iterations t, data set X
2: Initialize each $\mathbf{c} \in C$ with an $\mathbf{x}$ picked randomly from X
3: $\mathbf{v} \leftarrow 0$
4: **for** $i = 1$ to t **do**
5: $M \leftarrow b$ examples picked randomly from X
6: **for** $\mathbf{x} \in M$ **do**
7: $\mathbf{d}[\mathbf{x}] \leftarrow f(C, \mathbf{x})$ // Cache the center nearest to $\mathbf{x}$
8: **end for**
9: **for** $\mathbf{x} \in M$ **do**
10: $\mathbf{c} \leftarrow \mathbf{d}[\mathbf{x}]$ // Get cached center for this $\mathbf{x}$
11: $\mathbf{v}[\mathbf{c}] \leftarrow \mathbf{v}[\mathbf{c}] + 1$ // Update per-center counts
12: $\eta \leftarrow \frac{1}{\mathbf{v}[\mathbf{c}]}$ // Get per-center learning rate
13: $\mathbf{c} \leftarrow (1 - \eta)\mathbf{c} + \eta\mathbf{x}$ // Take gradient step
14: **end for**
15: **end for**

Figure 3.12. Sculley's mini-batch k-means as the paper states it, on Lecture 02, slide 17. Every step of the batch E/M alternation has shrunk to a small random batch M: assignment is cached per point (line 7), and the M-step becomes the per-centroid running-mean update of Equation (3.2) (lines 12–13). The cost of a round drops from N point–centroid distances to b of them, at the price of a noisy gradient-like step — the same trade that minibatch SGD makes against full gradient descent, and not a coincidence: the two are the same idea applied to different objectives.

Source: Lecture 02, slide 17 (algorithm of Sculley (2010)).

past the data once, never holding the whole dataset at hand. That is what makes the per-epoch clustering step of the hybrids affordable.

Worked Example — Three arrivals, one centroid

A centroid is freshly seeded and then sees three points, in the order $20, 2, 8$. Apply (3.2) each time: the first arrival gets $\eta = 1/1 = 1$, so the centroid jumps from its seed to 20; the second gets $\eta = 1/2$, giving $\frac{1}{2}(20) + \frac{1}{2}(2) = 11$; the third gets $\eta = 1/3$, giving $\frac{2}{3}(11) + \frac{1}{3}(8) = 10$. Final position $\boldsymbol{\mu} = 10$ — exactly $\text{mean}(20, 2, 8)$, the answer a full Lloyd M-step over those three points would return. The learning rate $\eta = 1/v_c$ is not an arbitrary schedule: it is the unique choice that keeps the online update equal to the running batch mean, whatever the arrival order.

3.2.5 DBSCAN: density instead of centroids

DBSCAN (Ester et al., 1996) abandons centroids for density (slide 18). With a pairwise distance matrix A thresholded into a neighborhood matrix $N = \{A \leq \varepsilon\}$ — that is, $N_{ij} = 1$ exactly when the distance between points i and j is within the radius ε — a point is a *core point* when at least `min_samples` points lie within distance ε of it; clusters grow by depth-first

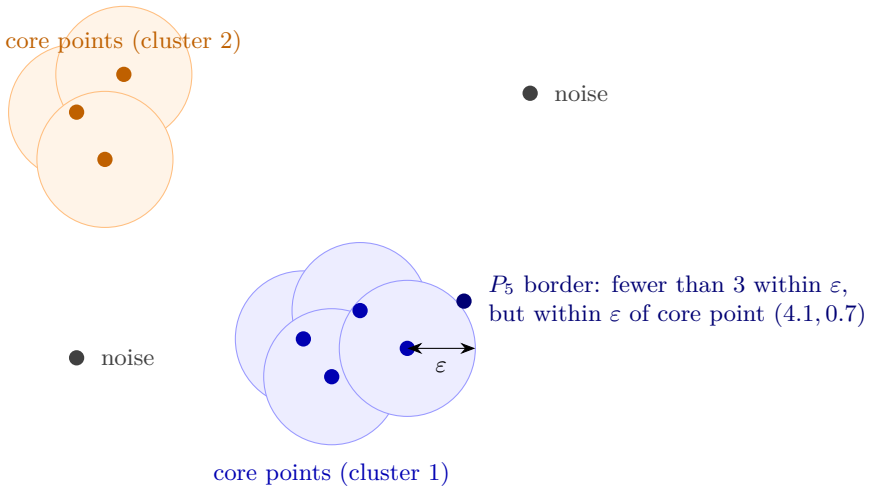


Figure 3.13. DBSCAN on ten points with `min_samples` $= 3$ (counting the point itself) and the shown radius ε (cf. Lecture 02, slide 18). Circles mark ε-neighborhoods of core points. The four blue core points are mutually within ε and merge into one cluster; P_5 fails the core test but touches a core point and joins as a *border* point; the top-left trio forms a second cluster; the two isolated points are honestly labeled noise rather than forced into a cluster.

search over connected core neighborhoods; border points attach to a neighboring core point; and everything else is declared an outlier. Figure 3.13 labels a ten-point layout.

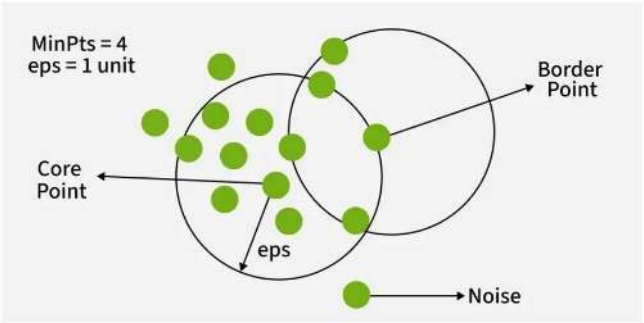


Figure 3.14. The lecture's DBSCAN picture, with `MinPts` $= 4$ and $\varepsilon = 1$ unit. The *core point* has at least 4 points in its ε-circle. The *border point* has fewer in its own circle but lies inside a core point's circle. The *noise* point is in nobody's circle.

Source: Lecture 02, slide 18 (image from an external tutorial, linked on the slide).

Reading slide 18's four steps precisely.

1. *Given* an affinity (distance) matrix A with $A_{ij} = d(\boldsymbol{x}_i, \boldsymbol{x}_j)$, a count `min_samples` and a radius ε.
2. *Neighbourhood matrix* $N = \{A \leq \varepsilon\}$: a 0/1 matrix, $N_{ij} = 1$ when i and j are within ε. For n points it has n^2 entries — the memory problem of slide 21.

3. *Core samples*: rows of N with enough ones. The slide says "$\geq$ `min_samples` *other* samples"; scikit-learn counts the point itself. The two conventions differ by one, so always state which you use (this book counts the point itself, as scikit-learn does).
4. *Depth-first search* from each unvisited core sample collects the connected core samples and their "leaves" (the border points). *Everything not reached is an outlier.*

In plain words — Chains of crowded circles

Draw a circle of radius ε around every point, and call a point *crowded* if its circle holds at least `min_samples` points (the point itself counts). A cluster is then nothing but a chain: start at any crowded point, hop anywhere inside its circle, and keep hopping out of every *crowded* point you reach, collecting everything the walk touches. A point the chain touches joins the cluster even if its own circle fails the crowd test — that is a *border* point — but it never extends the walk. Whatever no chain ever reaches is declared noise. That last verdict is the one k-means cannot give, and it is what rescues datasets full of strays.

Worked Example — Six points on a line: one cluster, two refusals

Take the points 0.0, 0.2, 0.35, 0.5 and 5.0, 5.2, with $\varepsilon = 0.25$ and `min_samples` $= 3$. Count each circle's contents: 0.0 reaches $\{0.0, 0.2\}$ — two points, too few. 0.2 reaches $\{0.0, 0.2, 0.35\}$ — three: *core*. 0.35 reaches $\{0.2, 0.35, 0.5\}$ — three: *core*. 0.5 reaches $\{0.35, 0.5\}$ — two. 5.0 reaches $\{5.0, 5.2\}$ and 5.2 reaches the same set — two each. Now grow: start at core 0.2; its circle hands the walk 0.0 (touch-only, joins as *border*) and 0.35 (crowded, so the chain extends); from 0.35 the walk reaches 0.5 (border). One cluster: $\{0.0, 0.2, 0.35, 0.5\}$. The far pair fares worse: neither point is crowded and neither touches a core circle, so 5.0 and 5.2 are noise — *even though they sit next to each other*; with `min_samples` $= 3$ no choice of ε can ever call the pair a cluster on its own. k-means at $K = 2$ would return the same split as the cluster verdict for the left four — but no K lets it say "nothing" for the right two.

```python
N    = {p: {q for q in X if dist(p, q) <= eps}  for p in X}   # threshold the affinities
core = {p for p in X if len(N[p]) >= min_samples}             # counting p itself
label = {p: UNSEEN for p in X}

for p in X:
    if label[p] != UNSEEN: continue
    if p not in core:
        label[p] = NOISE                # provisional: may be flipped to border below
        continue
    cid = new_cluster_id(); label[p] = cid
    stack = [p]                         # grow the chain depth-first
    while stack:
        q = stack.pop()
        for r in N[q]:
            was = label[r]
            if was != cid: label[r] = cid                  # border or unseen, both join
            if was == UNSEEN and r in core: stack.append(r)  # only fresh cores extend
```

Listing 3.5. DBSCAN in one pass over the points. The whole algorithm is the interaction of two tests: who is crowded (`core`), and — in the depth-first `while` — only cores extend the chain. Compare the roles assigned in Figure 3.13 and the counts in the worked example above.

Failure Modes

DBSCAN. Its signature failure is *varying density*: one global ε cannot describe a dataset whose clusters live at different densities — the ε that separates a sparse pair of clusters shreds a dense cluster into noise. Chapter 2 works this out by hand; the slide-level comparison (slides 19–20) adds the practical contrast: DBSCAN needs no K and accepts arbitrary distances (it only needs the neighborhood relation), tolerates and *reports* outliers, but its affinity matrix grows with N^2. Slide 21's memory workarounds are the standard trio: store the sparse neighborhood as arrays instead of a dense matrix; switch to OPTICS (Ankerst et al., 1999), which computes a density-robust *ordering* (reachability) rather than a single clustering, at higher compute cost; or shrink the problem first — reduce dimension (PCA in feature space) or subsample/pre-cluster to exemplars and cluster those. The sklearn-style moons-and-circles comparison of slide 20 is the geometric summary: k-means fails exactly where DBSCAN succeeds, and vice versa.

Slide 19's comparison, as a table.

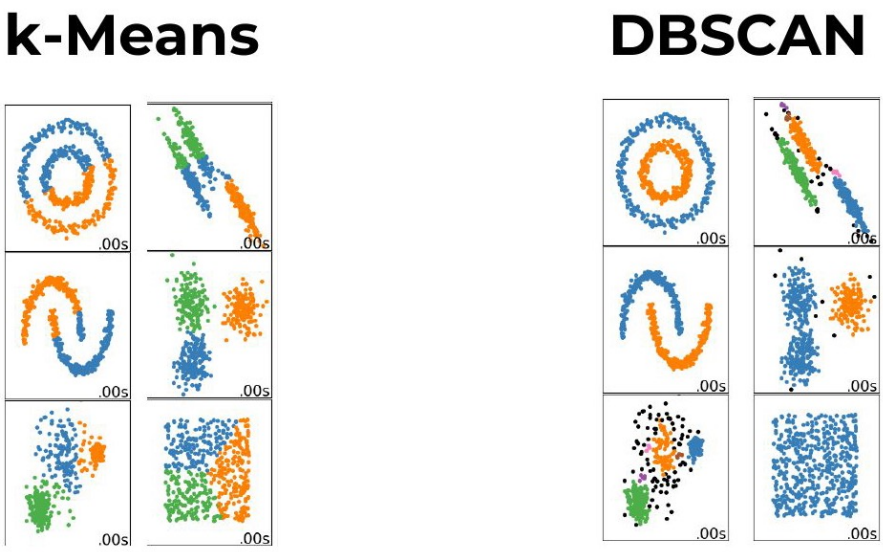


Figure 3.15. The same six datasets clustered by k-means (left) and by DBSCAN (right), exactly as the lecture lines them up (Lecture 02, slide 20). Row by row: concentric circles and slanted elongated blobs on top; two moons and three round blobs in the middle; blobs of uneven spread (with strays) and one featureless square at the bottom. k-means draws straight halfway-between-centroids boundaries everywhere — bisecting the outer circle, both moons, and the slanted stripe — because its cells are Voronoi regions, no matter the data. DBSCAN follows density instead: it threads each moon and separates ring from hub; on the uneven blobs it marks the strays as noise (the black points — the verdict k-means does not have); and on the structureless square it quietly returns one uniform bucket, the respectably boring answer a centroid method cannot give — k-means cuts even noise in half.

Source: Lecture 02, slide 20 (panels from the scikit-learn cluster comparison gallery).

	k-means	**DBSCAN**
number of clusters	must be given (K)	found automatically from density
what it needs from the data	a well-defined mean $\boldsymbol{\mu}_i = \frac{1}{\lvert X_i \rvert} \sum_{\boldsymbol{x} \in X_i} \boldsymbol{x}$ (so vectors in a space where averaging makes sense)	only a neighbourhood relation N — any distance works, even one without a mean (edit distance, graph distance)
scaling	runs on mini-batches, memory $O(Kd)$	memory grows with n^2 (the matrix N)
sensitive to	the choice of K (and the seeds)	the choice of ε (and `min_samples`)
outliers	skew the results: every point must join a cluster and pulls its mean	are ignored: reported as noise
cluster shape	convex, roughly round, similar size	anything connected and dense

The last row is not on the slide but is the lesson of slide 20 (Figure 3.15).

That summary deserves to be seen, not just quoted: Figure 3.15 is the slide itself.

Key Concept — Distance metric + clustering beats chance

A suitable distance metric plus clustering of the data points can yield an unsupervised classification that is better than chance — and anything better than chance can be exploited as a learning signal (Lecture 02, slide 23). This is the conceptual hinge of the chapter: it converts the classical toolbox of Sections 3.1–3.2 from an *evaluation* apparatus into a *training* apparatus. The next two sections are two different cashings-out of the same sentence: DeepCluster exploits *cluster assignments* that are better than chance; STEGO exploits *feature correspondences* that are better than chance.

Key takeaway — There is no best clustering algorithm, only a best fit

k-means asks you for the number of groups and then draws straight borders between round blobs of roughly equal size. DBSCAN asks you for a radius instead, follows density wherever it goes, and is allowed to answer "this point belongs to nothing." Each succeeds exactly where the other fails, which is why slide 20's moons-and-circles picture is worth remembering. For the rest of the chapter the important consequence is narrower: whichever routine you close into a training loop, *its assumptions become the network's assumptions*. Cluster with k-means and you have quietly told the network that categories are round blobs.

3.3 DeepCluster: Closing the Loop

3.3.1 DeepCluster: cluster your own features, then train on the cluster numbers

What is it? DeepCluster is a way to train an image network with no labels at all. Once per epoch it runs the whole dataset through the network, groups the resulting feature vectors with k-means, and writes down which group each image landed in. Those group numbers are then treated exactly like class labels: the network is trained with ordinary cross-entropy to predict them. Next epoch, the features have changed, so the grouping is redone from scratch and the process repeats.

Why was it introduced? The unsupervised methods before it worked by inventing a puzzle for the network to solve — put the shuffled tiles of an image back in order, colour in a grey photograph, predict where one patch sits relative to another. Each puzzle works, and each one bakes in exactly one designer's guess about what matters. Colourisation teaches a network that colour is important; jigsaw teaches it that spatial layout is important; neither teaches it what a *category* is. DeepCluster's ambition was to stop guessing: let the data say which images belong together, and train on that. Its price, as we will see, is that a loop with no external correction can also confidently learn the wrong thing.

Model at a glance: DeepCluster	Caron, Bojanowski, Joulin & Douze, 2018
In one line	Cluster the network's own features every epoch, call the cluster indices labels, and train on them with plain cross-entropy.
Problem it solves	Learning general visual features from unlabelled images without hand-designing a pretext puzzle that fixes in advance which invariance matters.
Key intuition	A clustering of deep features is better than chance even when the network is untrained — and anything better than chance is a usable training target (slide 23).
Inputs	Unlabelled images. The reference implementation first passes them through a fixed Sobel edge filter, which throws colour away so the clustering cannot latch onto it.
Architecture	A convnet $f_{\boldsymbol{\theta}}$ (AlexNet or VGG-16) plus a linear softmax head g_W over K pseudo-classes. The head is thrown away at the end; the convnet is the product.
Loss	Nothing new: softmax cross-entropy against the pseudo-label $c(\boldsymbol{x})$, Equation (3.4). All the novelty is in where $c(\boldsymbol{x})$ comes from.
Training	Alternate, once per epoch: (i) offline — extract features for every image, reduce to 256 dimensions with PCA, whiten, ℓ_2-normalise, run k-means with $K = 10{,}000$ on ImageNet; (ii) online — ordinary minibatch SGD against those frozen pseudo-labels, with images sampled uniformly *over clusters* rather than over images.
What you keep	The convnet's convolutional stages. The classification head is meaningless outside the epoch that produced it, because cluster index 17 means something different every epoch.
Strengths	No pretext engineering; uses a completely standard supervised training loop; best linear-probe numbers of its era (Table 3.3); transfers to detection and segmentation.
Main failure modes	Its own mistakes are re-taught as truth (*error reinforcement*); k-means can empty clusters or pile the dataset into one; the clustering can lock onto a trivial cue. All three are drawn in Figure 3.17.
Leads to	SwAV (Caron et al., 2020), which makes the assignment online and balanced (Chapter 10); and, at billion-image scale, balanced hierarchical k-means as a *data curation* tool inside DINOv3 (Vo et al., 2024) (Chapter 11).

DeepCluster's loop (slides 22–25; Caron et al., 2018). Every epoch runs five steps:

1. **Extract** features $f(\boldsymbol{x})$ for every image with the current network.
2. **Compress** them with PCA.
3. **Cluster** them with k-means into K groups.

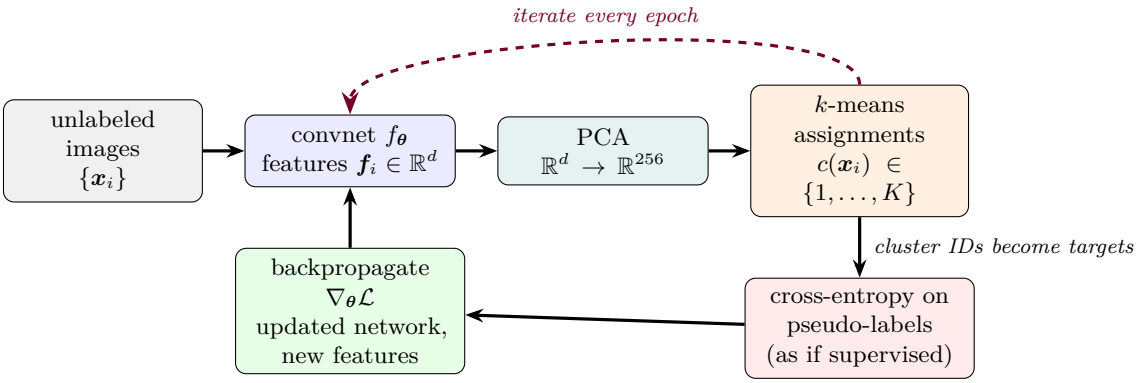


Figure 3.16. The DeepCluster loop (Lecture 02, slides 22–24). Features are extracted, PCA-reduced, and clustered; the cluster indices serve as pseudo-labels for cross-entropy training; the updated network yields better features, which are re-clustered. The classification head is discarded at evaluation — the learned *features* are the product.

4. **Pretend the cluster numbers are labels.** Give each image $\boldsymbol{x}$ its cluster number $c(\boldsymbol{x})$, and train a classifier head W to predict it with ordinary cross-entropy:

$$\mathcal{L} = -\sum_{\boldsymbol{x}} \log \operatorname{softmax}(W f(\boldsymbol{x}))_{c(\boldsymbol{x})}.$$

5. **Backpropagate** into the network, which changes the features — then go back to step 1.

Figure 3.16 draws the loop.

3.3.2 A round trip through one epoch

Three epochs, watched closely. **Epoch 1.** The network has random filters that act as crude colour and edge detectors. Every image is pushed through it, giving a vector of 4096 numbers per image. Those are first compressed (in 4096 dimensions k-means would suffer from the concentration of distances, Section 3.1.2), then clustered. The clusters are visibly wrong: a wolf and a poodle end up together because both stand on green grass. But they are better than chance — and that is enough.

Epoch 2. The network is trained to predict each image's cluster. To do that it must develop filters for whatever *really* separates the clusters: textures, line junctions, curves.

Epoch 3. The features are recomputed with those better filters and clustered again. The new clusters are cleaner, so the network gets a harder, better target.

No one outside the system ever corrects it. Yet the targets keep improving, because each round's targets come from features that the previous round made less naive.

Why doesn't this just go round in circles? At the start the network is random — but a random convolutional network is *not* a useless feature extractor. Its structure alone (local filters, weight sharing) already puts similar-looking images near each other.

The paper puts a number on it (slide 23): a small classifier trained on the last convolutional layer of a *random* AlexNet reaches 12% on ImageNet, where guessing gives 0.1% — more than a hundred times better than chance, from architecture alone.

So the very first k-means grouping is already slightly meaningful. The network is then trained to predict that grouping, which strengthens whatever real structure was in it, and each round builds on the last.

In plain words — Isn't training a network on its own opinion circular?

This is the objection everyone raises, and it deserves a real answer rather than a shrug. The answer is that "its own opinion" is not one opinion. Two different machines take turns, and they are blind in different directions.

k-means sees only *global geometry*. It is handed a cloud of N points and knows nothing whatsoever about images: not what a dog is, not which two pictures were taken in the same park, not even which point came from which file. All it can do is notice that the cloud has dense regions, and cut it into them. One consequence matters here: a mistake that affects *one* image — a bad crop, an odd exposure, a feature vector knocked sideways by noise — is invisible to it, because one point cannot move a centroid that is the average of thousands.

The network sees only *individual images*. Each gradient step looks at one batch, with no idea that a global partition exists. But it is the only half of the system that can look at a picture and ask "what is actually *in* here?" — and because it must give a single consistent answer for thousands of images at once, it cannot afford to honour a grouping that only one weird image supports. Idiosyncratic noise in the partition costs it loss, so the cheapest thing it can do is ignore that noise. Its features come out smoother than the partition that produced them.

So each half filters exactly the kind of error the other cannot detect: k-means removes per-image noise by averaging, and the network removes per-cluster noise by having to generalise across images. Feeding the cleaned output back in is therefore not a circle but a ratchet — *provided* the first grouping was better than chance. If it were pure noise, both halves would faithfully amplify pure noise, and the ratchet would turn the wrong way. That is not a hypothetical: it is failure mode (i) below.

Everyday analogy — A study group with no answer key

Five students revise for an exam and nobody has the answers. They mark each other's work anyway. Any one student's marking is unreliable — but they are unreliable in *different* places, so the group's majority verdict is noticeably better than any individual's, and studying against that verdict makes each of them better, which makes the next

verdict better still. That ratchet is DeepCluster: k-means is the group verdict and the network is the student revising against it.

Where the analogy breaks. Real students can check a textbook when something smells wrong. DeepCluster cannot; nothing outside the loop ever says "no." So when the whole group shares one misconception — everybody sorting by background colour instead of by animal — revising against the verdict does not correct it, it drills it in. A study group converges on being confidently wrong together, and so does this loop. The anti-collapse machinery below exists because there is no textbook.

Worked Example — Digits, loops, and junctions

Suppose you must learn to tell handwritten digits 0–9 apart without ever seeing a label. *Epoch 1.* The random convnet's translation-invariant filters already respond to crude visual traits, so the first clustering groups digits by silhouette statistics rather than identity: the loopy 0s and 8s fall near each other; the mostly-vertical 1s and 7s share a cell. *The update.* Those rough cells are now declared classes, and the network is trained to predict them with high confidence. Confidence is hard to buy with silhouette alone, so the optimizer is pushed to find more discriminative evidence — line junctions, curvature, closed loops. *Epoch 2.* Re-clustering the improved features exploits exactly that new evidence: the 1s and 7s, which the first partition fused, now separate, and the epoch-3 target list reflects it. The same dynamics amplify error just as happily as signal: had the first clustering ruled "curved strokes irrelevant," cross-entropy training would have taught the network to *ignore* curvature, freezing the mistake into the representation (the mechanism behind the failure modes below).

3.3.3 Twins of EM: the alternation, written down

The loop of slide 24 is an alternating minimization, and writing both halves makes its levers visible. Let $f_{\boldsymbol{\theta}}$ be the convnet, g_W a linear classification head, $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$ centroids, and $c(\boldsymbol{x}_i) \in \{1, \ldots, K\}$ the assignment of image $\boldsymbol{x}_i$. Each epoch first freezes the network and re-solves the clustering problem,

$$\min_{c(\cdot),\, \{\boldsymbol{\mu}_j\}} \sum_{i=1}^{N} \left\| f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - \boldsymbol{\mu}_{c(\boldsymbol{x}_i)} \right\|_2^2, \tag{3.3}$$

which (after the PCA step discussed below) is exactly the k-means objective of Section 3.2.2 — one E-step over the whole dataset, assigning every image a pseudo-label $c(\boldsymbol{x}_i)$. It then freezes the assignments and descends a standard supervised cross-entropy against them,

$$\min_{\boldsymbol{\theta}, W} \sum_{i=1}^{N} \mathcal{L}_{\mathrm{CE}}(g_W(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)),\, c(\boldsymbol{x}_i)), \tag{3.4}$$

Table 3.3. DeepCluster's results table as shown on slide 25 (Caron et al., 2018): a linear classifier is trained on the frozen activations of each convolutional layer of an AlexNet, on ImageNet and on Places; top-1 accuracy in %, averaged over 10 crops; numbers for the other methods from Zhang et al. The top block gives reference points: AlexNet trained *with* Places or ImageNet labels, and an untrained (random) AlexNet. The middle block lists earlier self-supervised methods (context encoders, context prediction, colourisation, BiGAN, jigsaw, counting, split-brain). Bold: best unsupervised result per column.

	ImageNet					Places				
Method	c1	c2	c3	c4	c5	c1	c2	c3	c4	c5
Places labels	–	–	–	–	–	22.1	35.1	40.2	43.3	44.6
ImageNet labels	19.3	36.3	44.2	48.3	50.5	22.7	34.8	38.4	39.4	38.7
Random	11.6	17.1	16.9	16.3	14.1	15.7	20.3	19.8	19.1	17.5
Pathak et al. (inpainting)	14.1	20.7	21.0	19.8	15.5	18.2	23.2	23.4	21.9	18.4
Doersch et al. (context)	16.2	23.3	30.2	31.7	29.6	19.7	26.7	31.9	32.7	30.9
Zhang et al. (colour)	12.5	24.5	30.4	31.5	30.3	16.0	25.7	29.6	30.3	29.7
Donahue et al. (BiGAN)	17.7	24.5	31.0	29.9	28.0	21.4	26.2	27.1	26.1	24.0
Noroozi & Favaro (jigsaw)	**18.2**	28.8	34.0	33.9	27.1	23.0	32.1	35.5	34.8	31.3
Noroozi et al. (counting)	18.0	30.6	34.3	32.5	25.7	**23.3**	**33.9**	36.3	34.7	29.6
Zhang et al. (split-brain)	17.7	29.3	35.4	35.2	32.8	21.3	30.7	34.0	34.1	32.5
DeepCluster	13.4	**32.3**	**41.0**	**39.6**	**38.2**	19.6	33.2	**39.2**	**39.8**	**34.7**

an ordinary training step for the network.

The two halves are very different.

- **Clustering** (Eq. (3.3)) is done *once per epoch*, over every image, and gives each image one fixed target.
- **Training** (Eq. (3.4)) is done in small random minibatches, like any supervised model.

Two consequences follow:

- **The targets move.** "Cluster 17" in one epoch has nothing to do with "cluster 17" in the next — the numbering is arbitrary — so the classifier head effectively faces a new problem every epoch.
- **k-means' assumptions become the whole method's weaknesses.** It assumes round clusters of similar size (Chapter 2), and wherever that is wrong, DeepCluster inherits the error.

Slide 25 shows the paper's main results table; Table 3.3 transcribes it in full.

Three things about this table recur in every evaluation in the book.

1. What is being measured. The features are frozen and only a linear classifier is trained on top. So the number tells you how *linearly separable* the classes already are in the learned space.

2. The best layer is in the middle. Accuracy peaks at conv3 (41.0) and *drops* by conv5 (38.2). The last layers are trained hardest on the pretext task — here, predicting arbitrary k-means clusters — and specialise to it. Middle layers hold the same meaningful structure in a less task-specific form. This pattern appears again and again in self-supervised learning, and the paper's filter pictures agree: conv3 units respond to object parts, conv5 units to whatever currently separates the pseudo-labels.

3. How close to supervised. On ImageNet conv5, DeepCluster (38.2) closes about two thirds of the gap between a random network (14.1) and full supervision (50.5). On Places — images DeepCluster never trained on — its conv4 (39.8) even *beats* ImageNet-supervised features (39.4), a first sign that self-supervised features transfer at least as well as supervised ones. Its weak conv1 (13.4) is most likely the Sobel filter's doing: the first layer never sees colour.

Two details of DeepCluster are easy to miss and essential.

1. Preprocess before clustering. The 4096-number AlexNet features are not clustered directly. They are first

- **reduced to** 256 **numbers with PCA** — in 4096 dimensions distances all look alike (Section 3.1.2), and noise dimensions would decide the clusters;
- **whitened** — many feature channels are correlated copies of each other, and whitening stops them from counting twice in the distance;
- **normalised to length** 1 (Caron et al., 2018).

Skip this, and the pseudo-labels become noise and poison the training half of the loop. (Section 3.2.5 used PCA for a different reason — to save memory. Here it is fixing the geometry.)

2. Guards against collapse. Important enough for a box of its own, below.

Failure Modes

DeepCluster (error reinforcement and collapse mechanics). Two distinct pathologies shadow the loop, and neither is optional to understand. (i) *Self-referential error reinforcement*: because the targets are self-generated, any systematic mistake of the clustering step gets treated as ground truth during backpropagation. If the random features mingle grass and road, or group wolves with poodles by background, the network is next trained to predict exactly that confusion with confidence, and the mistake is literally optimized in, epoch over epoch, and tightens. The signature in the results table is the stalled conv5 column: features that should escalate toward object parts saturate on texture slots instead. (ii) *Empty-cluster dynamics and trivial solutions*: nothing in the joint objective (3.3)–(3.4) forbids k-means from leaving clusters empty or piling most images into a few cells, after which cross-entropy trains a near-constant converter — a

self-confirming collapse. The paper's concrete defense is *empty-cluster reassignment*: when a cluster turns up empty, a populated centroid is copied with a small random perturbation to seed it, and the populated cluster's points are re-split between the two. Combined with cluster-uniform resampling at training time and the paper's choice of a *large overcomplete* K — the ImageNet run uses $K = 10{,}000$, ten times the class count — the reassignment gives each epoch enough spare slots that a lone big cluster can always be re-divided, and merged structure still carries signal. The chapter's companion lesson is a reading rule for the quiz-adjacent diagnostic: if embedding clustering mixes several categories in *every* cluster, blame the representation, not k-means; if a single cluster devours the dataset, blame the defenses instead. The long-run answer to both is SwAV's (Caron et al., 2020) online, balanced assignments — Chapter 10's territory.

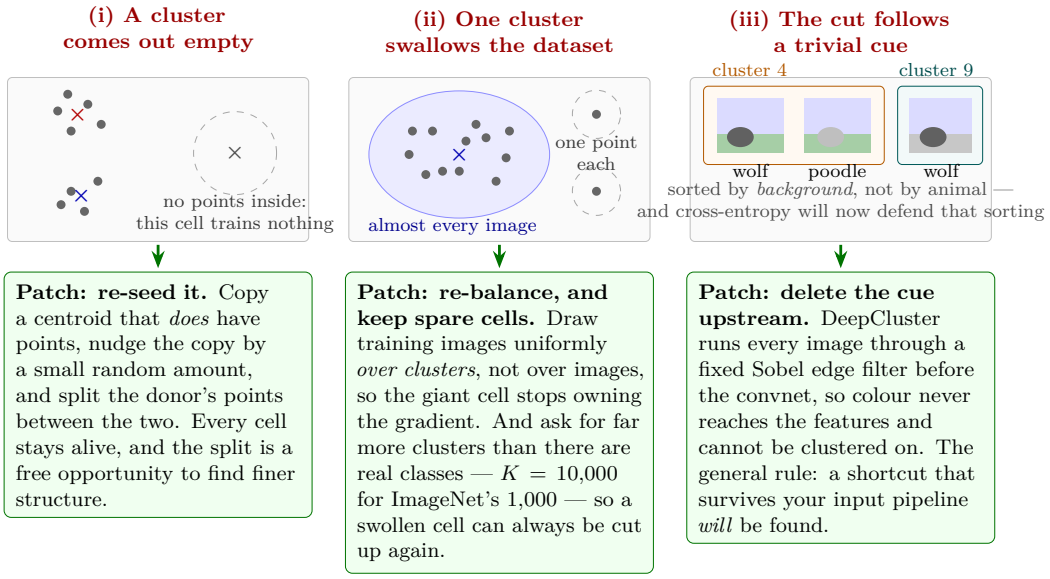


Figure 3.17. The three ways the DeepCluster loop degenerates, and the concrete patch for each (Caron et al., 2018). All three are consequences of the same structural fact: the loop has no external referee, so any degenerate partition that k-means produces is promptly defended by cross-entropy training. Note that the patches are not three variations of one idea — one repairs the clustering step, one reweights the training step, and one changes the *input* before either step runs.

How to read it: Read each column top to bottom: the red heading names the degeneracy, the grey panel draws it, and the green box is the fix that the paper actually ships. Then read across the bottom row — three fixes at three different points of the pipeline is the honest measure of how fragile the bare loop is.

Paper Analysis: Caron et al. 2018 — DeepCluster (Caron et al., 2018)

Problem. Learn visual features from unlabeled images, at scale, without pretext engineering. **Prior work.** Handcrafted pretext tasks — predicting context (Doersch et al., 2015), inpainting (Pathak et al., 2016), colorization (Zhang et al., 2016), jigsaw order (Noroozi and Favaro, 2016) — each bakes in one designer-chosen invariance. **Insight.** k-means on deep features produces assignments better than chance; better-than-chance is a learning signal. **Method.** Alternate *offline* k-means on PCA-reduced features with *online* cross-entropy training against the cluster indices (Figure 3.16). **Objective.** No new loss — standard softmax cross-entropy over K pseudo-classes; the novelty is the alternation and the sampling re-balance. **Experiments.** ImageNet linear probes at every AlexNet stage (Deng et al., 2009); transfer to Pascal VOC classification, detection, and segmentation (Everingham et al., 2010). **Results.** Conv5 linear probe 38.2% top-1 on AlexNet, peaking at conv3 (41.0%), and 48.4% with VGG-16 (Table 3.3) — best of its era. **Limitations.** An offline clustering pass every epoch; empty-cluster/trivial-solution instability; a fixed K. **Later work.** SwAV moves the assignment online and contrasts cluster assignments between augmented views (Caron et al., 2020); balanced hierarchical k-means returns at billion scale as a *data curation* tool inside the DINOv3 data engine (Vo et al., 2024) (Chapter 11). **What to remember.** Clustering is not only evaluation: in a loop with backpropagation it is a supervision source — the founding example of the affinity between classical methods and SSL.

The alternation is short enough to be instructive as code; the listing below compresses one epoch to its skeleton (offline pseudo-label fitting, then ordinary supervised training against the frozen targets), omitting the sampling re-balance, the empty-cluster bookkeeping, and the Sobel input transform of the reference implementation.

```
# ---- offline half: fit pseudo-labels on frozen features ----
features = np.stack([
    backbone(x) for x in unlabeled_loader       # (N, D), no gradients
])
pca = PCA(n_components=256, whiten=True)
zp = pca.fit_transform(features)                # 256-d, whitened
zp /= np.linalg.norm(zp, axis=1, keepdims=True) # L2-normalized
labels = KMeans(n_clusters=10_000).fit_predict(zp)  # c(x_i)

# ---- online half: supervise against the pseudo-labels ----
for x, idx in indexed_loader:
    loss = cross_entropy(head(backbone(x)), labels[idx])
    loss.backward(); optimizer.step()           # update theta, W
```

Listing 3.6. One DeepCluster epoch, distilled. The two halves correspond to (3.3) (lines up to the pseudo-label map) and (3.4) (the minibatch loop).

The separation of the two halves is what the next section throws away: everything above the comment line runs once per epoch, grows linearly with the dataset, and must be re-run from scratch when the features improve — while the loop below it is consumed in one gradient pass. Moving the first half inside the second is the program of the online-assignment methods of Chapter 10.

What changed — From hand-designed pretext tasks to DeepCluster

What stayed. The network, the optimiser, and the loss. DeepCluster trains an ordinary convnet with ordinary softmax cross-entropy and ordinary minibatch SGD; if you read only Equation (3.4) you could not tell it apart from supervised ImageNet training. The evaluation stayed too: freeze the features, fit a linear probe, report top-1.

What changed. *Where the target comes from.* A jigsaw or colourisation task has a target that a human designed once and that never moves: tile order, or the missing colour channel. DeepCluster's target is computed from the model itself and is recomputed every epoch. Supervision became a *fixed point* of the system rather than an input to it.

What it cost. Four things. (1) An extra full pass over the dataset every epoch — k-means on millions of vectors is not free, and it cannot be done in a minibatch. (2) Target instability: cluster 17 at epoch $t+1$ has nothing to do with cluster 17 at epoch t, so the classification head is re-learning a permuted problem each time and must be discarded at the end. (3) A new hyperparameter, K, that nobody can set from first principles. (4) A whole class of collapses that supervised training simply does not have (Figure 3.17), together with the three patches that keep them at bay.

Key takeaway — DeepCluster in one sentence

Run k-means on your own features, pretend the cluster numbers are labels, and train on them — the features improve, so the next clustering is better, and the loop ratchets upward. It works because the two halves are blind in different directions: k-means only sees the shape of the whole feature cloud, the network only sees individual images, and each therefore filters out noise the other cannot detect. It is fragile for the same reason: with nothing outside the loop to say "no," a systematic error is amplified just as efficiently as a signal, which is why the empty-cluster re-seed, the cluster-uniform sampling and the Sobel filter are load-bearing rather than cosmetic.

3.4 STEGO: Distilling Feature Correspondences into Segmentation

Slides 26–33 present the second hybrid, STEGO (Hamilton et al., 2022): labelling every pixel of an image into segments, with no labels at all, using *frozen* self-supervised features.

The starting observation (slide 27). A DINO backbone (Caron et al., 2021) (Chapter 11) already has *feature correspondence*: the same kind of part in different images — the wheels of different cars, grass in different scenes — gets similar features, although nobody told it what a wheel or grass is.

The question (slide 28): that correspondence is there but fuzzy. How do we sharpen it enough to cut images into clean segments? The slide's sketch shows the answer's shape: the same correspondences, before (left) and after (right) a small learned projection.

3.4.1 The whole idea first, as a story (almost no maths)

Start with yourself. Someone hands you a shoebox of a thousand holiday photos cut into small square pieces, and asks you to colour every piece by *what it shows* — sky, grass, person, bike — without a list of categories and without ever being told a single right answer. You would not stare at one piece in isolation. You would hold two pieces side by side and ask: "do these look like the same kind of stuff?" Sky next to sky: yes. Sky next to grass: no. After thousands of such comparisons, piles form by themselves, and each pile is a category you never had to name.

STEGO does exactly this, with one twist: the "eye" that judges "same kind of stuff?" is a *frozen* self-supervised network (DINO), which already has a decent but vague opinion. STEGO's job is to turn the vague opinion into a crisp one.

STEGO in four moments (Figure 3.18).

1. **Look.** The frozen DINO network cuts each photo into small patches (for example 8×8 pixels) and turns every patch into a feature vector. It is never trained again.
2. **Compare everything with everything.** For a pair of photos, measure how alike every patch of the first is to every patch of the second: a big table of similarities, called the *feature correspondence* F. Sky–sky pairs score high, sky–grass pairs lower. The pairs of photos are chosen three ways: a photo with *itself*, a photo with one of its *nearest neighbours* (a similar-looking photo found by retrieval), and a photo with a *random* other photo.
3. **Learn to say it decisively.** A small new network, the *segmentation head*, turns each DINO feature into a short vector (a few dozen numbers). Its own similarity table S is trained to *agree with F, but more firmly*: pairs that DINO finds clearly alike are pulled to "identical"; pairs DINO finds unlike are pushed to "unrelated". A threshold b decides which is which.
4. **Sort and tidy.** At test time, the short vectors of all patches are clustered with k-means — each cluster is a class — and a CRF snaps the cluster boundaries onto the edges in the photo.

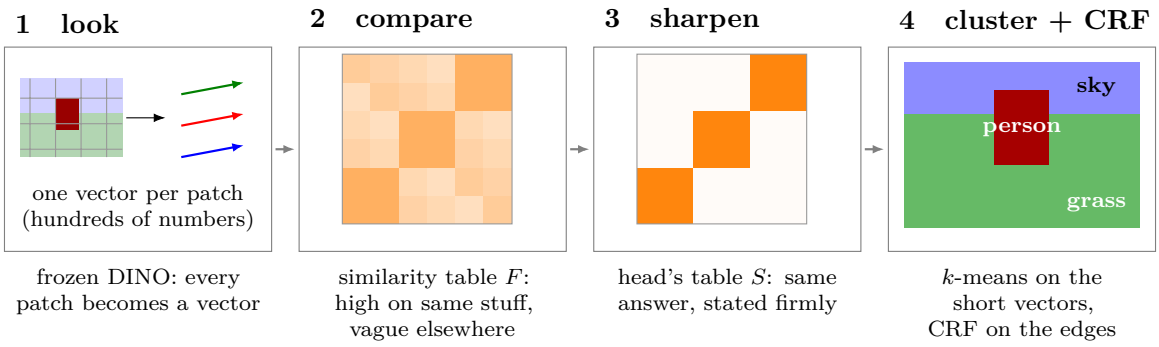


Figure 3.18. STEGO in four pictures. (1) The frozen DINO turns every patch into a vector. (2) Comparing every patch with every patch gives the table F (darker = more alike; rows and columns sorted as sky, sky, person, person, grass, grass): the diagonal blocks are darker, but everything is a little alike. (3) The trained head produces a table S with the same blocks, now black-and-white. (4) Clustering the head's vectors gives the segments; a CRF aligns them with the photo's edges.

How to read it: Compare the two tables: same three dark blocks, but in (2) the off-block cells are grey, in (3) they are white. That change is everything STEGO learns.

In plain words — the technical words, translated

patch, feature	a small square of the image; the vector the frozen network gives it
feature correspondence F	the table "how alike is patch a of photo 1 to patch b of photo 2", from the frozen network
segmentation head	the small network STEGO trains; turns each feature into a short vector
segmentation correspondence S	the same table, computed from the head's short vectors
bias b	the threshold between "pull together" and "push apart" (the "negative pressure")
KNN / self / random pairs	the three ways of choosing which two photos to compare
cluster probe	k-means on the head's vectors at test time; each cluster is a class
CRF	a classical smoother: neighbouring pixels of similar colour should get the same label

Everyday analogy — sorting the shoebox with a friend

A friend has glanced at the photo pieces before and has vague opinions ("these two are sort of alike"). You do not ask the friend to relearn anything; you ask for their opinion on every pair and then make a firm decision for each pair: above a threshold they go in

the same pile, below it they go in different piles. *Where it breaks:* if your friend cannot tell two things apart at all (DINO has no feature separating them), no amount of firm deciding will separate them — STEGO can only sharpen what the frozen network already sees.

3.4.2 STEGO: sharpen a correspondence that is already there

What is it? STEGO labels every pixel of an image with a semantic class, having never seen a single annotated pixel. It does not train a segmentation network in the usual sense. It takes a self-supervised backbone that is already good — and freezes it — then trains one small head whose only job is to *re-draw* the backbone's existing pattern of agreement between patches in a space small enough to cluster. At test time the head's outputs are clustered into segments and a classical smoother tidies the boundaries.

Why was it introduced? By 2022 it was well known that if you cut a DINO feature map into patches, patches showing the same kind of stuff land near each other — grass near grass, wheel near wheel — even across completely different photographs. That is exactly the "better than chance" signal of slide 23, but at pixel granularity instead of image granularity. The problem was that it is too blunt to use directly: cluster the raw backbone features and you get big, mushy blobs, and small objects such as street poles vanish entirely. Meanwhile the classical repair tool — a CRF — can snap a label map onto image edges, but it cannot invent a class that was not proposed to it. STEGO's contribution is to put a learned step between the two, one that sharpens the agreement pattern before anybody tries to cluster it.

Model at a glance: STEGO Hamilton, Zhang, Hariharan, Snavely & Freeman, 2022

In one line	Freeze a self-supervised backbone, learn a small projection that makes its patch-to-patch agreements crisper and lower-dimensional, then cluster and CRF-refine the result into segments.
Problem it solves	Semantic segmentation with no human annotation at any stage — not for the backbone, not for the head, not for the class list.
Key intuition	The *correspondence field* of a frozen self-supervised backbone is already semantic (slide 27). It needs distilling and sharpening, not replacing.
Inputs	An image, plus one partner image retrieved by nearest-neighbour search in the frozen backbone's own feature space — the ANN machinery of Section 3.1.3, reused to find semantically matched pairs.
The layer	One small *segmentation head* $\boldsymbol{\phi}$ mapping a backbone feature $\boldsymbol{v} \in \mathbb{R}^d$ to $\boldsymbol{\phi}(\boldsymbol{v}) \in \mathbb{R}^{d'}$ with $d' \ll d$. The DINO ViT (Caron et al., 2021) behind it is never updated — no gradients reach it.
Loss	The correspondence-distillation energy (3.9): $-\sum(F^{\mathrm{SC}}_{hwij} - b)\max(S_{hwij}, 0)$, where F is the frozen backbone's patch–patch cosine similarity and S the head's. Pairs the backbone already likes are pulled together; pairs below the bias b are pushed apart.

Training	Three correspondence terms summed per step — patches *within* one image, patches across its nearest-neighbour partner, and (with a small weight) patches across a random image, which supplies the negative pressure.
Inference	Project, cluster the projected features into segments (a *cluster probe*), then refine the label map with a fully connected CRF; the paper also reports a linear probe restricted to the discovered classes.
What you keep	The head ϕ and the cluster assignment. The backbone was never yours to change.
Strengths	No labels anywhere; large margins over prior label-free segmentation; recovers small objects (street poles) that clustering the frozen features directly misses.
Weaknesses	Inherits a spatial-smoothness bias from the CRF; cannot discover classes the backbone cannot see; the number of clusters and the bias b are unsupervised choices with sharp failure modes at both extremes.
Leads to	The general pattern of *probing* a frozen foundation model rather than retraining it — the default way DINO-family backbones are used in Chapters 11 and 14.

In plain words — What "feature correspondence" actually means

Take two photographs that have nothing to do with each other. Chop both into small square patches. Push every patch through a frozen self-supervised backbone, so each patch becomes a vector. Now measure the angle between every patch of the first image and every patch of the second.

The finding that STEGO is built on is that these angles are not arbitrary. A patch of sky in one photo points in almost the same direction as a patch of sky in the other; a patch of grass agrees with grass; a patch of a wheel agrees with a wheel. Nobody trained the backbone to do this — it fell out of self-supervision. That table of angles, one number per pair of patches, is the *correspondence field* F, and it is a semantic map of the world that cost nothing.

The trouble is that the agreement is real but weak. The angles say "these two are somewhat alike"; they do not say "these two are the same class and that third one is not." Feed them to a clustering routine and you get large mushy regions. So STEGO trains a small head to produce a second table of angles, S, and asks for S to be a *decisive* version of F: where F is above a threshold, drive S all the way towards agreement; where it is below, drive S all the way towards disagreement. Same opinions, louder — and in a space small enough that k-means can act on them. Figure 3.19 draws the before and after.

3.4.3 The recipe

STEGO's recipe (slide 29: "learn well-clustering features, then cluster at inference").

Training:

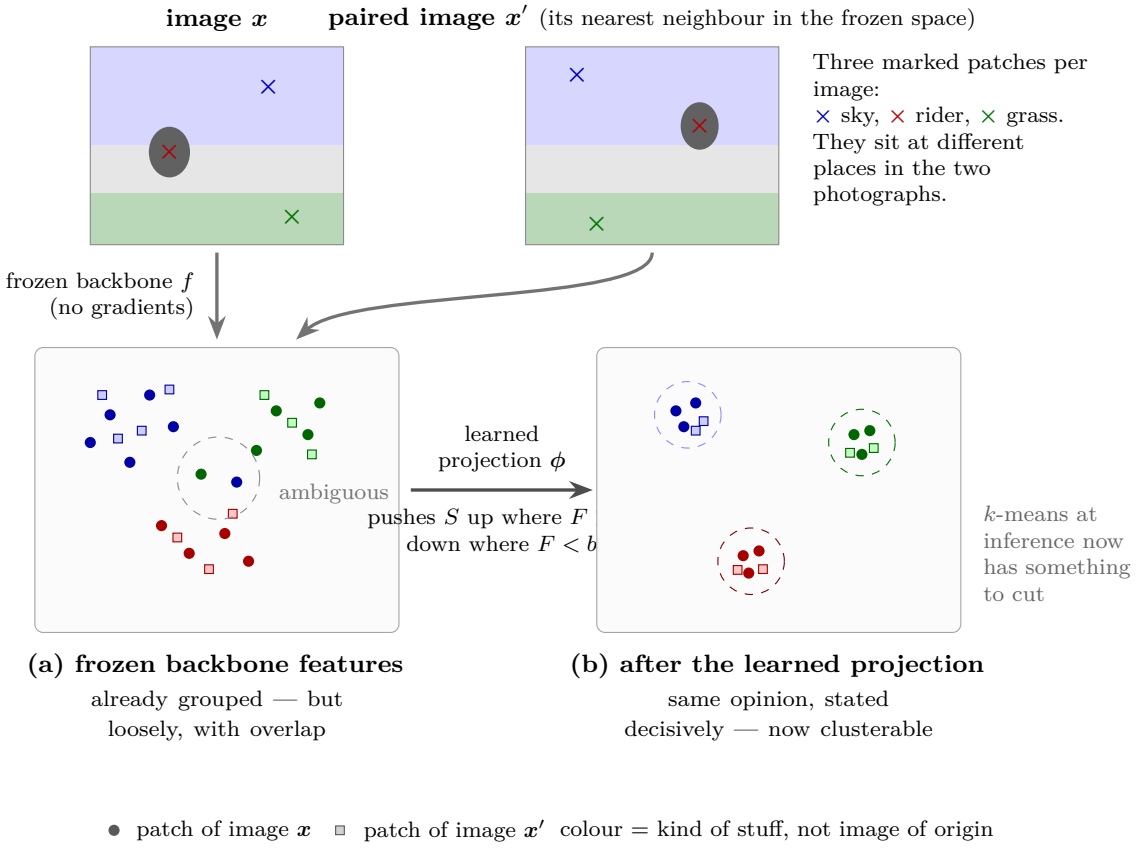


Figure 3.19. The observation STEGO starts from, and what the learned head does to it. Two unrelated photographs are chopped into patches and pushed through a *frozen* self-supervised backbone. In panel (a) the patches showing the same kind of stuff already sit near each other regardless of which photograph they came from — circles and squares of one colour mingle — which is the "better than chance" signal at pixel granularity. But the clouds are loose and touch, so clustering them directly yields mush. Panel (b) is the same information after the small learned projection ϕ: nothing new was added, the existing agreements were simply driven towards their extremes, and the result is separable.

How to read it: Compare the two panels on one question only — can you draw three circles that separate the colours? In (a) you cannot without cutting through the ambiguous middle; in (b) it is trivial. Note that the marker shape *never predicts the cluster: which photograph a patch came from is exactly the information the backbone has already discarded.*

Figure 3.20. The lecture's own version of the same observation (Lecture 02, slide 27). *Left:* one query image with three marked query patches — blue in the sky, red on the motorcyclist, green on the grass. *Middle and right:* two other photographs, with every patch tinted by how strongly it agrees, in the frozen backbone's feature space, with each query patch. Grass lights up green wherever grass appears; the riders light up red even at a different scale, pose and position; sky lights up blue. No labels and no training were involved in producing these maps — this is what a self-supervised backbone already believes, and it is the raw material STEGO sharpens.

Source: Lecture 02, slide 27 (figure from Hamilton et al. (2022)).

1. Pick pairs of images likely to show similar things. The partners are found by nearest-neighbour search in the frozen backbone's own feature space — the retrieval machinery of Section 3.1.3, reused.
2. Compute the frozen backbone features of both images.
3. Train a small projection into a lower-dimensional space that *strengthens* the similarities that were already high in the backbone.

Inference: cluster the projected features into segment labels (a "cluster probe", like k-means), and clean up the boundaries with a CRF. The paper also reports a linear probe for comparison. Figure 3.21 draws the pipeline; slide 33 recaps it after the CRF detour.

3.4.4 From the story to the formulas: why each piece of maths is needed

Figure 3.22 is the slide's full diagram. Each formula below answers one question from the story, in order, and each is checked on small numbers.

Question 1 (moment 2): "How alike are two patches?" Let the frozen backbone give image 1 a feature tensor f of shape $C \times H \times W$ (C channels, an $H \times W$ grid of patches) and image 2 a tensor g of shape $C \times I \times J$. Write $f_{hw} \in \mathbb{R}^C$ for the feature of patch (h, w). The slide's formula is the cosine similarity of every patch of image 1 with every patch of image 2:

$$F_{hwij} \;=\; \sum_c \frac{f_{chw}}{|f_{hw}|}\,\frac{g_{cij}}{|g_{ij}|} \;=\; \cos(f_{hw},\, g_{ij}) \;\in [-1, 1]. \tag{3.5}$$

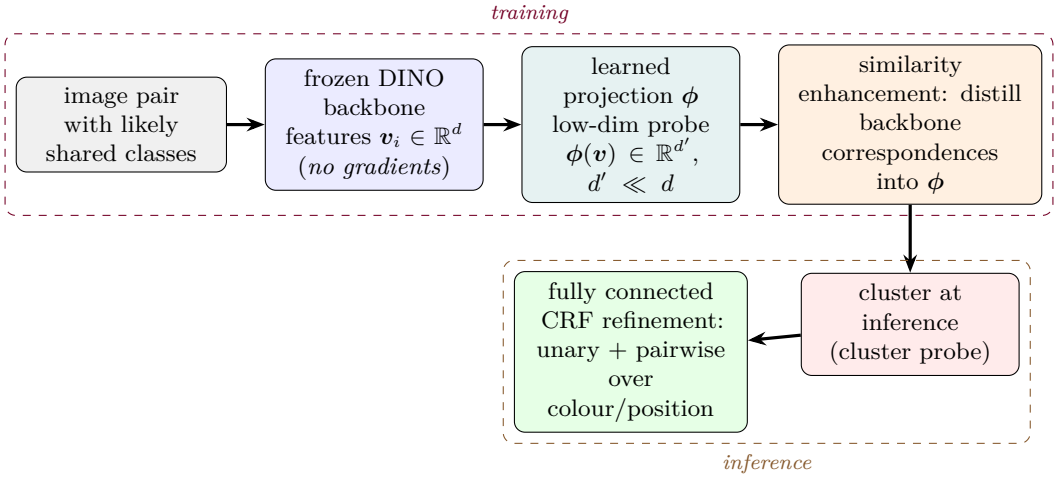


Figure 3.21. The STEGO pipeline (Lecture 02, slides 26–29, 33). A frozen DINO backbone's features are projected by a small learned head into a low-dimensional space; training pulls together exactly those projected feature pairs whose *backbone* similarity was already high — both within an image and across a paired image — sharpening raw correspondence into clusterable geometry. At inference, projected features are clustered and a CRF cleans up the label map.

Read it: divide both vectors by their length, multiply entry by entry, add up. F is a four-index table: for a 28×28 patch grid it has $28^4 \approx 615{,}000$ entries per image pair — which is why the slide draws it as a cube and calls its construction a *tensor product*. *Numbers:* $f_{hw} = (3, 4)$ and $g_{ij} = (4, 3)$ (two channels): lengths 5 and 5, so $F = (3 \cdot 4 + 4 \cdot 3)/25 = 0.96$ — very alike. With $g_{ij} = (4, -3)$ instead: $F = (12 - 12)/25 = 0$ — unrelated. *Why cosine:* it ignores the vectors' lengths (how "strongly" a patch fires) and keeps only their direction (what it shows) — the same reason DINO features are compared by cosine (slide 5).

Question 2 (moment 3): "What does the head produce?" The segmentation head $\mathcal{S}$ maps every backbone feature f_{hw} to a short vector $s_{hw} \in \mathbb{R}^K$ with $K \ll C$ (tens instead of hundreds of numbers); likewise t_{ij} for image 2. Its correspondence table is built the same way:

$$S_{hwij} \;=\; \cos(s_{hw},\, t_{ij}) \;\in [-1, 1]. \tag{3.6}$$

Why a new, smaller space: the backbone's space must describe everything about a patch (texture, position, part, object); the head's space only has to say "which segment". Low dimension forces it to keep only that, and makes k-means at test time fast and reliable.

Question 3 (moment 3): "What should S be?" The loss on the slide:

$$\mathcal{L}_{\text{simple}}(x, y, b) \;=\; -\sum_{h,w,i,j} (F_{hwij} - b)\, S_{hwij}. \tag{3.7}$$

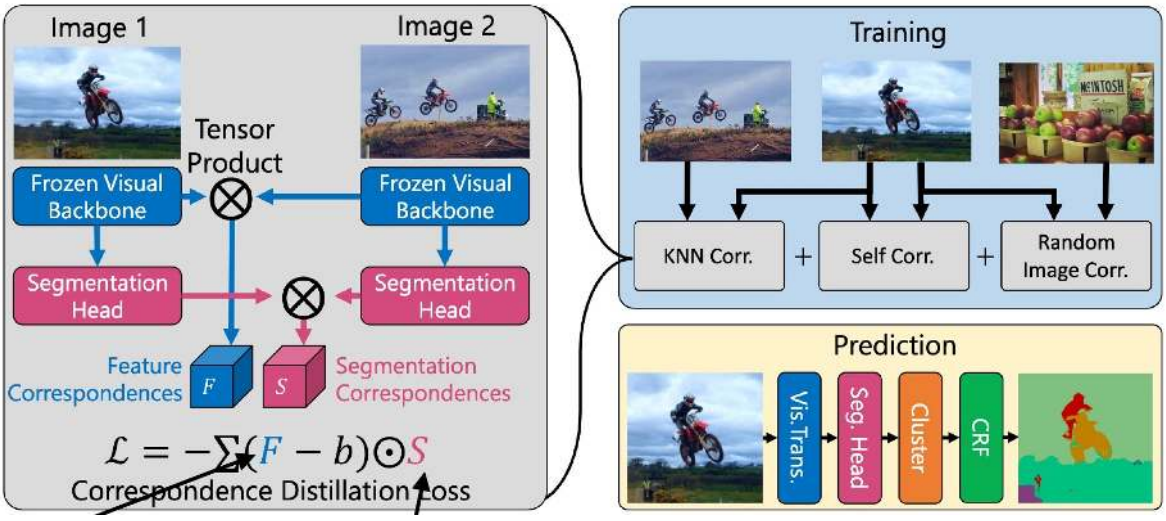


Figure 3.22. STEGO as shown in the lecture. *Left:* two images go through the same frozen backbone; the tensor product ($\otimes$) of their features gives the feature correspondences F (blue cube). The same two images also go through the trainable segmentation head; their tensor product gives the segmentation correspondences S (pink cube). The loss $\mathcal{L} = -\sum(F - b) \odot S$ compares the two. *Top right:* during training the loss is summed over three kinds of image pairs — a nearest neighbour (KNN), the image itself (self), and a random image. *Bottom right:* at prediction time the image goes through the vision transformer, the segmentation head, clustering and a CRF.

Source: Lecture 02, slide 28 (figure from Hamilton et al. (2022)).

Only S can change (the backbone is frozen, so F is a constant). The derivative with respect to one entry is $\partial\mathcal{L}/\partial S_{hwij} = -(F_{hwij} - b)$, so gradient descent moves each S_{hwij} in the direction of $F_{hwij} - b$ with a strength proportional to $|F_{hwij} - b|$:

- $F > b$: *attraction* — make the two patches' head vectors point the same way ($S \to +1$);
- $F < b$: *repulsion* — push them apart ($S \to -1$, or to 0 with the clamp of Question 5);
- $F \approx b$: almost no force.

The slide's highlighted sentence says the same: S is not trained to *equal* F but to reach total alignment or anti-alignment depending on the *sign* of $F - b$. Because the loss is linear in S and S is bounded, the optimum sits at the extremes — this is the "same opinion, said firmly" of the story (Figure 3.23).

Question 4: "Why is the bias b needed at all?" Suppose $b = 0$. DINO's cosine similarities are almost all positive (patches of natural images are rarely opposite), so every pair would be *attracted*. The cheapest way to satisfy every attraction is to map every patch to the *same* head vector: $S = 1$ everywhere, the loss is as low as it can be, and the segmentation has one segment. That is *collapse*, the same failure as in SimSiam or DeepCluster. Subtracting b turns the weakly-similar pairs into repelling pairs: the slide calls it "uniform negative pressure

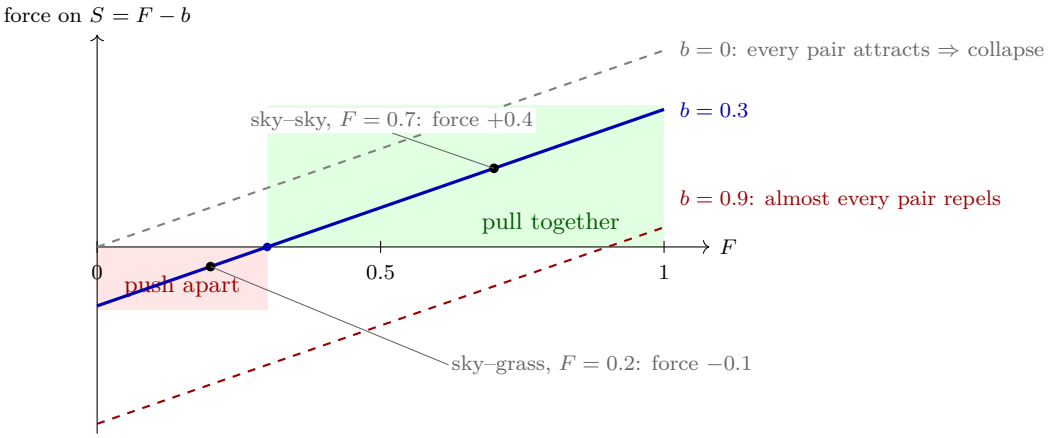
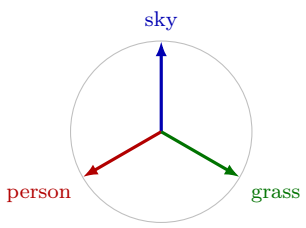


Figure 3.23. The force the loss puts on each pair of patches, as a function of the backbone similarity F of the pair (horizontal axis). With $b = 0.3$ (blue), a sky–sky pair ($F = 0.7$) is pulled together with force 0.4, a sky–grass pair ($F = 0.2$) pushed apart with force 0.1. The dashed lines show why b must be chosen with care.

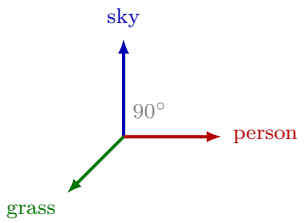
How to read it: Find where each line crosses zero: that crossing is b, the boundary between "same segment" and "different segment".

to prevent collapse". Too large a b fails the other way: nearly every pair repels, and each patch becomes its own island.

Question 5: "How far should unlike patches be pushed?" (the 0-clamp). In the paper's final loss the head similarity is clamped at zero: $\max(S_{hwij}, 0)$ replaces S_{hwij}. Once two head vectors are *perpendicular* ($S \leq 0$) the repulsion stops. The reason is geometric (Figure 3.24): you cannot make many vectors all *opposite* to each other — three vectors in a plane can at best be 120° apart ($\cos = -0.5$), and n vectors can at best reach an average cosine of $-1/(n-1)$ — but in K dimensions you can make K vectors all exactly perpendicular. With 27 classes and a head of a few dozen dimensions, "perpendicular" is achievable for every pair of classes; "opposite" is not, and chasing it destabilises training.



without the clamp: all three push to be opposite; the best a plane allows is 120° ($\cos = -0.5$), and each extra class makes it worse



with the clamp: pushing stops at 90° ($S = 0$); K dimensions hold K mutually perpendicular classes (drawn: 3 axes in 3D)

Figure 3.24. Why STEGO clamps the segmentation similarity at zero. "Unrelated" is expressed as *perpendicular*, which scales to many classes, not as *opposite*, which does not.

Question 6: "What about small objects?" (spatial centering). The paper's second refinement subtracts, for each patch (h, w) of image 1, its average similarity to all patches of image 2:

$$F^{\mathrm{SC}}_{hwij} \;=\; F_{hwij} - \frac{1}{IJ}\sum_{i',j'} F_{hwi'j'}. \tag{3.8}$$

Why: a patch on a thin pole in image 1 is compared with, say, 100 patches of image 2, of which only 2 show a pole ($F = 0.55$) and 98 show street and buildings ($F \approx 0.40$). With a threshold tuned for large regions, say $b = 0.30$, *all 100* pairs attract, and the 98 background pairs, each with force 0.10, outvote the 2 pole pairs (force 0.25 each): the pole is dragged into the background segment. After centering, the row's mean (0.403) is subtracted: the pole pairs become $+0.147$ and the background pairs about -0.003. With a small b applied after centering, only the pole pairs attract, and the pole keeps its own segment. Centering makes every patch judge its matches *relative to its own average*, so the few strong matches of a small object stand out. The paper reports that this helps small objects.

The full loss. Putting Questions 3–6 together, for one pair of images (x, y):

$$\mathcal{L}_{\mathrm{corr}}(x, y, b) \;=\; -\sum_{h,w,i,j} (F^{\mathrm{SC}}_{hwij} - b)\, \max(S_{hwij}, 0), \tag{3.9}$$

and training sums it over the three kinds of pairs of the slide's "Training" box:

$$\mathcal{L} \;=\; \lambda_{\mathrm{self}}\,\mathcal{L}_{\mathrm{corr}}(x, x, b_{\mathrm{self}}) + \lambda_{\mathrm{knn}}\,\mathcal{L}_{\mathrm{corr}}(x, x^{\mathrm{knn}}, b_{\mathrm{knn}}) + \lambda_{\mathrm{rand}}\,\mathcal{L}_{\mathrm{corr}}(x, x^{\mathrm{rand}}, b_{\mathrm{rand}}). \tag{3.10}$$

Question 7: "Why three kinds of pairs?"

pair	how it is chosen	what it teaches
self (x, x)	the image with itself	consistent labels inside one image: the grass patches of this photo belong together
KNN (x, x^{knn})	one of the image's nearest neighbours by the frozen backbone's global image feature (retrieval, §3.1)	the *same* segment across photos: this rider and that rider — the pull that makes classes consistent across the data set
random (x, x^{rand})	any other image	mostly unrelated content, so mostly *push*: the main source of negative pressure against collapse (its b is set to repel)

Each term has its own weight λ and its own threshold b. The KNN term is what the slide's recipe calls step 1, "sample images that (likely) contain similar classes" (slide 29): nearest neighbours in the frozen feature space tend to show the same kinds of things, so their cross-image pairs carry many true attractions.

Worked Example — STEGO's loss on three patches per image

Image x has patches a_1 (sky), a_2 (rider), a_3 (grass); its KNN partner y has b_1 (sky), b_2 (rider), b_3 (grass). The frozen backbone gives

F	b_1 sky	b_2 rider	b_3 grass
a_1 sky	**0.70**	0.25	0.30
a_2 rider	0.20	**0.60**	0.35
a_3 grass	0.30	0.35	**0.65**

All nine similarities are positive, and the matching pairs are only about twice as high as the others: DINO knows, but vaguely. Take $b = 0.4$ (no centering, to keep the numbers simple). The forces $F - b$ are $+0.30, +0.20, +0.25$ on the diagonal and between -0.05 and -0.20 off it: three attractions, six repulsions.

- *Start:* the untrained head gives, say, $S = 0.5$ everywhere. Loss $= -0.5 \times \sum(F - b) = -0.5 \times 0.10 = -0.05$.
- *After training:* $S = 1$ on the three matching pairs and $S \leq 0$ elsewhere (clamped to 0). Loss $= -(0.30 + 0.20 + 0.25) = -0.75$.
- *Collapse check:* $S = 1$ everywhere gives -0.10 — worse than the trained head. With $b = 0$ it would give $-\sum F = -3.70$, the best possible: collapse would win. That is Question 4 in numbers.

The trained head's table is the F table with its opinions pushed to the extremes: black diagonal, white elsewhere — panel 3 of Figure 3.18. k-means on the head's vectors then finds three clusters: sky, rider, grass.

```
def corr(a, b):                          # a: (C,H,W), b: (C,I,J) -> (H,W,I,J)
    a = normalize(a, dim=0); b = normalize(b, dim=0)
    return einsum("chw,cij->hwij", a, b)       # cosine of every patch pair

def loss_corr(x, y, b):
    f, g = backbone(x).detach(), backbone(y).detach()   # frozen DINO
    F = corr(f, g)
    F = F - F.mean(dim=(2, 3), keepdim=True)            # spatial centering
    S = corr(head(f), head(g))                          # the trainable part
    return -((F - b) * S.clamp(min=0)).mean()           # 0-clamp

x_knn = random_choice(knn_table[x])      # a nearest neighbour, precomputed
x_rand = random_image()
loss = (l_self * loss_corr(x, x, b_self)
        + l_knn * loss_corr(x, x_knn, b_knn)
        + l_rand * loss_corr(x, x_rand, b_rand))
loss.backward(); step(head)              # only the head is updated
```

Listing 3.7. STEGO's training step, following Equations (3.5)–(3.10).

Question 8: "How are segments read out, and how is it scored?"

- **Cluster probe** (the "Cluster" box of the prediction pipeline): k-means with cosine distance on the head's vectors of all patches of the data set, with k set to the number of classes of the benchmark (27 for COCO-Stuff's coarse classes). Each patch's cluster is its segment; upsample to pixels.
- **CRF** on the upsampled label map (next subsection).
- **Scoring**: the clusters have no names, so each cluster is matched one-to-one with a ground-truth class by the Hungarian algorithm (the matching with the largest total overlap), and accuracy and mean IoU are computed. Labels are used *only* for this matching and for scoring. A *linear probe* (a linear classifier on the head's vectors trained with labels) is reported as a second, supervised read-out.

The paper reports gains of +14 mIoU on COCO-Stuff and +9 mIoU on Cityscapes over the previous best unsupervised methods (Hamilton et al., 2022).

3.4.5 The CRF detour — and why STEGO's loss is one

Slides 30–31 detour through the Conditional Random Field (Lafferty et al., 2001), because classical segmentation pipelines use one as post-processing and because STEGO's loss will turn out to *be* a CRF energy. A dense CRF assigns pixel labels $\boldsymbol{y} = (y_1, \ldots, y_N)$ by minimizing

$$E(\boldsymbol{y}) = \sum_i \psi_u(y_i) + \sum_{i<j} \psi_p(y_i, y_j), \tag{3.11}$$

where $E(\boldsymbol{y})$ is the total cost of a labelling $\boldsymbol{y}$, and we want the labelling with the lowest cost. It has two kinds of term:

- the **unary** cost $\psi_u(y_i)$: how unhappy pixel i alone is with label y_i — usually the negative log of the network's predicted probability for that label;
- the **pairwise** cost $\psi_p(y_i, y_j)$: a penalty when two pixels that are close and similar in colour receive *different* labels.

The first says "trust the network, pixel by pixel"; the second says "neighbours that look alike should agree". The efficient version the slides cite (Krähenbühl and Koltun's fully connected CRF) links every pixel to every other, with the penalty fading with distance in position and colour, and uses a fast filtering trick to make that affordable. Concretely the pairwise potential is a compatibility-weighted sum of Gaussians,

$$\psi_p(y_i, y_j) \;=\; \mu(y_i, y_j) \left[w^{(1)} \exp\!\left(-\frac{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|_2^2}{2\theta_\alpha^2} - \frac{\|\boldsymbol{I}_i - \boldsymbol{I}_j\|_2^2}{2\theta_\beta^2} \right) + w^{(2)} \exp\!\left(-\frac{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|_2^2}{2\theta_\gamma^2} \right) \right], \tag{3.12}$$

where $\boldsymbol{p}_i$ is pixel i's position and $\boldsymbol{I}_i$ its colour.

Reading the pairwise term.

- $\mu(y_i, y_j) = \mathbf{1}[y_i \neq y_j]$: pay a penalty only when two pixels get *different* labels (the "Potts" rule). Which labels they are does not matter.
- **The first kernel** depends on both position and colour: nearby pixels *of similar colour* should agree. θ_α sets how far "nearby" reaches; θ_β sets how similar "similar colour" must be.
- **The second kernel** depends on position only: nearby pixels should agree, full stop. It removes speckle.

Why the colour part matters: make θ_β huge and colour is ignored, so the smoothing becomes blind to edges. The thin gaps between the slats of the bench on slide 31 — visible only as colour edges — would be smoothed away. Slide 31's three panels show the full story: the image, the noisy per-pixel guess, and the cleaned CRF result.

Slide 32 then makes the point this detour was for: *STEGO's loss optimises a very similar energy*, but over learned features rather than pixel labels,

$$E(\boldsymbol{\phi}) \;:=\; \sum_{\boldsymbol{v}_i, \boldsymbol{v}_j \in \mathcal{V}} w(\boldsymbol{v}_i, \boldsymbol{v}_j)\; \mu(\boldsymbol{\phi}(\boldsymbol{v}_i),\, \boldsymbol{\phi}(\boldsymbol{v}_j))\,, \tag{3.13}$$

where $\mathcal{V}$ is the set of all patch features (nodes of a graph); $\boldsymbol{\phi}$ is the learned feature mapping (the segmentation head); $w(\boldsymbol{v}_i, \boldsymbol{v}_j)$ is the *connectivity between features* — the slide's annotation: built from the backbone's cosine similarity, it is the data-dependent edge weight, like the colour-and-position kernels of a CRF; and $\mu(\boldsymbol{\phi}(\boldsymbol{v}_i), \boldsymbol{\phi}(\boldsymbol{v}_j))$ is the *connectivity between mapped features*, again from their cosine similarity — it plays the role of the CRF's label-compatibility cost.

Matching the two energies term by term. Put $w = F - b$ and $\mu = -S$ and Equation (3.13) is exactly the loss $-\sum(F - b)\, S$ of Equation (3.7):

	dense CRF (Equation (3.11))	**STEGO (Equation (3.13))**
nodes	pixels	patch features $\boldsymbol{v} \in \mathcal{V}$
what is chosen	a discrete label x_i per pixel	a continuous vector $\boldsymbol{\phi}(\boldsymbol{v}_i)$ per patch
edge weight	Gaussian of colour and position difference	$w = F - b$: backbone similarity minus threshold (may be negative)
pair cost	Potts: pay when labels differ	$\mu = -S$: pay when mapped vectors disagree where $w > 0$, agree where $w < 0$
unary term	$-\log p(x_i)$ from a classifier	none: there is no classifier; the backbone's opinion is already in w
how it is minimised	mean-field inference at test time	gradient descent on the head's weights at training time

Two differences are worth noticing. The CRF's edge weights are always positive (similar pixels should agree), whereas STEGO's $w = F - b$ can be negative: pairs below the threshold are

actively pushed apart. And the CRF optimises the labels of *one* image at test time, whereas STEGO's energy trains a network once, which then labels any image in one forward pass.

The link to CRFs, completed. In the energy of Equation (3.13), the frozen similarity F plays the role of the edge weight and the head's S plays the role of the pairwise cost. What happened to the CRF's *unary* term — the per-pixel belief about what each pixel is? It has no place in the loss, because the frozen backbone *is* that belief: it was already used to compute F. STEGO's loss is purely about *relationships between pixels*, which is why it never needs a fixed list of class names during training.

Research Frontier

The CRF-as-loss reading. Clustering features and then refining the label map with a CRF is the visible half of slide 33's composite figure; the deeper contribution is treating Eq. (3.11) not as inference to run at test time but as an *energy to differentiate through during training*. Equation (3.13) replaces CRF labels with feature embeddings, the edge weights with backbone cosine similarities, and mean-field inference with gradient descent on a probe. This blurs a boundary the field once kept sharp: graphical-model energies become self-supervised objectives. Open questions follow directly. Can the CRF's label space itself be learned end-to-end rather than fixed by post-hoc clustering? Does the energy view predict *which* frozen backbone distills best? And why does distilling *correspondence* sharpen semantics at all, rather than merely copying them — the paper's small-object finding suggests the probe discovers structure the energy was never explicitly told about?

Paper Analysis: Hamilton et al. 2022 — STEGO (Hamilton et al., 2022)

Paper. "Unsupervised Semantic Segmentation by Distilling Feature Correspondences" (ICLR 2022). **Problem.** Assign a semantic segment label to every pixel, with no human annotations at any stage. **Prior work.** Frozen foundation features cluster into objects, but too coarsely and too loosely for dense prediction; classical CRF refinement sharpens edges but cannot create semantics. **Insight.** The raw *correspondence field* of a self-supervised backbone is already semantic (slide 27); it needs distillation and sharpening, not replacement. **Method.** Freeze a DINO ViT (Caron et al., 2021); learn a low-dimensional projection whose pairwise cosine distances are pulled toward agreeing with the backbone's, within and across paired images (Figure 3.21, Eq. (3.13)); inference clusters the projected features and applies a CRF. **Objective.** The correspondence-distillation energy above; the backbone is never updated. **Experiments and results.** Evaluated on Cityscapes and COCO-Stuff with both linear and cluster probes; the headline finding is that the resulting segments exceed prior label-free methods by wide margins and discover small objects (e.g. street poles) that clustering the frozen backbone's features directly misses — the projection *sharpens* the features rather than merely copying them. **Limitations.** See the Failure Modes box; cluster count and class granularity remain unsupervised choices. **What to remember.** Segmentation without labels, from a frozen self-supervised backbone, by treating correspondence as the learning signal.

Failure Modes

STEGO. (i) *What the CRF fixes, and what it costs*: the side-by-side outputs of slide 33 show that without the CRF the label map is speckled even where the classes are right; the CRF snaps labels onto the colour edges actually in the image. Its own settings bring a new risk: the smoothness width (θ_γ) and colour width (θ_β) decide what counts as a "real edge", and wherever that is ambiguous the CRF smooths thin or low-contrast structures into their surroundings. (ii) *Contiguity bias*: spatial smoothness is built into the formulation; the method's characteristic qualitative failures are disconnected or elongated structures that a spatially smooth labeling assigns to the wrong side. (iii) *Backbone ceiling*: correspondence can only be distilled where the frozen features already carry it; classes invisible to the backbone do not emerge. (Likewise, if two classes are not separable by cosine similarity in the backbone space, no probe can separate them: the head can only sharpen structure the frozen features already have.)

What changed — From DeepCluster to STEGO

What stayed. The thesis. Both methods find structure that a classical routine can extract from features, and both turn it into a training signal because it is better than chance. Both also lean on the retrieval machinery of Section 3.1 — DeepCluster to cluster, STEGO to find each image a partner.

Figure 3.25. STEGO's output on ten COCO-Stuff scenes, and the CRF ablation that the failure-modes box above describes (Lecture 02, slide 33). *Row 1:* the input photographs. *Row 2:* the human ground-truth label map, shown only for comparison — no part of the method ever saw it. *Row 3:* STEGO's prediction with the CRF refinement. *Row 4:* the same prediction without the CRF. Rows 3 and 4 agree almost everywhere on *which* class a region is; they differ in how cleanly the region's boundary is drawn. The speckle in row 4 is what the CRF removes, and the cost of removing it is that thin and disconnected structures get smoothed towards their surroundings.

Source: Lecture 02, slide 33 (figure from Hamilton et al. (2022)).

What changed. Three things, and they are linked. (1) *Granularity*: DeepCluster's unit is a whole image and its signal is a cluster index; STEGO's unit is a patch and its signal is the agreement between *pairs* of patches. (2) *Who learns*: DeepCluster updates the backbone itself, so the features and the targets move together; STEGO freezes the backbone and trains only a small head on top, so the targets never move at all. (3) *What the classical tool is for*: in DeepCluster k-means *generates* the target; in STEGO the classical part (the CRF) is a *post-processor*, and the target comes from the frozen features.
What it cost. Freezing removes the ratchet. STEGO cannot become better than the correspondence field it was handed — a class the backbone cannot see will never be discovered, no matter how long you train the head. In exchange it also removes every collapse of Figure 3.17: with the backbone fixed there is nothing for a bad target to corrupt. STEGO's own instability lives elsewhere, in the single scalar b: at $b \to 0$ the head collapses to one direction, at $b \to 1$ it shatters into many.

3.4.6 Both hybrids, one thesis

One idea, two scales. DeepCluster and STEGO are the same idea.

	DeepCluster	STEGO
works on	whole images	individual pixels
weak signal it finds	clusters of features	matching features across images
classical tool it uses	k-means	CRF-style pairwise energy
inherited weakness	round clusters, must choose K	over-smoothing

Both point forward: refining pseudo-labels is Chapter 12's subject, and the DINO features STEGO relies on are trained in Chapter 11. The lecture's summary (slide 34) closes with the thesis of slide 23, now earned: *any structure a classical method can find, however weak, is a seed that backpropagation can grow.*

Key takeaway — STEGO in one sentence

A frozen self-supervised backbone already agrees with itself about which patches show the same kind of stuff, across completely unrelated images; a small learned head turns that weak agreement into a decisive one in a space small enough to cluster; a CRF then snaps the resulting label map onto the edges that are actually in the picture. The whole method is a lesson about what to do with a good frozen model — do not retrain it, read it more carefully.

3.5 Chapter summary

The chapter had one thesis and three layers of machinery under it.

The thesis is slide 23's sentence, and it is worth restating without any jargon: if some routine sorts your unlabelled data better than a coin flip would, you can train on its output, and the training will improve the features that the routine was sorting. Do it again and the sorting improves. Nothing external is ever consulted. The two hybrids of this chapter are that sentence applied at two granularities — whole images for DeepCluster, individual patches for STEGO — and their differences all follow from that one choice.

Underneath sit the classical tools. *Search* came first (Section 3.1), because you cannot group what you cannot compare: KD-trees give exact neighbours by cutting space with axis-aligned planes and pruning any subtree that is farther away than the best candidate so far; that pruning test stops firing above roughly twenty dimensions, so everything at deep-feature scale runs on approximate indexes instead — hashing, proximity graphs, or compressed codebooks. *Clustering* came second (Section 3.2): k-means wants a number of groups and draws round cells; DBSCAN wants a radius, follows density, and is allowed to call a point noise. Neither dominates, and whichever you use inside a training loop quietly donates its assumptions to the network.

Then the two loops. **DeepCluster** (Section 3.3) clusters the features once per epoch and trains the network to predict the clusters.

- *Why it works:* a random convnet is already better than chance (a small classifier on random AlexNet features scores 12%, chance is 0.1%), and the clustering and the network make different kinds of mistakes, so each corrects the other.
- *Why it is fragile:* nothing outside the loop can correct it, so it needs explicit guards — re-seeding empty clusters, sampling evenly across clusters, and a Sobel filter (Figure 3.17).

STEGO (Section 3.4) does the opposite: freeze a backbone that is already good, and train only a small head that sharpens its correspondences into something clusterable. More stable — but it can never be better than the backbone it froze.

Table 3.4 collects the vocabulary the chapter introduced.

Table 3.4. The chapter's working vocabulary, in the order the ideas appeared.

Term	What it means, in plain words
Descriptor	A vector that stands in for an image or a patch. Comparing images becomes comparing descriptors.
Exact vs. approximate search	Exact search always returns the true nearest neighbour. Approximate search sometimes returns *nothing* where a true neighbour existed — it does not degrade gently.
Curse of dimensionality	Above roughly twenty dimensions all pairwise distances become similar, so pruning tests stop firing and tree search degenerates into a full scan.
Core / border / noise point	DBSCAN's three verdicts: enough neighbours within the radius; not enough but touching someone who has; neither.
k-means++ seeding	Choosing the starting centroids one at a time, each with probability proportional to its squared distance from the nearest seed already chosen. A little effort up front so the seeds land in different dense regions; Lloyd's loop itself is unchanged.
Mini-batch k-means	The streaming version of Lloyd's update: a centroid absorbs each arriving point with weight $1/v_c$, where v_c counts how many it has already taken. The result equals a running mean, at a fixed small cost per round.
Pseudo-label	A target the system computed for itself — here, a cluster index — used exactly as if a human had written it down.
Alternating minimisation	Optimise one half of a problem with the other frozen, then swap. DeepCluster's epoch is one round of this.
Overcomplete K	Asking for many more clusters than there are real classes (10,000 vs. 1,000), so that merged structure can still be split later.
Linear probe	Freeze the features, train only a linear classifier on top. It measures how *linearly available* the information is, not how good a classifier you could build.
Mid-layer transferability	The observation that the most transferable features sit in the middle of the network (conv3/conv4), not at the end, because the last layers over-specialise to the pretext task.
Feature correspondence	The table of angles between every patch of one image and every patch of another, in a frozen backbone's feature space. STEGO's raw material.
Unary / pairwise term	A CRF's two costs: what one pixel's own evidence says, and what it costs for two pixels to disagree. STEGO's loss has only the second.
Negative pressure	The bias b subtracted from the correspondence field, which turns weakly-agreeing pairs into repelling pairs and so prevents the head collapsing to a single direction.

Key takeaway — The one thing to carry out of this chapter

Better than chance is enough. Every self-supervised method in the rest of this book is a different answer to the question "where do I get a signal that beats a coin flip, and how do I stop the loop from amplifying its own mistakes?" DeepCluster's answer is a clustering plus three anti-collapse patches; STEGO's is a frozen backbone plus a bias term. Later chapters swap in augmentations (Chapters 8–9), a slowly-moving teacher copy (Chapters 10–11), or masked reconstruction (Chapter 5) — but the question never changes, and neither does the fact that the anti-collapse part is never optional.

3.6 Check Your Understanding

Check Your Understanding

Level 1 (recall).

1. Write down the Euclidean and cosine measures between descriptors $\boldsymbol{a}$ and $\boldsymbol{b}$, and state in one sentence which geometric quantity each emphasizes.
2. In the (r, c, p_1, p_2) definition of an LSH family, what two quantities are bounded, and what is the role of the gap factor c?
3. Define core, border, and noise points of DBSCAN in terms of ε and `min_samples`.

Level 2 (compute).

4. For $K = 2$ and points $0, 2, 4, 10$ on the line with initial centroids $\boldsymbol{\mu}_1 = 0$, $\boldsymbol{\mu}_2 = 6$, run Lloyd's algorithm to convergence and report the final within-cluster sum of squares.
5. In Figure 3.3, verify that F is the true nearest neighbor of $q = (4.6, 0.7)$ (check at least B and C), and state the two pruning tests — which split planes, with which numbers — that let the algorithm certify this after inspecting one leaf.
6. With $k = 4$ bits per band and a single-bit collision probability $s = 0.9$ for a true neighbor pair, how many tables L give at least 99% probability of recovering the pair in at least one bucket?
7. Using Table 3.3, compute (a) the gain of DeepCluster over the random baseline at conv5 on ImageNet, and (b) the gap between DeepCluster at conv3 and the supervised ImageNet-labels row at conv5. Comment on the two patterns.

Level 3 (reason).

8. Why does a KD-tree query degrade toward a linear scan in high dimension? Phrase your answer in terms of the pruning test of Figure 3.3.
9. Describe a 2D dataset where DBSCAN succeeds but every k-means run fails, and a second where k-means succeeds but any single-ε DBSCAN fails.
10. At initialization a convnet's features are random, yet DeepCluster improves from its first epoch on. What non-random structure does the first k-means pass exploit?

Level 4 (connect).

11. Map every term of the STEGO energy (3.13) onto the terms of the CRF energy (3.11). What plays the role of the unary term, if anything?
12. Both DeepCluster and a contrastive method train on targets derived from the data itself. Why is DeepCluster's target *weaker* per epoch, and what does the offline re-clustering step compensate?
13. State the trivial-solution failure of DeepCluster formally: write the joint objective's degenerate fixed point and name the paper's three concrete defenses (one for empty clusters, one for cluster balance, one architectural).
14. In Eq. (3.7), take the limit $b \to 1$. What sign does the force matrix $(F_{hwij} - b)$ then carry for almost every pair, and what behavior does the probe learn as a consequence?

Level 5 (extend).

14. STEGO couples a frozen DINO backbone, a learned low-dimensional probe, and a CRF. Design an ablation that tests the paper's claim that the probe *sharpens* rather than merely copies the backbone's correspondence, and predict its outcome for small objects given the CRF's boundary bias.

Solution sketches.

1. Euclidean $\|\boldsymbol{a} - \boldsymbol{b}\|^2$ measures displacement in the space; cosine $\boldsymbol{a}^\top \boldsymbol{b}/(\|\boldsymbol{a}\|\,\|\boldsymbol{b}\|)$ measures only direction, discarding magnitude.
2. The collision probability of *close* pairs ($d \le r$) is bounded below by p_1, that of *far* pairs ($d \ge c\,r$) above by p_2; c sets the gap between "close enough to usually collide" and "far enough to rarely collide."
3. Core: at least `min_samples` points within ε (counting itself, in the usual convention); border: fewer, but within ε of some core point; noise: neither, i.e. outside every core neighborhood.
4. Round 1: $0, 2$ are closer to $\boldsymbol{\mu}_1 = 0$ (the point 2 sits at distance 2 vs. 4) and $4, 10$ are closer to $\boldsymbol{\mu}_2 = 6$ (4 sits at distance 4 vs. 2), giving $\{0, 2\}, \{4, 10\}$; update $\boldsymbol{\mu}_1 = 1$, $\boldsymbol{\mu}_2 = 7$. Round 2: the point 4 now ties at distance 3 from each centroid. Break the tie toward $\boldsymbol{\mu}_2$: the assignment is unchanged, so Lloyd stops at $\{0, 2\}, \{4, 10\}$ with $\text{WCSS} = (1 + 1) + (9 + 9) = 20$. Break it toward $\boldsymbol{\mu}_1$ instead and the run continues to $\{0, 2, 4\}, \{10\}$ with $\text{WCSS} = 8$ — which is the global optimum (10 is the odd one out). The lesson: Lloyd's algorithm can stop at a fixed point more than twice as costly as the best one, and something as small as a tie-break decides which one you get.
5. $d(q, F) = \sqrt{0.4^2 + 0.1^2} \approx 0.41$; $d(q, B) = \sqrt{1.6^2 + 0.3^2} \approx 1.63$; $d(q, C) = 2.6$. Unwinding past F's parent: E's cell lies beyond the $y = 1.9$ plane, a gap of $1.2 > 0.41$, so it is pruned; at the root, the left subtree lies beyond $x = 4$, a gap of $0.6 > 0.41$, so A, B, C, D are pruned. One leaf inspected, exact answer certified.

6. Per table the pair fails to collide with probability $1 - s^k = 1 - 0.9^4 = 0.3439$; need $0.3439^L \leq 0.01$, so $L \geq \log 0.01 / \log 0.3439 \approx 4.32$: $L = 5$ tables.
7. (a) $38.2 - 14.1 = 24.1$ points over chance-floor random features at conv5, from a zero-shot-starting loop — the signature of a bootstrap with no supervision anywhere. (b) $50.5 - 41.0 = 9.5$ points behind full supervision at the best layer, the calendar- year gap self-supervision had left to close before the contrastive era; and DeepCluster peaks at conv3 (41.0) rather than conv5 — the mid-layer transferability pattern flagged in the text.
8. Pruning requires the best-distance ball around the query to miss the splitting plane. In high dimension, distances concentrate (all pairs are comparably far), so the ball's radius stays large while the exponentially many cells shrink: almost every plane cuts the ball, the test almost never fires, and nearly every leaf is visited.
9. Two interleaving half-moons: any bisector of two centroids cuts through one moon, but DBSCAN with moderate ε follows the density of each. Conversely, one dense Gaussian blob next to one sparse uniform square: k-means separates them by centroid distance, but no single ε fits both densities — the ε that resolves the sparse square dissolves the dense blob.
10. The architecture's inductive bias: convolutional locality places visually similar images at nearby feature loci even under random weights, so the first clustering is already better than chance — precisely the seed the loop amplifies (Section 3.3).
11. $w(\boldsymbol{v}_i, \boldsymbol{v}_j)$ realizes the pairwise edge — the data-dependent weight inside ψ_p — and $d(\boldsymbol{\phi}(\boldsymbol{v}_i), \boldsymbol{\phi}(\boldsymbol{v}_j))$ is the disagreement cost between two "assignments," with embeddings standing in for labels. There is no unary term inside (3.13): the frozen backbone's raw features play the unary's role externally, fixing the geometry the pairwise energy is asked to respect.
12. Pseudo-labels are stale (they reflect last epoch's features), discrete, and capped at K classes, so each epoch's signal is coarser than a per-pair contrastive target. The offline re-clustering compensates by handing each epoch a *globally re-fit* target that tracks the current (improving) feature geometry instead of a fixed pairing.
13. Solve Eq. (3.3) by assigning all images to cluster 1 (so k-means needs only $\boldsymbol{\mu}_1$, at the dataset mean), then Eq. (3.4) trains $g_W \circ f_{\boldsymbol{\theta}}$ to always answer "1" — zero loss, zero representation. Three concrete defenses keep training informative: empty-cluster reassignment (re-seed an empty centroid on a perturbed populous one); cluster-uniform sampling at training time; and a large overcomplete K (10,000 for ImageNet), which pre-commits mergeable structure rather than forcing a learner with only as many cells as classes.
14. As $b \to 1$, $F_{hwij} - b \leq 0$ for every pair (since $F \leq 1$), so attraction vanishes and repulsion dominates: the probe maximizes disagreement, spreading its outputs across directions with no cohesive clusters to find at inference. The same parameter taken to 0 yields the opposite flop — attraction only, and geometric collapse into one high-magnitude direction.

15. Hold the backbone fixed and compare (a) clustering the frozen features directly, (b) clustering the trained probe, and (c) a probe trained without the cross-image term, scored on a split with small-object annotations. "Sharpening" predicts (b) > (a) overall, and specifically (b) recovering objects absent from (a)'s clusters. Given the CRF's bias, the same small objects should lose boundary accuracy after refinement even as large regions gain — the ablation should report per-size-class scores to separate the two effects.

Concept Map — how this chapter connects

Backward. Chapter 1 posed the question this chapter answers with machinery: what signal can be extracted from X alone? Chapter 2 built every tool used here — the distance metrics and hyperspherical geometry, cross-entropy, KD-trees, LSH, k-means, DBSCAN/OPTICS, and PCA — and its failure-mode boxes are quoted throughout; the DeepCluster loop is the foundations chapter's k-means closed into backpropagation, exactly as promised there.

Forward. Chapter 4 replaces the classical structure extraction with a learned reconstruction objective (autoencoders). Chapter 10 studies the collapse taxonomy of online targets, where SwAV (Caron et al., 2020) replaces DeepCluster's offline loop with differentiable assignments. Chapter 11 trains the DINO features (Caron et al., 2021) that STEGO freezes — cosine-compared descriptor sets, exactly the slide-5 metric — and scales balanced k-means to billion-image data curation (Vo et al., 2024) over the ANN infrastructure of Section 3.1.4. Chapter 12 turns pseudo-labels into self-training, and Chapter 14 runs this same clustering and retrieval machinery inside autonomous systems.

CHAPTER 4

Autoencoders and Variational Autoencoders

> Every layer can be shallow, but they can be "stacked" to form a deep autoencoder.
>
> Lecture 3 (course numbering), slide on stacked autoencoders

In this chapter

4.1 The Autoencoder Idea: Compress, Then Reconstruct 142
4.2 The Information-Bottleneck View, and Why It Is Not a Guarantee 145
4.3 Training Autoencoders: Reconstruction Loss, Depth, BatchNorm 148
4.4 The Variational Autoencoder: A Latent-Variable Model 151
4.5 The ELBO, Step by Step ..... 155 • Derivation of the evidence lower bound
4.6 Making the ELBO Computable .. 157 • The reconstruction term is disguised squared error • The reparameterization trick: backprop through a sample • The KL term in closed form • The practical training objective • What a trained latent space buys: smooth manifolds
4.7 Why Does the VAE Actually Work? 169
4.8 Where the Lecture Goes Next: Masking Instead of Bottlenecking 170

Chapter roadmap

Before you start. You need: what a neural network layer is (Section 2.2) and squared-error loss (Section 2.3.2). For the second half you also need the Gaussian distribution, Jensen's inequality and KL divergence (Section 2.6) — but the chapter re-explains each of them where it uses them.

The idea in one sentence. Force information through a narrow pipe and demand it come out intact; whatever survives the squeeze is the structure of the data.

The arc, one difficulty step at a time.

- *The plain autoencoder*: squeeze and rebuild. Simple, and it works — until you ask it to *generate* something new, at which point it falls over. We will see exactly why.

- *The diagnosis*: nothing ever told the squeezed codes where to live, so most of the code space is territory the decoder has never visited.
- *The variational autoencoder*: the same hourglass, but the code becomes a *random* variable with a designed shape, so the whole code space becomes visited territory.
- *The mathematics*: the ELBO, derived line by line with a reason given for every line, and the reparameterisation trick — the one piece of cleverness that makes the whole thing trainable.

Afterwards you will be able to say what each of the VAE's two loss terms does and what happens when either one wins; derive the ELBO without looking it up; explain why you cannot simply sample a code from a plain autoencoder; and read Chapters 5 and 6 knowing what problem each is reacting to.

A warning about one word. "Variational" sounds forbidding. It means nothing more than: *we could not compute the thing we wanted, so we optimised the best computable approximation instead.* That is the entire content of the term.

The clustering methods of Chapter 3 exploited a weak but cheap signal: points that are close together are probably alike. This chapter introduces the first genuinely *learned* self-supervised signal of the course, and the one the lecture (Kingma and Welling, 2014; He et al., 2022) builds its whole middle act on: *take the input away and ask the model to give it back.* An autoencoder compresses an input $\boldsymbol{x}$ into a small latent code $\boldsymbol{z}$ and must reconstruct the input from that code alone. Whatever survives the bottleneck must, by necessity, be the structure of the data—so the encoder is forced to discover that structure without a single label.

The catch in "the label is the input". Supervised learning needs someone to label every $\boldsymbol{x}$ with a y. Here the label *is* $\boldsymbol{x}$, free. That makes reconstruction the cheapest learning task you could write down — and the easiest to cheat: simply copying the input gets a perfect score and learns nothing.

So all of classical autoencoder theory is about *preventing the copy*: squeeze the signal through a narrow waist, limit the network's capacity, or add noise, until the only way to pass is to understand the data.

Everyday analogy — ten pages of notes

Copying a 1,000-page book page by page teaches you nothing. But if you may keep only ten pages of notes, from which a classmate must rebuild the book, you are forced to find its themes, characters and logic — and to drop the margins and typos. The code $\boldsymbol{z}$ is those ten pages.

Where the analogy breaks. The classmate here (the decoder) is trained alongside you, so the two of you can invent a private shorthand. A narrow bottleneck limits that, but does not rule it out entirely.

Every method in this chapter is a different way of limiting the notebook.

We follow the lecture's arc: the classical autoencoder and its failure to generalize, the practical question of how such networks were ever trained, and then the variational autoencoder (VAE), which turns the same hourglass into a principled generative model. The chapter stops where Lecture 3 pivots to masked autoencoders; that continuation is Chapter 5.

4.1 The Autoencoder Idea: Compress, Then Reconstruct

Model at a glance: Autoencoder (AE) idea from the 1980s; deep versions from 2006 onward

In one line	A network shaped like an hourglass: squeeze the input down to a few numbers, then rebuild the input from those numbers alone.
Problem it solves	Learning useful features with no labels. The target is the input itself, so the supervision is free.
Key intuition	If the pipe in the middle is narrower than the input, the network cannot simply copy — it must decide what to throw away. Deciding what to throw away *is* understanding the data.
Parts	An *encoder* $e_{\boldsymbol{\phi}} : \boldsymbol{x} \mapsto \boldsymbol{z}$, a *bottleneck* code $\boldsymbol{z}$ with far fewer numbers than $\boldsymbol{x}$, and a *decoder* $d_{\boldsymbol{\theta}} : \boldsymbol{z} \mapsto \hat{\boldsymbol{x}}$.
Loss	$\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2$ — how wrong the rebuild was. Nothing else.
What you keep	The encoder. The decoder is scaffolding.
Good at	Compression, denoising, dimensionality reduction, and historically at initialising deep networks layer by layer.
Bad at	Generating new data. Draw a random code, decode it, and you usually get nonsense — and this single failure is what the rest of the chapter is about.
Leads to	The VAE (this chapter), the denoising autoencoder (Chapter 7) and the masked autoencoder (Chapter 5).

Key Concept — Autoencoder and bottleneck

An *autoencoder* (AE) is a pair of learned maps: an *encoder* $e_{\boldsymbol{\phi}} : \mathbb{R}^D \to \mathbb{R}^K$ with parameters $\boldsymbol{\phi}$ and a *decoder* $d_{\boldsymbol{\theta}} : \mathbb{R}^K \to \mathbb{R}^D$ with parameters $\boldsymbol{\theta}$, trained so that $\hat{\boldsymbol{x}} = d_{\boldsymbol{\theta}}(e_{\boldsymbol{\phi}}(\boldsymbol{x})) \approx \boldsymbol{x}$ for data points $\boldsymbol{x} \in \mathbb{R}^D$. When $K \ll D$, the code $\boldsymbol{z} = e_{\boldsymbol{\phi}}(\boldsymbol{x})$ is a *bottleneck*: a low-dimensional summary through which all information needed for reconstruction must pass.

Figure 4.1 shows the canonical architecture with concrete dimensions. Why should compress-then-reconstruct teach a network anything? Because a 784-dimensional photograph does not contain 784 independent numbers worth of information: pixels are correlated by edges, lighting, objects—structure. A code of 32 numbers that suffices to rebuild the image must have captured exactly that structure; redundancy that does not survive the bottleneck was, by definition, compressible. This is the dogma of Chapter 1—structure in $\boldsymbol{x}$ yields signal without labels—made architectural. The lecture's own slide lists three hoped-for outcomes

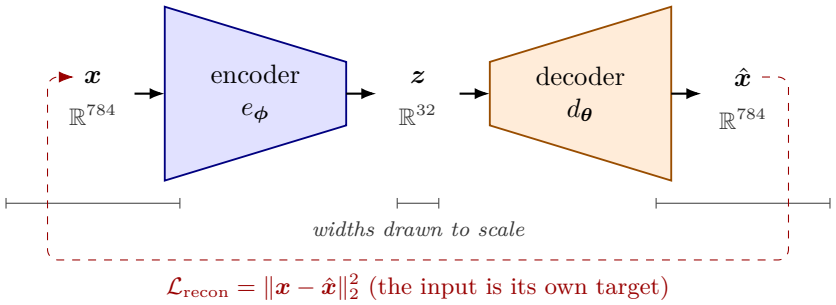


Figure 4.1. The autoencoder hourglass, the lecture's recurring pictogram. The encoder $e_{\boldsymbol{\phi}}$ maps an input $\boldsymbol{x} \in \mathbb{R}^{784}$ (e.g. a 28×28 grayscale image) to a narrow code $\boldsymbol{z} \in \mathbb{R}^{32}$; the decoder $d_{\boldsymbol{\theta}}$ maps the code back to a reconstruction $\hat{\boldsymbol{x}}$ of the original dimensionality. The dimensional widths of the three stages are drawn to scale: the code is the thin waist of the hourglass. Training minimizes the reconstruction error between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$ (dashed loop); the input itself is the target, so no labels are used.

of this design: *remove redundant information*, *reduce dimensionality*, and *discover abstract patterns*. Hold the third one loosely. The first two are guaranteed by the architecture; the third is a design goal, not a theorem, and most of Sections 4.2 through 4.3 is about the ways pattern discovery fails to happen. Whether $K \ll D$ buys abstraction or merely index-shrinkage is decided by the knobs below.

Key Concept — Undercomplete autoencoder

An autoencoder is *undercomplete* when $K = \dim(\boldsymbol{z}) < D = \dim(\boldsymbol{x})$, so the code cannot hold a copy of its input. The term carries the whole classical intuition at once: undercompleteness is precisely the degree of freedom the encoder does *not* have, and the hope of the method is that this scarcity is what forces semantic compression (named on Lecture 3, slide on hyperparameters, p. 2).

Worked Example — Linear autoencoders are PCA

Strip the hourglass to the bone: one fully connected layer, no nonlinearity anywhere, $e_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{W}_e\boldsymbol{x}$, $d_{\boldsymbol{\theta}}(\boldsymbol{z}) = \boldsymbol{W}_d\boldsymbol{z}$. The reconstruction loss becomes

$$\mathcal{L} \;=\; \frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{x}^{(i)} - \boldsymbol{W}_d\boldsymbol{W}_e\boldsymbol{x}^{(i)}\|_2^2,$$

which, with $\operatorname{rank}(\boldsymbol{W}_d\boldsymbol{W}_e) \le K < D$, is exactly the problem of finding the best rank-K linear approximation of the data. By the Eckart–Young–Mirsky theorem (Jolliffe, 2002), the optimum projects $\boldsymbol{x}$ onto the subspace spanned by the K leading eigenvectors of the data covariance—that is, onto the principal components. Two morals. First, the AE generalizes PCA the way a ReLU stack generalizes a matrix: keep the objective, replace the map. Second, adding depth buys nothing unless the waist stays narrow *and* the

functions stay tame; unlimited capacity lets a nonlinear encoder thread a one-dimensional curve through every training point and store each sample's arclength along it as its code—a degenerate "PCA" that reconstructs perfectly and means nothing (this is the memorization failure of Section 4.2 in its purest geometric form).

The lecture lists three design knobs, each of which reappears throughout the chapter (Kingma and Welling, 2019). **Dimensionality of z.** Usually much smaller than x; too small and reconstruction is impossible in principle, too large and compression is vacuous. **Capacity of encoder and decoder.** Too small and the loss will not converge; too large and the network can *store the dataset* instead of learning structure (Section 4.2). **Regularization.** Constraints that *shape* the latent space—weight decay, dropout, noise injection, or a distributional prior. The VAE of Sections 4.4–4.5 is the principled version of this knob, replacing a free-form code with a structured distribution.

Three different knobs. They are not three ways of saying "make it small":

- **Code size** limits *how many numbers* the code may carry.
- **Network capacity** limits *how clever* the encoder can be. A big enough encoder can learn a lookup table of the training images — and then code size stops mattering.
- **Regularisation** limits *which kinds* of code are cheap.

They interact: a wide code with regularisation can be more restrictive than a narrow code without; a narrow code is defeated by a huge encoder (the "codebook" of the next section).

What none of them does is *define* what counts as signal. The lecture's answer: the code should keep what is *explainable from the dataset* — textures, shapes and objects that recur across images — and drop per-image noise, which never recurs. But give the network enough capacity and even sensor noise becomes "explainable", one memorised image at a time. Section 4.2 puts a price on that failure.

Learning note from study sessions

The PID-controller reading of an autoencoder that surfaced in study sessions: the three knobs above are the proportional choke, the plant gain, and the setpoint shaping, and they are tuned against a single scalar readout—held-out reconstruction error. Train error near zero tells you nothing you did not already know; it is the *gap to validation error* that reports which knobs are currently miscalibrated. A wide gap at tiny waist is undercapacity; a wide gap at generous waist is codebook memorization; a small gap at high error is simply a bad budget. One number, three distinct repairs.

4.2 The Information-Bottleneck View, and Why It Is Not a Guarantee

Chapter 2 introduced mutual information $I(X;T)$, the number of bits a variable T carries about X. The *information bottleneck (IB) principle* (Tishby et al., 2000) frames every hidden layer T of a network as a negotiated settlement between two opposing pressures:

$$\min_{T} \; I(X;T) \; - \; \beta \, I(T;Y), \tag{4.1}$$

where X is the input, T the representation, and Y what we care about.

- $I(X;T)$ — how much of the input the representation remembers. The objective *charges* for this: memory costs.
- $I(T;Y)$ — how much the representation tells us about Y. The objective *pays* for this: usefulness earns.
- β — the exchange rate between the charge and the pay.

The ideal representation keeps everything about X that matters for Y, and nothing more — a *minimal sufficient statistic*. (Recall from Chapter 2 that $I(X;T) = H(X) - H(X \mid T)$: how much knowing T reduces your uncertainty about X.)

Simplified intuition vs. more precise view

Simple story: a network learns by keeping useful information and throwing the rest away, and Equation (4.1) measures exactly that.

More precise view: Equation (4.1) is a *design principle*, not a theorem you can check. Mutual information between high-dimensional continuous vectors is very hard even to estimate; the famous claim that deep networks train by first expanding and then compressing $I(X;T)$ did not hold up under later scrutiny; and for fully deterministic networks the quantities behave in degenerate ways. We use the IB view as the lecture does — as a vocabulary for the two pressures every method in Part II balances, not as something anyone optimises directly.

What the bottleneck is supposed to keep. An autoencoder has no labels; its target is its own input. So which information should survive the squeeze? The answer: *whatever many images have in common*, and not what is unique to one image.

Why does the bottleneck enforce that? Think about cost. The shape of a dog's snout appears in thousands of training images, so one learned description serves them all — cheap per image. The random sensor noise in one particular photo appears in no other image, so describing it would cost a full, separate description for that single image. A narrow bottleneck cannot afford that. It keeps the shared structure and drops the one-off noise — which is exactly what we wanted.

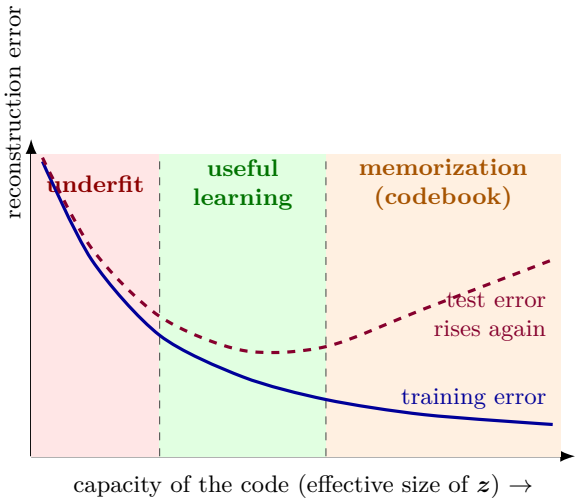


Figure 4.2. Schematic reconstruction error as a function of how much compression the latent code permits. *Left (underfit):* too little capacity—the code cannot hold the signal, error saturates at a high value for train and test alike. *Middle (useful learning):* the code is forced to retain exactly the structure shared across the dataset; test error tracks training error. *Right (memorization):* with enough effective capacity, even a one-dimensional $\boldsymbol{z}$ can index the training set (a *codebook*), driving training error toward zero while giving nothing on unseen data — the dashed test-error curve turns back up. The IB objective (4.1) is the attempt to stand precisely at the green region's left edge.

Figure 4.2 makes the tension concrete. The lecture's example is a dataset of three dog photos. Suppose $\boldsymbol{z}$ is a *single scalar*. There are only three training points, so a sufficiently clever encoder can emit $z = 1, 2, 3$ as an index, and a sufficiently powerful decoder can store one image per integer. The training reconstruction is then perfect, the code is as small as possible—and nothing has been learned: a fourth dog, unseen at training time, maps to a meaningless index. The hourglass alone cannot distinguish "$\boldsymbol{z}$ summarizes the concept *dog*" from "$\boldsymbol{z}$ is a primary key into a lookup table."

The mechanism deserves one more sentence because students reliably underestimate it: the distinguishing information never passes through the bottleneck at all. The encoder is doing the recognition—it is, effectively, a classifier that has memorized which of the three training images it is currently looking at—and it communicates its verdict to the decoder through a channel of capacity three-ish values. A one-dimensional code only limits the *number of distinct messages*, not the *richness of what each message can invoke*: the decoder is a free-parameter library of full-resolution images waiting to be selected. Nothing in the L2 reconstruction objective (4.2) prefers learning descriptions over learning indices. Both achieve zero training loss, so the loss function has literally nothing further to say; the difference between them lives entirely in the held-out error, where the codebook collapses and the real compressor does not.

Worked Example — The three-dog codebook, made numeric

The dataset is three images of 784 pixels each. A serious code needs to describe roughly $3 \times 784 = 2352$ distinct pixel values. With $K = 1$, the bottleneck channel can carry at most one real number—so how does the AE cheat? It does not carry information about the *pixel content*; it carries information about the *identity*. Three legal messages fit easily along one dimension, say $\{-1, 0, +1\}$ at decoder-threshold distance, and $N = 3$ training points is trivially few enough for a multilayer encoder to map onto those three values with wide margins. The decoder then stores three full images in its weights—at 784 outputs, comfortably within a few thousand parameters—and emits the one selected. Train loss = 0, test loss = disaster. Now scale up: with N training images and a latent z of any dimension ≥ 1, a continuous non-overlapping assignment of one parameter value per training point always exists in principle; what makes the scheme visible in practice is that the required per-sample reconstruction is implemented as N disjoint decoder basins, so between-sample codes decode to garbage. That smoothness failure is the diagnostic fingerprint: a genuine compressor tolerates perturbations of $\boldsymbol{z}$; a codebook does not. Chapter 5's MAE will block this shortcut from the other end, by withholding the content rather than rationing the channel.

Failure Modes

Failure modes of the plain autoencoder. (1) Memorization / codebook. With enough encoder–decoder capacity, the AE assigns each training point an index and stores the dataset in the decoder weights (Figure 4.2, right; Worked Example 4.2). Training loss goes to zero; generalization is nil. The cure is not a narrower waist alone but *interference*: noise on the input (denoising AEs, Chapter 7), dropped content (masked AEs, Chapter 5), or a distributional prior on the code (the VAE, Section 4.4). **(2) Identity mapping.** If $K \geq D$ and nothing constrains the code, $\boldsymbol{z} = \boldsymbol{x}$ is a trivial optimum. **(3) Free-form latent geometry.** Even a successful AE leaves $\boldsymbol{z}$-space unstructured: points near a training code may decode to garbage, and there is no sampling recipe. This last defect is exactly what the VAE fixes, so it gets its own box in Section 4.6.4.

Common Confusion

"Noise" is budget-relative, not absolute. Two tempting mistakes meet here. First, believing that a sufficiently narrow $\boldsymbol{z}$ *must* discard sensor noise: false—Worked Example 4.2 shows a narrow code storing the dataset by identity. Second, believing that discarding fine detail *always* constitutes success: also false—an AE that blurs every face into the dataset average has "discarded noise" so aggressively it has discarded the object too. The right criterion is comparative: information worth keeping is exactly whatever *generalizes to held-out data at the same code budget*. Whatever survives that test is, operationally, structure; whatever does not is, operationally, noise—for *this* network, *this* data, *this* budget. Change any of the three and the dividing line moves.

4.3 Training Autoencoders: Reconstruction Loss, Depth, BatchNorm

The loss. The canonical training objective measures squared L_2 distance between input and reconstruction:

$$\mathcal{L}_{\text{AE}}(\boldsymbol{\theta}, \boldsymbol{\phi}) \;=\; \frac{1}{N}\sum_{i=1}^{N} \left\| \boldsymbol{x}^{(i)} - d_{\boldsymbol{\theta}}(e_{\boldsymbol{\phi}}(\boldsymbol{x}^{(i)})) \right\|_2^2, \tag{4.2}$$

where N is the number of training images, $\boldsymbol{x}^{(i)}$ the i-th one, and $\|\cdot\|_2^2$ adds up the squared pixel differences. (The lecture writes the norm without the square; squaring gives the same best solution and matches the probabilistic reading used later.)

The input is its own label — self-supervision in its purest form.

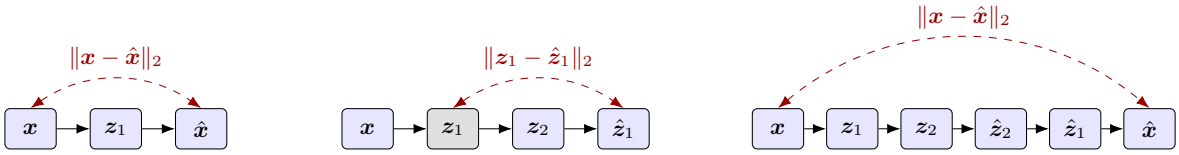
Two remarks that matter later:

- **Squared error rewards the average.** The best possible prediction under squared error is the *average* of all plausible answers. Averaging many sharp images gives a blurry one — which is the root of the VAE's blurriness (Section 4.6.4).
- **Squaring is a choice.** A pixel off by 0.5 costs 25 times more than one off by 0.1 (0.25 vs 0.01). That weighting is exactly what Gaussian noise implies, and Section 4.6 shows squared error is really a log-likelihood in disguise.

Common Confusion

What "collapse" means for an AE. Readers arriving from contrastive learning expect *collapse* to mean "everything maps to one point." An autoencoder cannot collapse that way—the reconstruction term forbids it, since identical codes require identical outputs and the inputs differ. The AE's degenerate optimum is the opposite failure: the *identity*, where each code is too rich rather than too poor, and nothing was paid by the encoder. The codebook of Figure 4.2 is a subtler variant of the same direction: *no sharing*, rather than *total sharing*. Keep the two directions straight when the failure-mode vocabulary returns for VAEs (posterior collapse, Section 4.6.4) and for contrastive models (Chapter 9).

The historical depth problem. Early AEs were shallow for an unglamorous reason: with random initialization, gradients backpropagated through more than about three layers became vanishingly small, and deep networks simply would not train. The mechanism is a chain-of-multiplications: with saturating sigmoids whose derivatives never exceed 0.25, the Jacobian product connecting the loss to a weight twelve layers down has norm scaling like $0.25^{12} \approx 6 \times 10^{-8}$, and the weight sits still. The pre-2010 workaround was *layer-wise greedy pretraining*, shown in Figure 4.3: train a two-layer AE on the raw input, freeze the trained encoder, run every training point through it once, and *use the resulting codes as the new dataset*—repeating the recipe on this cleaner, lower-dimensional problem, one shallow layer at a time (Vincent et al., 2010). The freezing is essential, not incidental: it is what makes the inner layer's input distribution *stationary*, so that layer two optimizes against a fixed



(a) train layer 1 on raw input

(b) freeze layer 1, train layer 2 on codes

(c) unroll and fine-tune the deep stack

Figure 4.3. Greedy layer-wise pretraining of a stacked autoencoder, as a sequence of three shallow problems. (a) A single AE is trained on the raw input with loss $\|\boldsymbol{x}-\hat{\boldsymbol{x}}\|_2$. (b) Its encoder (gray: frozen) produces codes $\boldsymbol{z}_1$ that become the "data" for training the next layer, with inner loss $\|\boldsymbol{z}_1-\hat{\boldsymbol{z}}_1\|_2$. (c) After all layers are pretrained, the stack is unrolled $\boldsymbol{x}\to\boldsymbol{z}_1\to\boldsymbol{z}_2\to\hat{\boldsymbol{z}}_2\to\hat{\boldsymbol{z}}_1\to\hat{\boldsymbol{x}}$ and fine-tuned end-to-end on the outer loss. Each training step sees only a shallow network, sidestepping vanishing gradients.

target function rather than a moving one. Stacked denoising variants (Vincent et al., 2008) applied the same recipe to noisy inputs. Once stacked, the layers are unrolled into a single deep hourglass $x\to z_1\to z_2\to\cdots\to\hat{z}_2\to\hat{z}_1\to\hat{x}$ and fine-tuned end-to-end. Notice the two losses in Figure 4.3: the outermost pair minimizes pixel error $\|\boldsymbol{x}-\hat{\boldsymbol{x}}\|_2$, while each inner pair minimizes *code error* $\|\boldsymbol{z}_i-\hat{\boldsymbol{z}}_i\|_2$—each layer gets the reconstruction treatment at its own level of abstraction. Greedy pretraining was the dominant deep-learning trick of its era; Figure 4.3 is now mostly of historical interest, because of the next item.

The modern fix: BatchNorm. The lecture's one-line summary of a decade of progress is that the depth limitation is "usually not a problem anymore with BatchNorm" (Ioffe and Szegedy, 2015). Batch Normalization inserts, after each layer, a per-feature rescaling computed *over the current mini-batch*: each feature is shifted to zero mean and unit variance within the batch, then given back two learned degrees of freedom. Formally, for feature d over a batch of B samples with activations $x_1,\dots,x_B$:

$$\mu_b=\frac{1}{B}\sum_{s=1}^{B}x_s,\qquad \sigma_b^2=\frac{1}{B}\sum_{s=1}^{B}(x_s-\mu_b)^2,\qquad \hat{x}_s=\frac{x_s-\mu_b}{\sqrt{\sigma_b^2+\epsilon}},\qquad y_s=\gamma\,\hat{x}_s+\beta, \tag{4.3}$$

where μ_b,σ_b^2 are the batch mean and variance of this feature, $\hat{x}_s$ the normalized activation, $\epsilon>0$ a small constant for numerical stability (10^{-2} in our example; 10^{-5} in practice), and γ,β learned per-feature gain and shift that let the layer undo the normalization if doing so helps the loss.

Why normalisation lets networks go deeper. Small growth in each layer compounds. If every layer makes its outputs just 5% larger, then after ten layers they are $1.05^{10}\approx 1.6$ times larger, enough to push the next non-linearity into its flat, useless region. Batch normalisation is cruise control: whatever scale arrives, each layer passes on inputs with mean 0 and variance 1, so drift cannot build up.

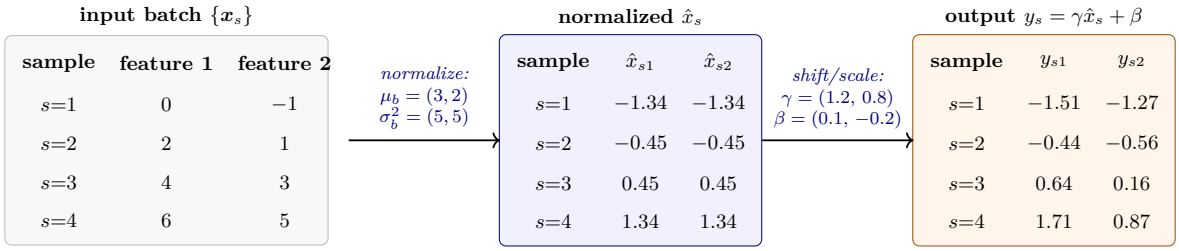


Figure 4.4. Batch Normalization, worked through Equation (4.3) on a concrete batch: $B = 4$ samples, two features. *Left:* raw activations. *Middle:* each feature column is normalized by its own batch statistics—e.g. feature 1 has $\mu_b = (0+2+4+6)/4 = 3$, $\sigma_b^2 = (9+1+1+9)/4 = 5$, so s=1 maps to $(0-3)/\sqrt{5.01} \approx -1.34$ (with $\epsilon = 0.01$). *Right:* the learned affine step rescales each feature independently, restoring expressive power. Note that the two input features end up with different output spreads—γ and β recover whatever scale the downstream layers prefer.

Why the learned γ, β. Forcing mean 0 and variance 1 everywhere would limit what a layer can express. So the layer then multiplies by a learned γ and adds a learned β, giving back the freedom to choose its own scale and shift — but only when training says it helps.

A side effect. Because the mean and variance are computed over the batch, each example's output depends slightly on the other examples in the batch. That acts as a mild regulariser. At test time the batch statistics are replaced by running averages saved during training, so a prediction no longer depends on its batch-mates. Figure 4.4 works one batch through Equation (4.3) with real numbers — check one row by hand, because the exercises assume you can.

Keep the roles clean, because the lecture's phrasing invites a conflation it does not endorse: BatchNorm fixes *optimization*, not *memorization*. A deep autoencoder with BatchNorm trains perfectly smoothly—and can use that smooth training to converge *faster* onto exactly the lookup table of Figure 4.2. The two axes are independent knobs; Sections 4.1–4.2 govern what can be learned, this section governs whether gradient descent can find it. Confusing them is the highest-frequency reasoning error the examiners report for this material.

With BatchNorm (and its cousins LayerNorm and residual connections), deep AEs train directly end-to-end, and the greedy recipe of Figure 4.3 survives mainly as the ancestor of a later, more important idea: *pretrain without labels, then fine-tune with them.* Chapter 5's masked autoencoder is that idea at scale.

```
# e_phi: encoder with BatchNorm + ReLU blocks, output dim K
# d_theta: decoder mirroring it, output dim D
x_hat = d_theta(e_phi(x))              # forward pass: full hourglass
loss    = ((x - x_hat) ** 2).mean()    # Eq. (eq:ch04-recon-loss)
loss.backward()                        # gradients flow decoder -> encoder
optimizer.step()                       # one SGD/Adam update of (phi, theta)
```

Listing 4.1. A compact deterministic autoencoder training step, including the batch-normalized encoder of this section. The forward pass returns both the reconstruction and the code because downstream code later consumes $\boldsymbol{z}$; training touches only the reconstruction.

The loop hides nothing—which is the point. There is no place to inject labels, no contrastive pair, and no special sampling; the entire SSL content of the classical AE is the line that computes $\hat{\boldsymbol{x}}$ from $\boldsymbol{x}$ and the identity of the target with the input. The single genuinely modern line is `e_phi`, whose internals (BatchNorm after each linear layer) are exactly what freed this loop from the stacking machinery of Figure 4.3 a decade ago.

4.4 The Variational Autoencoder: A Latent-Variable Model

Model at a glance: Variational Autoencoder (VAE) Kingma & Welling, 2014; Rezende, Mohamed & Wierstra, 2014

In one line	An autoencoder whose code is deliberately fuzzy, plus a penalty that keeps all the fuzzy codes packed into one tidy blob — which makes the code space safe to sample from.
Problem it solves	A plain autoencoder's codes land wherever they like, leaving holes the decoder has never seen. Draw a code at random and you land in a hole. The VAE removes the holes.
The two changes	(1) The encoder outputs a *mean* and a *spread* instead of a single point, and the code is drawn from that Gaussian. (2) A second loss term pulls every one of those little Gaussians towards the same standard Gaussian.
Why fuzziness helps	If one image's code is a small cloud rather than a point, the decoder is forced to produce a sensible image for *every point in the cloud*. Overlapping clouds then tile the space with no gaps left over.
Loss	$\underbrace{\lVert \boldsymbol{x} - \hat{\boldsymbol{x}} \rVert^2}_{\text{rebuild it properly}} + \beta \underbrace{D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, \mathcal{N}(\mathbf{0}, \mathbf{I})\big)}_{\text{and keep the codes tidy}}$ — two forces in direct opposition.
Training	Ordinary backpropagation, once the *reparameterisation trick* removes the random draw from the gradient path.
Sampling	Draw $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, run the decoder. One pass, no iteration.
Main failure modes	Blurry samples (the squared-error term averages over everything the code did not pin down); *posterior collapse* (the KL term wins, the code carries nothing, the decoder ignores it).

Position in the family	The VAE computes a *bound* on the likelihood, not the likelihood. Chapter 6's flows pay an architectural price to get the exact number; Chapter 7's diffusion models give up one-pass sampling to get better pictures.

Everyday analogy — why a fuzzy code fixes a sharp problem

Think of the code space as a map and each training image as a pin stuck in it. A plain autoencoder places pins wherever it likes: dense in some regions, empty in others. Throw a dart at the map later and you will usually hit emptiness — territory the decoder has never had to make sense of, so out comes garbage.

The VAE replaces each pin with a small blot of ink. The decoder is now graded on the whole blot, not just its centre, so it must produce something sensible anywhere within it. And the second loss term is a rule saying "everybody's blot must sit near the middle of the map". Between them the blots overlap and cover the map, so a dart now lands somewhere the decoder has been trained and a plausible image comes out.

Where the analogy breaks. Overlapping blots also mean two different images can share code space — which is exactly why VAE samples are blurry. The decoder is averaging over everything the code did not pin down. The blur is not a bug to be fixed later; it is the price of the same overlap that made sampling work at all.

The plain AE's last failure mode was a free-form, unsampleable latent space. The *variational autoencoder* (Kingma and Welling, 2014; Rezende et al., 2014) repairs this by upgrading $\boldsymbol{z}$ from a code to a *random variable* with a designed distribution.

Put the complaint concretely: run a trained AE's encoder over the dataset and scatter-plot the codes. They land in some shapeless constellation—clusters where dogs are, a spur where text is, and across most of the space nothing at all: regions where the decoder was never once asked to perform, and where its output is consequently ungoverned. Try to *generate* by feeding the decoder a random $\boldsymbol{z}$ and the odds are overwhelming that you have picked such a hole; what comes back is pixel soup. It is not that the AE stored bad descriptions—it stored unindexed ones. There is no map of where anything lives, so you cannot take a walk in latent space.

The VAE's fix, in three pieces.

1. **Make both $\boldsymbol{x}$ and $\boldsymbol{z}$ random.** The decoder no longer maps a code to *an* image; it gives a *distribution* over images.
2. **Choose the prior $p(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ up front.** A simple round blob that is trivial to sample from. The idea: any point drawn from it should decode to something sensible.
3. **Let the encoder output a distribution, not a point** — a mean and a spread — and charge the ELBO (Section 4.5) whenever that distribution strays from the prior.

Result: each training image now occupies a little *region* of code space, and all those regions are pulled towards the same blob until they overlap and the gaps close — exactly what the plain autoencoder was missing. In one sentence: *replace a point code with a distribution plus a penalty, and the latent space becomes an address book you can pick new addresses from.*

Common Confusion

Which Greek letter owns which network? The deck itself switches conventions between slides: its formulation slide labels the encoder $q_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})$ and the decoder $p_{\boldsymbol{\phi}}(\boldsymbol{x} \mid \boldsymbol{z})$ in prose, but its own ELBO equation uses $q_{\boldsymbol{\phi}}$ and $p_{\boldsymbol{\theta}}$, and the figures fifteen slides later keep $q_{\boldsymbol{\phi}}$, $p_{\boldsymbol{\theta}}$. We follow the equation and the literature: $\boldsymbol{\phi}$ *belongs to the inference/recognition network (encoder),* $\boldsymbol{\theta}$ *to the generative network (decoder).* Whenever the slides' bullet text disagrees with their displayed math on this point, trust the math (Kingma and Welling, 2019).

Key Concept — VAE

A *variational autoencoder* is a latent-variable generative model. It posits a *prior* over codes, $p(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, and a learned *decoder* (likelihood) $p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})$ with parameters $\boldsymbol{\theta}$: to generate data, draw $\boldsymbol{z} \sim p(\boldsymbol{z})$ and pass it through the decoder. Because the true posterior $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})$ is intractable, a learned *encoder* (recognition model) $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \operatorname{diag} \boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\boldsymbol{x}))$ with parameters $\boldsymbol{\phi}$ approximates it, predicting the mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ of a diagonal Gaussian over codes.

Historical Note

The VAE appeared nearly simultaneously in two 2014 papers. Kingma and Welling (2014), presented at ICLR 2014 as *Auto-Encoding Variational Bayes*, gave the model its name, its ELBO objective, and the reparameterization-trick training recipe used everywhere today. Rezende et al. (2014), at ICML 2014, introduced the same architecture from the direction of *stochastic backpropagation* in deep latent Gaussian models. The two lineages emphasized complementary halves—variational Bayes for the former, recognition networks for the latter—and within two years Doersch (2016) and the tutorial of Kingma and Welling (2019) had fused them into the standard account reproduced in the lecture. The reconstruction loss framing, in turn, descends from the classic autoencoder work of the 1980s–2000s, which Vincent et al. (2010) extended with stacked denoising pretraining (Figure 4.3).

Figure 4.5 shows the whole model in one picture (after Kingma and Welling, 2019; the lecture draws the prior, encoder, decoder and data as boxes linking the small z-space to the image space, Kingma and Welling, 2014). Two points to notice:

- **The prior is a choice.** Saying "codes follow $\mathcal{N}(\mathbf{0}, \mathbf{I})$" is a decision we make, not something learned. It is what makes sampling free (just draw from a Gaussian) and interpolation meaningful.

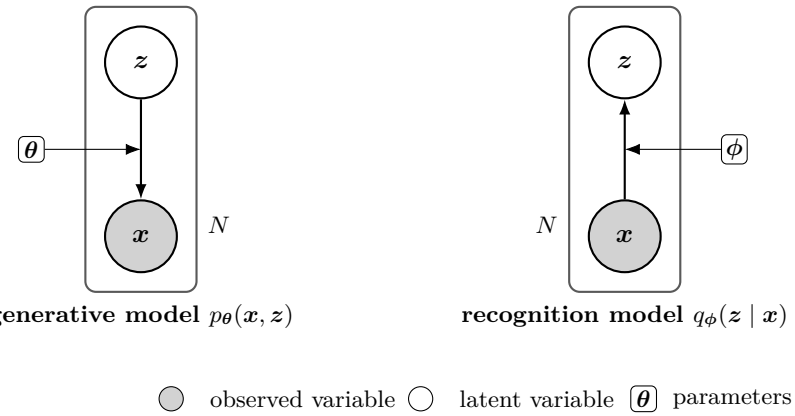


Figure 4.5. Plate diagram of the VAE, following the idiomatic convention used in a decade of ML lecture notes: shaded circles are observed variables, unshaded circles are latent, small solid boxes are parameters, and an enclosing *plate* indexed N means the contents repeat once per dataset sample. *Left:* the generative model—parameters $\boldsymbol{\theta}$ define the decoder distribution, and each sample follows $\boldsymbol{z} \to \boldsymbol{x}$: draw a code from the prior, decode it. The true posterior $\boldsymbol{z} \mid \boldsymbol{x}$ would require inverting that arrow, which is intractable. *Right:* the recognition model supplies the missing inference direction, $\boldsymbol{x} \to \boldsymbol{z}$, with its own parameters $\boldsymbol{\phi}$.

- **The two networks do different jobs.** Given an image, the encoder proposes a *range* of plausible codes. Given a code, the decoder says how well it explains the image.

The box below clears up a common misreading — that a VAE is just "an autoencoder with noise added".

Common Confusion

AE vs. DAE vs. VAE — three autoencoders, three different signals. A plain *AE* (Section 4.1) answers: *what code reproduces $\boldsymbol{x}$?* — a compression signal. A *denoising AE* (DAE; Chapter 7 expands it into the story of diffusion) feeds a *corrupted* input $\tilde{\boldsymbol{x}}$ and asks for the clean $\boldsymbol{x}$ back (Vincent et al., 2008): *what code survives corruption?* — a robustness signal. A *VAE* never commits to one code: it asks *what distribution over codes explains $\boldsymbol{x}$, while staying consistent with a global prior?* — a probabilistic-density signal. All three share the hourglass silhouette, but they optimize genuinely different objectives and fail in genuinely different ways; memorization hurts the AE, aggressive corruption hurts the DAE, and blur plus prior holes hurt the VAE (Sections 4.2 and 4.6.4).

The encoder is only needed for training. A VAE can generate new data, which a plain autoencoder cannot: throw the encoder away, draw $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, decode, done. Figure 4.6 shows this split between training and generating (the lecture's "VAE is a generative model" diagram).

The encoder's only job is bookkeeping during training: it supplies the approximate posterior that the ELBO needs. Once the decoder has learned, the encoder is never used

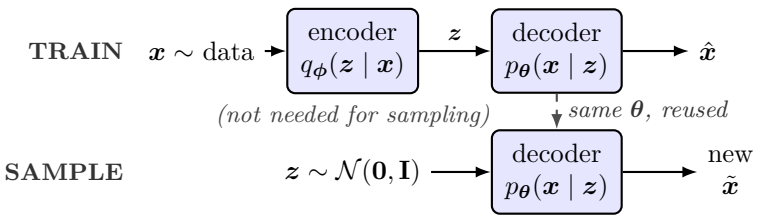


Figure 4.6. The VAE's two lives. *Top row (train):* real data flows through encoder *and* decoder, optimizing the ELBO over $(\boldsymbol{\theta}, \boldsymbol{\phi})$ jointly. *Bottom row (sample):* the encoder is gone; a fresh draw from the fixed prior passes through the decoder alone to synthesize a new data point. This asymmetry—train with two networks, deploy with one—is the defining operational signature of a generative model in this book, and Chapter 5's MAE will reproduce it in a different key (train with encoder and decoder, transfer with the encoder alone).

again for generation. That switch — from "compress" to "generate" — leads straight into Chapter 6, where generating data is itself the self-supervised task.

4.5 The ELBO, Step by Step

The goal, and why it is hard. We want to adjust $\boldsymbol{\theta}$ so that real images get high probability: make $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ large. But

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \log \int p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z})\, d\boldsymbol{z}$$

means "add up, over *every* possible code $\boldsymbol{z}$, how well that code explains $\boldsymbol{x}$". Two things make that impossible:

1. **Too many codes.** $\boldsymbol{z}$ is continuous and has many dimensions; there is no way to sum over all of them.
2. **We can't go backwards either.** The "which codes explain this image?" distribution, $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) / p_{\boldsymbol{\theta}}(\boldsymbol{x})$, has the same impossible integral in its denominator.

Older models were designed so this integral had a formula. With a neural network as the decoder, it does not. The way out is a *lower bound* that we can compute. One tool is needed first.

Key Concept — Jensen's inequality

For a convex function h and random variable X, $\mathbb{E}[h(X)] \geq h(\mathbb{E}[X])$; for a concave function the inequality flips. Picture a chord drawn between two points of the curve: it lies *above* a convex curve and *below* a concave one—averaging before versus after applying h is what the inequality compares. Since log is concave,

$$\mathbb{E}[\log X] \;\leq\; \log \mathbb{E}[X], \tag{4.4}$$

where X is any positive random variable. Applied to an integral written as an expectation, this lets us pull a log *inside* an expectation at the price of turning equality into a lower bound. (The lecture credits the Jensen/ELBO slides to Grosse & Ba, University of Toronto.)

4.5.1 Derivation of the evidence lower bound

Fix a data point $\boldsymbol{x}$, and let $q(\boldsymbol{z})$ be *any* distribution over codes (in practice $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$, the encoder's output). Then:

Step 1 — marginalize over the latent.

$$\log p(\boldsymbol{x}) \;=\; \log \int p(\boldsymbol{z})\, p(\boldsymbol{x} \mid \boldsymbol{z})\, d\boldsymbol{z}$$

Why this step is legit: the law of total probability; $\boldsymbol{z}$ is marginalized out of the joint $p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{z})p(\boldsymbol{x} \mid \boldsymbol{z})$. Pure definition, nothing approximated yet.

Step 2 — introduce the variational distribution q.

$$= \log \int q(\boldsymbol{z})\, \frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\, p(\boldsymbol{x} \mid \boldsymbol{z})\, d\boldsymbol{z}$$

Why this step is legit: we multiplied and divided by $q(\boldsymbol{z})$ wherever $q(\boldsymbol{z}) > 0$; the integrand is unchanged. This is the *importance-sampling* maneuver: the integral now reads as an *expectation under* q—and expectations are things we can estimate by sampling. The freedom to choose q is the "variational" in variational Bayes: we will eventually let $q = q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ and learn the choice. Nothing has been approximated yet; the equality is exact.

Step 3 — push the $\log$ inside via Jensen.

$$\geq \int q(\boldsymbol{z})\, \log\!\left[\frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\, p(\boldsymbol{x} \mid \boldsymbol{z})\right] d\boldsymbol{z}$$

Why this step is legit: this is where the inequality is born. The integral in Step 2 is $\log \mathbb{E}_q[\cdot]$, and concavity of log gives $\log \mathbb{E}_q[\cdot] \geq \mathbb{E}_q[\log(\cdot)]$ by Equation (4.4). The bound is the price of tractability—and the gap that opens here determines how far below $\log p(\boldsymbol{x})$ we land.

Step 4 — split the $\log$ of a product into recognizable terms.

$$= \underbrace{\mathbb{E}_{q(\boldsymbol{z})}\big[\log p(\boldsymbol{x} \mid \boldsymbol{z})\big]}_{\text{reconstruction}} - \underbrace{D_{\mathrm{KL}}\big(q(\boldsymbol{z}) \,\|\, p(\boldsymbol{z})\big)}_{\text{prior matching}} =: \mathrm{ELBO}(\boldsymbol{x}). \tag{4.5}$$

Why this step is legit: $\log(ab/c) = \log a + \log b - \log c$, and $\mathbb{E}_q[\log p(\boldsymbol{z}) - \log q(\boldsymbol{z})] = -D_{\mathrm{KL}}(q\|p)$ by the definition of KL divergence (Chapter 2). Restoring the parameter subscripts and conditioning on the observed point gives the objective used in code,

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) \;=\; -\,D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p(\boldsymbol{z})\big) \;+\; \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\big], \tag{4.6}$$

where D_{KL} is the Kullback–Leibler divergence, the reconstruction term is an expectation over codes sampled from the encoder, and all quantities are tractable or Monte-Carlo estimable. Training *maximizes* Equation (4.6); most frameworks minimize, so the implementation carries a minus sign.

Key Concept — The ELBO and its gap

The *evidence lower bound* $\mathrm{ELBO}(\boldsymbol{x}) \le \log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ is tight exactly when the approximate posterior equals the true one:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \;=\; \mathrm{ELBO}(\boldsymbol{x}) \;+\; D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})\big).$$

The slack is therefore not an oracle mystery: it is precisely the price, in nats, of our encoder not yet being a perfect Bayesian inverter of the decoder. Maximizing the ELBO in $\boldsymbol{\phi}$ literally drives this D_{KL} down; maximizing it in $\boldsymbol{\theta}$ lifts the bound from below. Joint training improves both at once—and it is impossible to say the model truly "learned $\log p(\boldsymbol{x})$" any better than the encoder could keep up.

Simplified intuition vs. more precise view

Simplified intuition: the decoder learns $p(\boldsymbol{x} \mid \boldsymbol{z})$—given a code, paint the picture. **More precise view:** the model never optimizes $\log p(\boldsymbol{x})$ directly; it optimizes the ELBO, a *lower bound* on it. Maximizing the bound pushes the likelihood up only indirectly, and the gap between the two is exactly $D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \| p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}))$—the distance from our approximate posterior to the true, intractable one. Since KL is non-negative, $\mathrm{ELBO}(\boldsymbol{x}) \le \log p(\boldsymbol{x})$ always; the bound is *tight* iff $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ equals the true posterior. Training improves the model ($\boldsymbol{\theta}$) and the approximation ($\boldsymbol{\phi}$) simultaneously, and only the former buys real likelihood. When you read "the VAE learned $p(\boldsymbol{x})$," mentally add "up to whatever slack the encoder left in the bound."

4.6 Making the ELBO Computable

Equation (4.6) contains two fresh problems: an expectation we cannot integrate, and a sampling operation gradients cannot pass through. Three pieces close the loop: a Gaussian decoder that turns the first ELBO term into squared error, a reparameterization that makes sampling differentiable, and a closed-form KL.

4.6.1 The reconstruction term is disguised squared error

Choose a Gaussian observation model for the decoder, $p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{x};\, G_{\boldsymbol{\theta}}(\boldsymbol{z}),\, \eta \mathbf{I})$, where $G_{\boldsymbol{\theta}}(\boldsymbol{z})$ is the decoder network's output (the mean image) and $\eta > 0$ a fixed observation-noise variance. Its log-density at a D-dimensional point is

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) \;=\; \log\!\left[\frac{1}{(2\pi\eta)^{D/2}} \exp\!\Big(-\tfrac{1}{2\eta}\|\boldsymbol{x} - G_{\boldsymbol{\theta}}(\boldsymbol{z})\|^2\Big)\right] \;=\; -\frac{1}{2\eta}\|\boldsymbol{x} - G_{\boldsymbol{\theta}}(\boldsymbol{z})\|^2 + \text{const}, \tag{4.7}$$

where "const" collects everything that does not depend on the network. (The expansion is just $\log(ab) = \log a + \log b$ and $\log e^u = u$, applied to the Gaussian density — the centre column of the Grosse & Ba slide.)

So with η fixed, **maximising the reconstruction term is exactly minimising squared error**. The plain autoencoder loss of Equation (4.2) was a log-likelihood all along.

Two consequences:

- The best decoder outputs the *average* of all images that fit the code, $\mathbb{E}[\boldsymbol{x} \mid \boldsymbol{z}]$ — the source of the blur (Section 4.6.4).
- η quietly changes the balance between the ELBO's two terms (see the worked example that follows).

Worked Example — Gaussian negative log-likelihood collapses to MSE

One pixel, decoder mean $\mu = G_{\boldsymbol{\theta}}(\boldsymbol{z}) = 0.30$, noise variance fixed at $\eta = 0.25$, observed value $x = 0.60$.

$$\begin{aligned}\log \mathcal{N}(x; \mu, \eta) &= -\tfrac{1}{2}\log(2\pi\eta) - \tfrac{(x-\mu)^2}{2\eta} \\ &= \underbrace{-0.2258}_{\text{const}} - \frac{(0.60 - 0.30)^2}{0.5} = -0.2258 - 0.18 = -0.4058.\end{aligned}$$

Now move the observation to $x = 0.90$: only the second term changes, $-(0.60)^2/0.5 = -0.72$, giving $\log p = -0.9458$. The drop in log-likelihood, 0.54, is exactly the increase in squared error, $0.36 - 0.09 = 0.27$, times the fixed factor $1/(2\eta) = 2$. Parameter gradients see none of the constant and none of η: maximizing Gaussian likelihood and minimizing MSE are the same optimization problem whenever η is held fixed. What η *does* set is the exchange rate between the two ELBO terms in Equation (4.6): smaller η weights reconstruction more heavily against the KL.

4.6.2 The reparameterization trick: backprop through a sample

In plain words — The problem and the fix in two sentences

Problem: the VAE's loss needs a *random* code $\boldsymbol{z}$, and you cannot backpropagate through "pick a random number". **Fix:** pick the random number *first*, from a fixed bag that has nothing to learn, and then *compute* $\boldsymbol{z}$ from it with ordinary arithmetic — shift by $\boldsymbol{\mu}_\phi$, stretch by $\boldsymbol{\sigma}_\phi$. Arithmetic can be backpropagated through.

Context. VAEs — or *any* model that optimises an average over a distribution that itself depends on learnable parameters. Throughout, $\boldsymbol{\phi}$ are the encoder's parameters and $\boldsymbol{\theta}$ the decoder's.

1. The problem. The ELBO (Equation (4.6)) has a term like

$$\mathcal{L}(\boldsymbol{\phi}) \;=\; \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}\mid\boldsymbol{z})\big]. \tag{4.8}$$

We need $\nabla_{\boldsymbol{\phi}}\mathcal{L}$. But look where $\boldsymbol{\phi}$ sits: in the *distribution we sample from*, not only in the thing being averaged.

- Sampling $\boldsymbol{z}\sim\mathcal{N}(\boldsymbol{\mu}_\phi,\boldsymbol{\sigma}_\phi^2)$ is a *stochastic node*. Nudging $\boldsymbol{\mu}_\phi$ changes which values are *likely*, but an already-drawn value such as $z=0.73$ has no derivative with respect to $\boldsymbol{\mu}_\phi$. So backprop has *no path* from $\boldsymbol{z}$ back to $\boldsymbol{\mu}_\phi,\boldsymbol{\sigma}_\phi$.
- We also cannot simply push the gradient inside the average:

  $$\nabla_{\boldsymbol{\phi}}\,\mathbb{E}_{q_{\boldsymbol{\phi}}}[f] \;\neq\; \mathbb{E}_{q_{\boldsymbol{\phi}}}[\nabla_{\boldsymbol{\phi}} f].$$

  The right-hand side is actually *zero*, because once $\boldsymbol{z}$ is a fixed number, $f(\boldsymbol{z})$ contains no $\boldsymbol{\phi}$ at all.

Tiny example. Let $z\sim\mathcal{N}(\mu,\sigma^2)$ and $f(z)=z^2$. Then $\mathbb{E}[f]=\mu^2+\sigma^2$, so the true gradient is $\partial\mathbb{E}[f]/\partial\mu=2\mu$ — but "differentiate inside" gives 0. The gradient lives in how the *distribution* moves, which is exactly what a single sample hides.

2. The trick. Write the sample as a *deterministic function of* $\boldsymbol{\phi}$ applied to *parameter-free noise* (Kingma and Welling, 2014; Rezende et al., 2014; Doersch, 2016):

$$\boldsymbol{z} \;=\; g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}) \;=\; \boldsymbol{\mu}_\phi \;+\; \boldsymbol{\sigma}_\phi \odot \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I}), \tag{4.9}$$

where $\odot$ is element-wise multiplication. This $\boldsymbol{z}$ has exactly the same distribution as before, $\mathcal{N}(\boldsymbol{\mu}_\phi,\operatorname{diag}\boldsymbol{\sigma}_\phi^2)$: a standard Gaussian, stretched by $\boldsymbol{\sigma}_\phi$ and shifted by $\boldsymbol{\mu}_\phi$. But now the

expectation is over $\boldsymbol{\epsilon}$, whose distribution does *not* depend on $\boldsymbol{\phi}$ — so the gradient *may* move inside:

$$\nabla_{\boldsymbol{\phi}}\, \mathbb{E}_{\boldsymbol{\epsilon}}\big[f(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}))\big] \;=\; \mathbb{E}_{\boldsymbol{\epsilon}}\big[\nabla_{\boldsymbol{\phi}} f(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}))\big]. \tag{4.10}$$

Inside, it is the ordinary chain rule. Since $\partial \boldsymbol{z}/\partial \boldsymbol{\mu}_{\boldsymbol{\phi}} = \mathbf{1}$ and $\partial \boldsymbol{z}/\partial \boldsymbol{\sigma}_{\boldsymbol{\phi}} = \boldsymbol{\epsilon}$,

$$\frac{\partial f}{\partial \boldsymbol{\mu}_{\boldsymbol{\phi}}} = \nabla_{\boldsymbol{z}} f, \qquad \frac{\partial f}{\partial \boldsymbol{\sigma}_{\boldsymbol{\phi}}} = \nabla_{\boldsymbol{z}} f \odot \boldsymbol{\epsilon}.$$

The randomness has become an *input* to the graph, and $\boldsymbol{z}$ is differentiable with respect to $\boldsymbol{\mu}_{\boldsymbol{\phi}}$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}$. Standard backprop works, and a Monte-Carlo estimate — usually with a *single* sample of $\boldsymbol{\epsilon}$ per example — gives the gradient. Figure 4.7 draws the computation before and after.

Everyday analogy — Rolling a standard die

You want a random number of steps near μ, with spread σ, and you want to *learn* good values of μ and σ. *Before:* you ask a friend to "pick something around μ". The result is random, and you cannot tell how it would have changed if μ had been slightly bigger. *After:* you roll a *standard* die yourself (that is ϵ), then compute $\mu + \sigma \cdot \epsilon$. Now if μ grows by 0.1, the result grows by exactly 0.1; if σ grows by 0.1, it grows by $0.1\,\epsilon$. The die never changes — only the arithmetic after it does, and arithmetic has derivatives. **Where the analogy breaks:** a die is discrete, but the trick needs $g_{\boldsymbol{\phi}}$ to be *smooth* in $\boldsymbol{\phi}$ — which is exactly why it fails for discrete latents (see the caveat below).

Step by step — One VAE training step with the trick

Step 1. The encoder reads $\boldsymbol{x}$ and outputs $\boldsymbol{\mu}_{\boldsymbol{\phi}}$ and $\log \boldsymbol{\sigma}^2_{\boldsymbol{\phi}}$.

Step 2. Draw noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ — the only random step, and it has no parameters.

Step 3. Compute $\boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}} \odot \boldsymbol{\epsilon}$: plain arithmetic.

Step 4. The decoder turns $\boldsymbol{z}$ into $\hat{\boldsymbol{x}}$; compute the loss.

Step 5. Backpropagate: loss → decoder → $\boldsymbol{z}$ → $(\boldsymbol{\mu}_{\boldsymbol{\phi}}, \boldsymbol{\sigma}_{\boldsymbol{\phi}})$ → encoder. $\boldsymbol{\epsilon}$ is treated as a constant.

3. Why not the alternative? There is a general-purpose way to get the gradient without the trick, the *score-function* (REINFORCE) estimator:

$$\nabla_{\boldsymbol{\phi}}\, \mathbb{E}_{q_{\boldsymbol{\phi}}}[f(\boldsymbol{z})] \;=\; \mathbb{E}_{q_{\boldsymbol{\phi}}}\big[f(\boldsymbol{z})\, \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\boldsymbol{z})\big]. \tag{4.11}$$

It also works and is **unbiased** — for *any* q and *any* f, even discrete $\boldsymbol{z}$ and non-differentiable f. But it has **very high variance**, and the formula shows why:

- it uses only the *scalar value* $f(\boldsymbol{z})$ — here a whole image's reconstruction error, a large and highly variable number — multiplied by a random direction;

- it never uses the *gradient* $\nabla_{\boldsymbol{z}} f$, i.e. *which way* f gets better.

Reparameterisation *does* exploit $\nabla_{\boldsymbol{z}} f$, so it gives **much lower-variance** gradients — which means faster, more stable training. That is why Equation (4.9) won for VAEs.

A practical detail. Networks do not output $\boldsymbol{\sigma}$ directly, because a standard deviation must be positive and a network output can be anything. They output $\log \boldsymbol{\sigma}^2$ instead, which may be any real number, and recover $\boldsymbol{\sigma} = \exp(\frac{1}{2} \log \boldsymbol{\sigma}^2)$ — positive automatically.

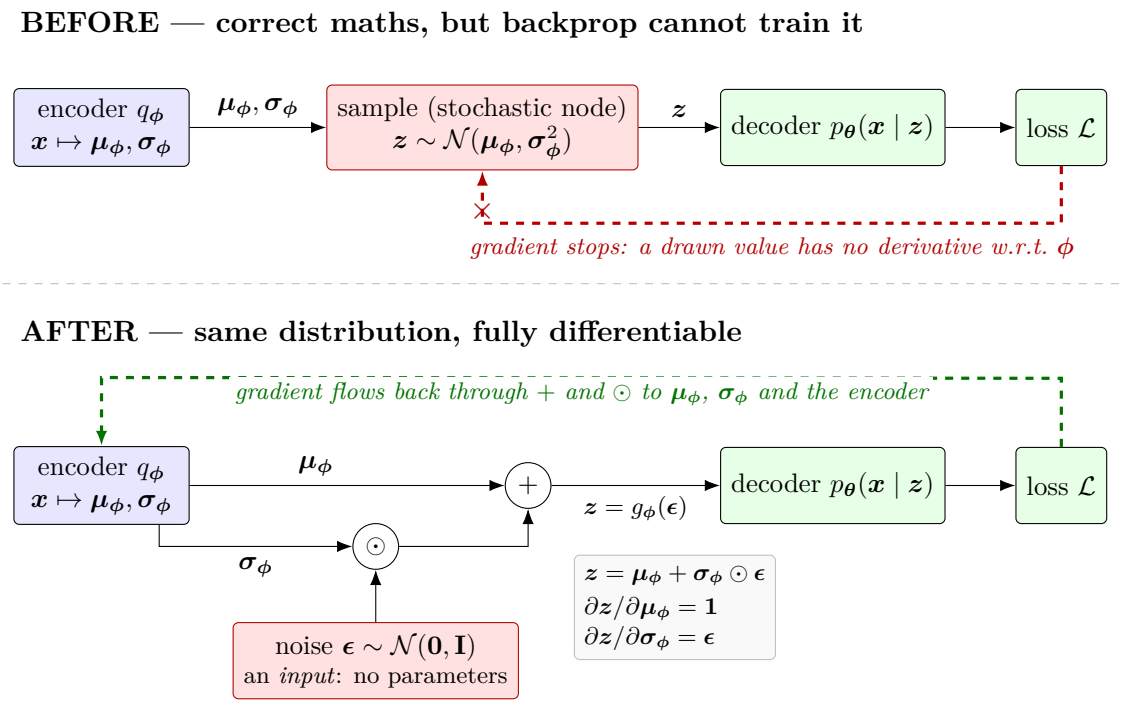


Figure 4.7. The reparameterization trick as two computation graphs (cf. Doersch (2016), Figure 4). *Top:* $\boldsymbol{z}$ is sampled *inside* the network (red box, a stochastic node). The gradient from the loss reaches the sampler and stops, because a drawn value has no derivative with respect to $\boldsymbol{\phi}$. *Bottom:* Equation (4.9) rewrites the same sample as $\boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}} \odot \boldsymbol{\epsilon}$. The randomness now enters as an *input* $\boldsymbol{\epsilon}$ that needs no gradient, while $\boldsymbol{\mu}_{\boldsymbol{\phi}}$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}$ reach the decoder through ordinary arithmetic ($\odot$, $+$). Both graphs produce samples with the same distribution; only the bottom one can be trained by backpropagation.

How to read it: Follow the dashed arrow in each row: red ends at the sampler with a ×; green runs all the way back to the encoder.

Worked Example — Both estimators on a problem you can solve by hand

Take one latent, $z \sim \mathcal{N}(\mu, \sigma^2)$ with $\mu = 1$, $\sigma = 1$, and $f(z) = z^2$. Exact answer: $\mathbb{E}[f] = \mu^2 + \sigma^2$, so the true gradients are $\partial/\partial\mu = 2\mu = 2$ and $\partial/\partial\sigma = 2\sigma = 2$. Figure 4.8 draws what follows.

One sample. Suppose the noise comes out as $\epsilon = 0.5$, so $z = 1 + 1 \cdot 0.5 = 1.5$.

- *Reparameterisation:* $f'(z) = 2z = 3$. Gradient for μ: $f'(z) = 3$. Gradient for σ: $f'(z)\,\epsilon = 3 \times 0.5 = 1.5$.
- *Score function:* for a Gaussian, $\partial_\mu \log q = (z-\mu)/\sigma^2 = \epsilon$ and $\partial_\sigma \log q = (\epsilon^2 - 1)/\sigma$. Gradient for μ: $f(z)\,\epsilon = 2.25 \times 0.5 = 1.125$. Gradient for σ: $2.25 \times (0.25 - 1) = -1.6875$ — the wrong sign!

A single sample of either is off; what matters is how much they scatter.

Many samples. Both average to the true value 2 (both are unbiased). Their *variances* are, exactly (using the Gaussian moments $\mathbb{E}[\epsilon^2] = 1$, $\mathbb{E}[\epsilon^4] = 3$, $\mathbb{E}[\epsilon^6] = 15$, $\mathbb{E}[\epsilon^8] = 105$, and checked by simulating two million samples):

	gradient for μ	**gradient for σ**
reparameterisation: $2z$, $2z\epsilon$	variance 4	variance 12
score function: $z^2\epsilon$, $z^2(\epsilon^2 - 1)$	variance 30	variance 136

The score-function estimate is about 7.5× noisier for μ and 11× noisier for σ — on a one-dimensional toy. To match the reparameterised estimate's accuracy you would need that many times more samples. With thousands of latent dimensions and a whole image's reconstruction error as f, the gap gets far worse.

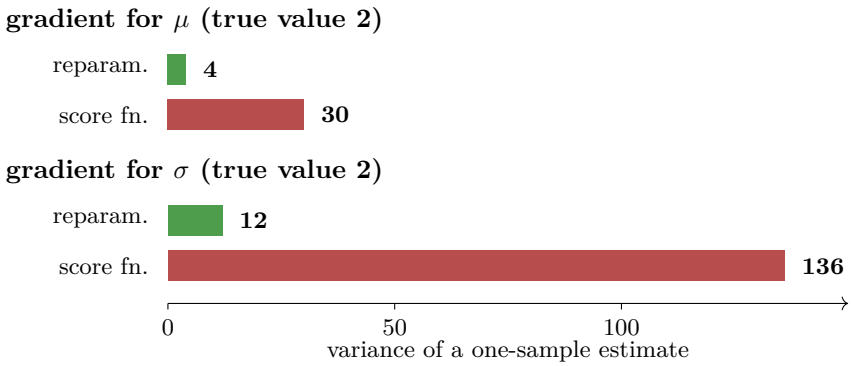


Figure 4.8. Both estimators are unbiased, but they are not equally useful. For $z \sim \mathcal{N}(1, 1)$ and $f(z) = z^2$, a one-sample *score-function* (REINFORCE) gradient scatters 7.5× (for μ) to 11× (for σ) more than a *reparameterised* one. The reason: the reparameterised gradient uses the slope $f'(z)$; the score-function gradient only uses the value $f(z)$ times a random direction.

How to read it: Compare the lengths of each red bar with the green bar just above it — same true gradient, same number of samples, very different noise.

4. Summary. The comparison exams ask for:

	Score function (REINFORCE)	Reparameterization
Gradient estimate	$f(\boldsymbol{z})\,\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\boldsymbol{z})$	$\nabla_{\boldsymbol{\phi}} f(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}))$
Unbiased	✓	✓
Variance	High	Low
Uses	only the value $f(\boldsymbol{z})$	the gradient $\nabla_{\boldsymbol{z}} f$
Needs differentiable f	×	✓
Needs reparameterizable q	×	✓ (e.g. Gaussian, location–scale families)
Works for discrete $\boldsymbol{z}$	✓	× (not directly)
Typical home	reinforcement learning, discrete choices	VAEs, continuous latents

5. Caveat: discrete latents. The trick does not work directly if $\boldsymbol{z}$ is discrete (a category, a token, a yes/no), because there is no differentiable $g_{\boldsymbol{\phi}}$: a tiny change in $\boldsymbol{\phi}$ either leaves the choice unchanged (gradient zero) or flips it (a jump). The usual workarounds:

- **Gumbel-Softmax / Concrete relaxation** — replace the hard choice with a soft, temperature-controlled softmax that *is* reparameterisable, and sharpen it during training;
- **straight-through estimator** — use the hard choice in the forward pass, but pretend it was the identity in the backward pass;
- **REINFORCE with a baseline** — use Equation (4.11), but subtract a running average of f first to cut the variance.

Common Confusion

- **"The trick removes the randomness."** No. $\boldsymbol{z}$ is exactly as random as before. The randomness is *moved*: from a node inside the network to an input $\boldsymbol{\epsilon}$ that backprop treats as a constant.
- **"The trick changes the model."** No. $\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ has exactly the distribution $\mathcal{N}(\boldsymbol{\mu}, \operatorname{diag} \boldsymbol{\sigma}^2)$. Only the *computation graph* changes.
- **"The score-function estimator is biased."** No — both are unbiased. The difference is *variance*.
- **"The KL term needs the trick too."** Not for Gaussians: it has the closed form of Section 4.6.3. The trick is needed for the *reconstruction* term, which has no closed form.

Key takeaway — If an exam asks "what is the reparameterisation trick and why is it needed?"

- **Need:** the ELBO contains $\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\cdot]$, where $\boldsymbol{\phi}$ sits in the *sampling distribution*; sampling has no gradient, and $\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}}[f] \neq \mathbb{E}_{q_{\boldsymbol{\phi}}}[\nabla_{\boldsymbol{\phi}} f]$.
- **Trick:** write $\boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}} \odot \boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$; the expectation is now over parameter-free noise, so the gradient moves inside and ordinary backprop works.

- **Why not REINFORCE:** it is also unbiased but has much higher variance, because it uses only f's value, not its gradient.
- **Limit:** needs continuous, reparameterisable q; for discrete latents use Gumbel-Softmax, straight-through, or REINFORCE with a baseline.

4.6.3 The KL term in closed form

The second ELBO term compares two *diagonal Gaussians*—the encoder's $\mathcal{N}(\boldsymbol{\mu}, \operatorname{diag}\boldsymbol{\sigma}^2)$ against the prior $\mathcal{N}(\mathbf{0}, \mathbf{I})$—and for Gaussians the KL divergence has a closed form, no sampling needed:

$$-D_{\mathrm{KL}}\big(\mathcal{N}(\boldsymbol{\mu}, \operatorname{diag}\boldsymbol{\sigma}^2)\|\mathcal{N}(\mathbf{0}, \mathbf{I})\big) \;=\; \frac{1}{2}\sum_{j=1}^{J}\Big(1 + \log\sigma_j^2 - \mu_j^2 - \sigma_j^2\Big), \tag{4.12}$$

where J is the number of latent dimensions and μ_j, σ_j^2 are the encoder's mean and variance for dimension j.

What each piece does. Each dimension contributes a small tug of war:

- the $-\mu_j^2$ piece pulls the **mean** towards 0 (the centre of the prior);
- the $1 + \log\sigma_j^2 - \sigma_j^2$ piece pulls the **spread** towards 1 (the width of the prior). It is largest, exactly 0, at $\sigma_j = 1$.

A few values, per dimension: $(\mu, \sigma) = (0, 1)$ costs 0; $(1, 1)$ costs 0.5; $(0, 0.5)$ costs 0.32; $(0, 2)$ costs 0.81. Moving the mean or making the code too sharp or too fuzzy all cost something.

Where the formula comes from. Start from the definition and write out both Gaussian log-densities for one dimension:

$$D_{\mathrm{KL}}(q\|p) = \mathbb{E}_q\big[\log q(\boldsymbol{z}) - \log p(\boldsymbol{z})\big] = -\tfrac{1}{2}\log\sigma_j^2 - \frac{\mathbb{E}_q[(z_j - \mu_j)^2]}{2\sigma_j^2} + \frac{\mathbb{E}_q[z_j^2]}{2}$$

(the $\log 2\pi$ constants cancel). The two averages are σ_j^2 and $\sigma_j^2 + \mu_j^2$, using $\mathbb{E}[Z^2] = \mathrm{Var}(Z) + \mathbb{E}[Z]^2$. Substituting and tidying gives $-\frac{1}{2}(1 + \log\sigma_j^2 - \mu_j^2 - \sigma_j^2)$ — the line every implementation writes as `0.5 * sum(1 + logvar - mu**2 - exp(logvar))`, with the sign flipped for the loss.

Worked Example — The KL term for one concrete (μ, σ)

Take one latent dimension ($J = 1$) with encoder outputs $\mu = 2$, $\sigma^2 = 4$ (so $\sigma = 2$). Plug into the summand of Equation (4.12):

$$\tfrac{1}{2}(1 + \log 4 - 2^2 - 4) = \tfrac{1}{2}(1 + 1.3863 - 4 - 4) = \tfrac{1}{2}(-5.6137) \approx -2.807,$$

so the encoder *loses* 2.807 nats of ELBO per sample in this dimension for placing a wide posterior at $\mu = 2$. Repeat for a lazier answer, $\mu = 2$ with $\sigma^2 = 1$: $\frac{1}{2}(1+0-4-1) = -2.000$ nats. And the prior's own answer, $\mu = 0$, $\sigma^2 = 1$: $\frac{1}{2}(1+0-0-1) = 0$—no penalty at all, but also no information about $\boldsymbol{x}$ encoded. The VAE's posterior never sits at either extreme; it pays exactly as much KL as the reconstruction term can profitably spend.

4.6.4 The practical training objective

Assemble the pieces. One data point $\boldsymbol{x}^{(i)}$; the encoder emits $\boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{(i)}$; draw L Monte-Carlo samples $\boldsymbol{z}^{(i,l)} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(l)}$ with $\boldsymbol{\epsilon}^{(l)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (in practice $L = 1$ per step already works and is what the lecture uses) (Kingma and Welling, 2014):

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}) \;\simeq\; \frac{1}{2} \sum_{j=1}^{J} \left(1 + \log\left(\sigma_j^{(i)}\right)^2 - \left(\mu_j^{(i)}\right)^2 - \left(\sigma_j^{(i)}\right)^2\right) \;+\; \frac{1}{L} \sum_{l=1}^{L} \log p_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)} \mid \boldsymbol{z}^{(i,l)}\right), \tag{4.13}$$

where the first term is Equation (4.12) evaluated exactly and the second estimates the reconstruction expectation; under the Gaussian decoder of Equation (4.7) the last term is implemented as $-\frac{1}{2\eta}\|\boldsymbol{x}^{(i)} - G_{\boldsymbol{\theta}}(\boldsymbol{z}^{(i,l)})\|_2^2$ plus a constant (with $\eta = 1$ in many codebases). Every quantity is a differentiable function of $(\boldsymbol{\theta}, \boldsymbol{\phi})$. Figure 4.9 shows how the two terms pull the same baton in opposite directions.

```
h       = encoder_backbone(x)
mu      = head_mu(h)                # E[z | x]
logvar  = head_logvar(h)            # log sigma^2 -- unbounded head
std     = exp(0.5 * logvar)
eps     = randn_like(std)           # ~ N(0, I), off the gradient path
z       = mu + std * eps            # Eq. (eq:ch04-reparam)
x_hat   = decoder(z)
kl      = -0.5 * sum(1 + logvar - mu**2 - exp(logvar))  # Eq. (eq:ch04-gauss-kl)
recon   = 0.5 * ((x - x_hat) ** 2).sum() / eta          # Eq. (eq:ch04-gauss-decoder)
loss    = recon + kl                # minimize == maximize negated ELBO
loss.backward()
```

Listing 4.2. The per-sample ELBO computation. The encoder head emits `mu` and `logvar`; the reparameterized sample is built on one line from an external noise draw; the two loss terms correspond line-for-line to Equations (4.12) and (4.7).

The listing makes one design fact unusually legible that the equations obscure at first reading: the ELBO's two terms depend on the encoder *through different gates*. The KL term sees the encoder's outputs $\boldsymbol{\mu}, \boldsymbol{\sigma}^2$ directly; the reconstruction term sees them only through one sampled $\boldsymbol{z}$. That asymmetry is the source of the failure mode we meet in Figure 4.9.

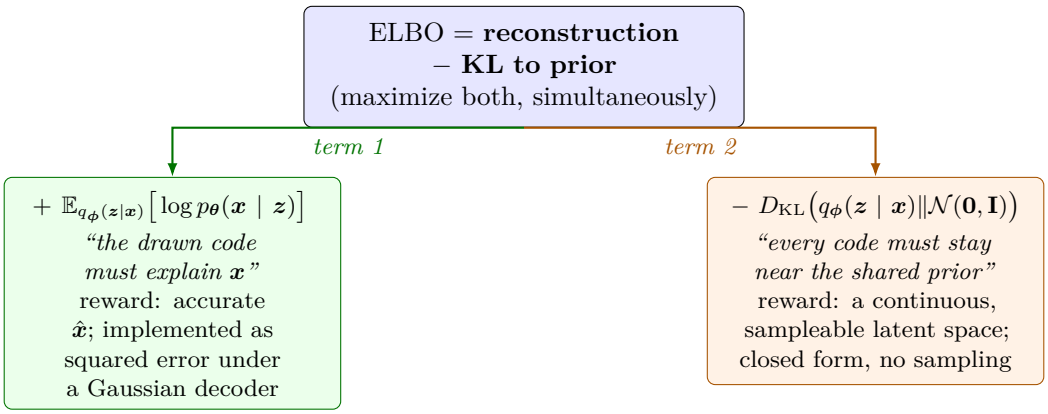


Figure 4.9. Decomposition of the VAE objective (4.6) into its two opposing terms. The reconstruction pointer (left) rewards codes from which $\boldsymbol{x}$ can be rebuilt—if it wins outright, the VAE degenerates into a memorizing AE with an unstructured latent space. The KL pointer (right) taxes any posterior that drifts from the standard-normal prior—if it wins outright, the encoder outputs the prior for every $\boldsymbol{x}$ and encodes nothing (posterior collapse). Useful VAEs live at the negotiated equilibrium.

Failure Modes

Failure modes of the VAE. (1) Blurry samples. The Gaussian decoder of Equation (4.7) scores reconstructions by pixel-wise squared error. Facing irreducible ambiguity about fine detail, the likelihood-optimal answer is the *conditional mean over plausible details*—which is a blur. The mechanism is one line of optimization: if two equally plausible reconstructions $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$ differ only in high-frequency content, mean-squared error is minimized at their midpoint $G_{\boldsymbol{\theta}}(\boldsymbol{z}) = (\boldsymbol{x}_a + \boldsymbol{x}_b)/2$, not at either one. The Frey-face and MNIST generation grids shown in the lecture (Figures 4.11 and 4.12) are smooth but visibly soft. Adversarial losses (Chapter 6) and autoregressive or diffusion decoders (Chapter 7) are the standard repairs; modern image generators also frequently run their diffusion *inside* a VAE's latent space, so the two architectures end up stacked, not competing. **(2) Prior hole.** Training pushes the average posterior $\mathbb{E}_{\boldsymbol{x}}[q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})]$ only weakly toward the prior $p(\boldsymbol{z})$; the ELBO controls the KL *per data point*, not the shape of the aggregate cloud of codes. The result can be a dimpled, disconnected latent space with regions—"holes"—that the prior says are likely but no training point ever used. Sampling $\boldsymbol{z}$ there decodes into garbage: the latent manifold is not guaranteed to be filled, only restricted. **(3) Posterior collapse.** A strong decoder (e.g. a pixel-autoregressive one) can model $\boldsymbol{x}$ while ignoring $\boldsymbol{z}$ entirely; the KL term gladly shrinks to zero by copying the prior and the latent channel carries no information. Annealing the KL weight or using weak decoders mitigates it.

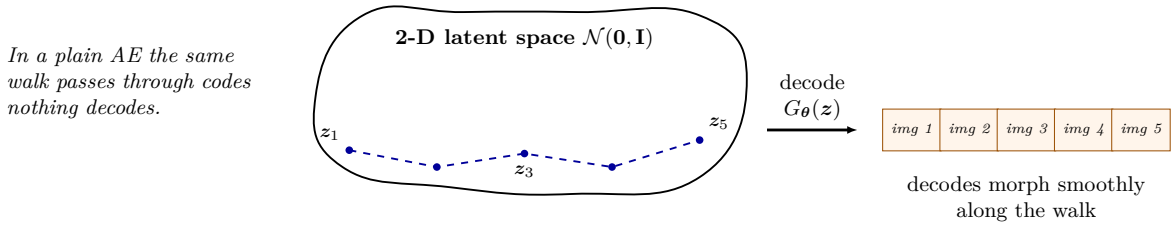


Figure 4.10. Walking a grid through the VAE's two-dimensional latent space. Because every training posterior was dragged toward the shared prior, neighboring codes decode to neighboring images: decoding the sequence $\boldsymbol{z}_1 \to \cdots \to \boldsymbol{z}_5$ morphs content continuously (pose, expression, digit identity) rather than jumping between memorized exemplars or hitting undecodable holes. This is the abstract content of the lecture's two Kingma-and-Welling grids, shown as placeholders in Figures 4.11 and 4.12.

4.6.5 What a trained latent space buys: smooth manifolds

What the latent grids show — and what they don't. Figure 4.10 shows the payoff of the KL term: walk a straight line through the latent space and the decoded image changes smoothly. The lecture uses the original 2014 grids: faces changing smoothly in expression and pose (Figure 4.11) and digits morphing across a 20×20 grid (Figure 4.12) (Kingma and Welling, 2014).

- **They do show** that nearby codes decode to similar images, so a straight walk never falls off a cliff.
- **They do not show** that the directions *mean* anything. Nothing in the training said "this axis is pose"; we read that into the picture afterwards.

Whether directions can be made meaningful is the *disentanglement* question of the Research Frontier box below.

How the grids were made. The cells are *not* evenly spaced in the latent space. Each axis runs through 20 values chosen so that every cell is equally likely under the Gaussian prior (they are its quantiles, via the inverse CDF Φ^{-1}). So the prior, not the data, decides where the tour stops.

Notice both the success and the failure in the same picture: the faces and digits change smoothly everywhere, but fine detail — individual hairs, the ends of pen strokes — is soft throughout. That softness is exactly what the squared-error decoder of Section 4.6.4 predicts.

Try the same sampling protocol with a plain deterministic autoencoder and the trick collapses. A draw $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ lands with high probability in a latent region no training posterior ever occupied—one of the holes the KL term exists to fill—and the decoder, never queried there, outputs something unrelated to any image. This is the worked counterpart to the AE's third failure mode from Section 4.2: an unstructured latent space is one you can

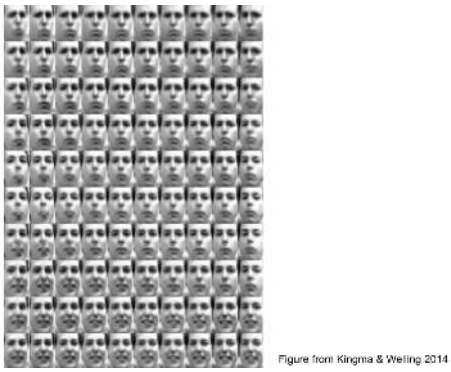


Figure 4.11. Left-to-right, top-to-bottom samples of the Frey face dataset decoded from a grid of codes in the 2-D latent space of a trained VAE (evenly spaced *quantiles* of the Gaussian prior, as explained in the text) (figure from Kingma and Welling (2014), as shown on the Lecture 03 slide). Moving horizontally sweeps facial expression; moving vertically sweeps pose. Every interior code decodes to a plausible face—latent-space continuity made visible.

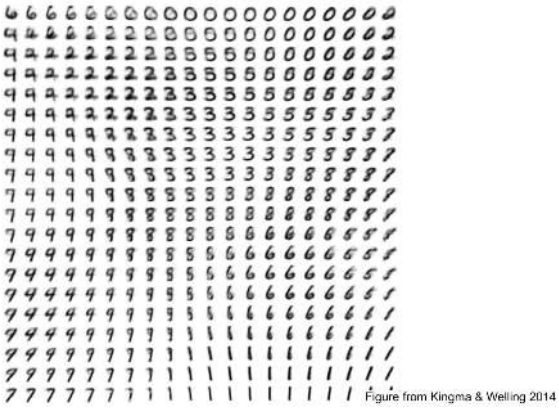


Figure 4.12. Samples from a VAE with 2-D latent space trained on handwritten digits (figure from Kingma and Welling (2014), as shown on the Lecture 03 slide), decoding a 20×20 grid of latent quantiles. Digit identity morphs across the axes (e.g. $9 \to 4 \to 2 \to 3 \to 5 \to 0$ along one direction, $7 \to 1$ along another): the KL-to-prior term has organized a semantic atlas from pixels alone.

interpolate *near data* in but cannot *generate from the prior* out of. The VAE is the minimal repair that makes the prior the sampling rule rather than a fiction.

Research Frontier

β-VAE and the entanglement problem. Looking again at Equation (4.6): what if the two terms should not trade at 1:1? The β-VAE multiplies the KL term by $\beta > 1$, making every bit of information in the latent code more expensive—higher pressure to explain $\boldsymbol{x}$ with a few independent factors. Practitioners report that strong KL pressure teases apart generative factors (rotation, width, lighting) into distinct latent dimensions—*disentanglement*—but pays with blurrier reconstructions, because Equation (4.6) trades

reconstruction against compression. The conceptual lesson reaches beyond VAEs: *an objective's exchange rates are design decisions*, and the shape of a learned representation inherits them. Disentanglement without identifiable supervision remains unresolved—Chapter 15 returns to it when weighing competing explanations of why any of this works at all.

Paper Analysis: Kingma & Welling (2014), Auto-Encoding Variational Bayes

Problem. Deep directed latent-variable models were trainable only with expensive per-datapoint inference (MCMC); there was no scalable way to backpropagate through the posterior of a continuous latent variable. **Prior work.** Wake-sleep algorithms and Helmholtz machines of the 1990s already paired generative and recognition networks; mean-field variational inference was exact for conjugate models but not for neural decoders. **Insight.** Choose a Gaussian recognition network $q_{\phi}(\boldsymbol{z} \mid \boldsymbol{x})$, then push the randomness out of the gradient path via $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ so ordinary SGD optimizes a variational bound *jointly* over $(\boldsymbol{\theta}, \boldsymbol{\phi})$. **Method.** Per-minibatch ELBO (4.13): closed-form Gaussian KL against a standard-normal prior plus a Monte-Carlo estimate of the decoder log-likelihood with one sample. **Experiments.** Generative modeling on MNIST and Frey faces; 2-D latent grids demonstrate smooth learned manifolds (reproduced in the lecture, Figures 4.11–4.12). **Results.** Competitive log-likelihoods with a training cost comparable to a plain deterministic autoencoder. **Limitations.** Blurry pixel-L2 samples; posterior collapse risk with expressive decoders; prior holes unaddressed (Section 4.6.4). **Later work.** Doersch (2016) popularized the computation-graph view; Kingma and Welling (2019) wrote the canonical tutorial; normalizing-flow posteriors (Chapter 6) and VAE–diffusion hybrids (Chapter 7) descend from it. **What to remember.** Variational Bayes + reparameterization ⇒ generative models trainable by plain backprop; the ELBO's two terms, not an architecture, define a VAE.

4.7 Why Does the VAE Actually Work?

Strip away the machinery and three ideas do all the work.

1. A bottleneck with a price. Reconstruction alone lets the codes go anywhere. The prior gives them a preferred shape, and the KL term charges the encoder for straying from it. The encoder will only pay that charge for code dimensions that genuinely improve the reconstruction. This is the information-bottleneck trade-off of Equation (4.1), made trainable.

2. A bound you can compute. One use of Jensen's inequality turns an impossible integral into two manageable terms (4.5), and the entire gap between the bound and the truth sits in one place: how far the encoder's guess is from the true posterior.

3. Noise as a teacher. The random $\boldsymbol{\epsilon}$ of Equation (4.9) means the decoder must turn not just one code but a whole small *neighbourhood* of codes back into $\boldsymbol{x}$. That is what makes the latent space smooth — and smoothness is exactly what a sampler needs.

All three come back later: Chapter 5 swaps the probabilistic bottleneck for a spatial one (masking), and Chapter 7 swaps one dose of noise for a whole schedule of them (diffusion).

4.8 Where the Lecture Goes Next: Masking Instead of Bottlenecking

This chapter's two halves attack the same problem—stop the hourglass from cheating—with non-overlapping tools, and the lecture puts them in sequence precisely so you notice the baton pass. The classical AE *narrows the channel* and loses to capacity: Section 4.2's codebook is what unlimited width does with a narrow waist. The VAE *prices the channel* and loses to slack: a strong enough decoder reads around the code and the encoder quietly collapses to the prior. The masked autoencoder (MAE) does something conceptually different, and it is worth three paragraphs here so Chapter 5 lands on a mechanism rather than a surprise.

The move. Rather than restricting *what* can pass through the bottleneck, MAE removes most of the *content from the input itself*. Given an image divided into non-overlapping patches, it hides a large fraction of them—typically 75% (He et al., 2022)—and asks the system to reconstruct the hidden ones from what remains. The trivial shortcuts are dismantled upstream: there is no longer a near-complete input to copy, so the codebook strategy of Section 4.2 has nothing to index into; and with three quarters of the pixels inadmissible, pixel-level interpolation no longer solves the task, so the model is forced toward exactly the object-level structure the lecture has been gesturing at since its IB slide.

The asymmetry. Reconstruct from 25% of the tokens and the balance of compute changes sides. The encoder, which is the artifact worth keeping for downstream tasks, needs to process only the *visible* patches (about a quarter of the full sequence), so it can be made very large within a fixed budget. The decoder, which is needed only during pretraining, can stay small—it recombines the encoded visible tokens with shared learned mask tokens, and is discarded after training, which is the reverse of the VAE's lifecycle in Figure 4.6. The result is an *asymmetric* autoencoder: heavy encoder on the quarter of tokens, light decoder on the full sequence, and that is where the scalability of the method comes from.

The diagnostic. MAE reframes the "AE vs. VAE" axis from *generative model with structured representation* against *compressor with unstructured representation*, into a third option: a *representation learner that does not try to be generative at all*. Its decoder exists to prove the encoder understood, not to be sampled from; the latent it leaves behind is judged on transfer, not on log-likelihood. The question this poses—*is a good SSL method one that generates well or one that transfers well, and are those the same thing?*—is the seam between Chapters 5 and 6, and it will keep reopening through the rest of Part II. This chapter's job was to build the vocabulary in which the question is legible.

Check Your Understanding

Level 1. **(1)** State the three hyperparameters of an autoencoder the lecture emphasizes, and one symptom of setting each too high or too low. **(2)** In Figure 4.1, what exactly flows through the dashed loop at the bottom, and why can this loop replace labeled data?
Level 2. **(3)** A friend proposes training an AE with a 1-D latent on N images and brags about zero training error. Construct two incompatible explanations and say what measurement separates them. **(4)** In Figure 4.3 panel (b), why is the loss $\|\boldsymbol{z}_1 - \hat{\boldsymbol{z}}_1\|_2$ rather than $\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|_2$? What does the gray (frozen) box guarantee? **(5)** Redo the s=4 column of Figure 4.4 by hand (both features), checking each stage of Equation (4.3).
Level 3. **(6)** Which step of the ELBO derivation in Section 4.5.1 is the *only* inequality, and what single quantity measures the size of the gap it opens? **(7)** The encoder outputs $\mu = 1$, $\sigma^2 = 2$ for one latent dimension. Compute this dimension's contribution to Equation (4.12), and the contribution if σ^2 doubles at fixed μ. **(8)** In Equation (4.9), compute $\partial \boldsymbol{z}/\partial \boldsymbol{\sigma}$. Why does placing $\boldsymbol{\epsilon}$ outside the network suffice for training $\boldsymbol{\phi}$, even though $\boldsymbol{\epsilon}$ changes on every iteration?
Level 4. **(9)** Show from Equation (4.7) that the reconstruction term of the ELBO, at fixed η, differs from mean squared error only by a multiplicative constant and an additive one. What does this imply about which parameter η *really* controls? **(10)** Explain posterior collapse in one sentence using only the two terms of Figure 4.9, then name one architectural and one optimization-side remedy. **(11)** A VAE's average posterior $\mathbb{E}_{\boldsymbol{x}}[q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})]$ is bimodal but the prior is not. Which ELBO term should have prevented this, and why does it fail?
Level 5. **(12)** β-VAE multiplies the KL term by $\beta > 1$. Predict, with reasoning, what happens to (a) reconstruction sharpness, (b) latent disentanglement, (c) posterior collapse, as $\beta \to \infty$. **(13)** Give two distinct reasons why the latent walk of Figure 4.10 can pass through a code whose decode is garbage even in a well-trained VAE. **(14)** Show from first principles that a rank-K *linear* autoencoder trained to minimize Equation (4.2) recovers the principal-component subspace, and say which modeling addition breaks the equivalence. **(15)** Worked Example 4.2 claimed that an AE with a *one-dimensional* latent can perfectly memorize any finite training set given enough capacity. Explain why "the code was one-dimensional" does not contradict the claim, and name the measured quantity that separates this regime from genuine compression.

Solution sketches.

- **(1)** Latent dimensionality (too small: irreducibly poor reconstruction; too large: vacuous compression); encoder/decoder capacity (too small: loss diverges or stalls; too large: dataset memorization); regularization on the code (too weak: unstructured, codebook-prone latent; too strong: underfitting, posterior collapse at the extreme).

- **(2)** The reconstruction error $\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|_2^2$ and its gradients flow decoder→encoder. It replaces labels because the target *is* the input—supervision is synthesized from the data itself.
- **(3)** Explanation A: the code indexes the training set (codebook; Figure 4.2 right); explanation B: the data lie on a 1-D manifold the model found. Reconstruction error on *held-out* images separates them: near-zero only in case B.
- **(4)** Layer 2's job is to compress *codes*, so its input/output are $\boldsymbol{z}_1, \hat{\boldsymbol{z}}_1$; pixel loss would re-solve layer 1's problem. Freezing layer 1 guarantees the second AE sees a stationary input distribution throughout its own training.
- **(5)** Feature 1: $\hat{x} = (6-3)/\sqrt{5.01} \approx 1.34$, then $y = 1.2(1.34) + 0.1 \approx 1.71$. Feature 2: $\hat{x} = (5-2)/\sqrt{5.01} \approx 1.34$, then $y = 0.8(1.34) - 0.2 \approx 0.87$. Both match the figure.
- **(6)** Step 3. The gap equals $D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}))$, the approximate-versus-true posterior divergence; the bound is tight iff this KL is zero.
- **(7)** At $\mu = 1, \sigma^2 = 2$: $\frac{1}{2}(1 + \log 2 - 1 - 2) \approx -0.653$ nats. At $\sigma^2 = 4$: $\frac{1}{2}(1 + \log 4 - 1 - 4) \approx -1.807$ nats. Widening the posterior costs more because the $-\sigma^2$ term grows linearly while the $+\log \sigma^2$ term that partly offsets it grows only logarithmically.
- **(8)** $\partial \boldsymbol{z} / \partial \boldsymbol{\sigma} = \mathrm{diag}(\boldsymbol{\epsilon})$ (element-wise $\boldsymbol{\epsilon}$). Changing $\boldsymbol{\epsilon}$ each iteration changes the *value* of $\boldsymbol{z}$ but not the *relationship* $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$; gradient descent on $\boldsymbol{\phi}$ optimizes the expectation over $\boldsymbol{\epsilon}$, estimated unbiasedly by each draw.
- **(9)** $\log p(\boldsymbol{x} \mid \boldsymbol{z}) = -\frac{1}{2\eta}\|\boldsymbol{x} - G_{\boldsymbol{\theta}}(\boldsymbol{z})\|^2 - \frac{D}{2}\log(2\pi\eta)$. Multiplying the ELBO through, $1/(2\eta)$ rescales the reconstruction term relative to the KL term: fixed η is a disguised β on the reconstruction side, controlling the reconstruction-compression exchange rate.
- **(10)** A decoder strong enough to predict $\boldsymbol{x}$ without reading $\boldsymbol{z}$ lets the KL term win unconditionally: collapse means "term 1 gains nothing from reading the code, so term 2 deletes the channel." Architectural remedy: weaken the decoder (non-autoregressive likelihood); optimization remedy: KL annealing or free-bits (minimum information per dimension).
- **(11)** The KL term. It penalizes each $q(\boldsymbol{z} \mid \boldsymbol{x}^{(i)})$ against the prior *individually*; a mixture of two standard-normal-like lobes (bimodal aggregate) yields the same per-point penalty as one lobe, so the aggregate mismatch—the prior hole—is invisible to the ELBO.
- **(12)** (a) Sharper pressure against reconstruction ⇒ *blurrier* output; (b) more disentanglement up to a point, as independent factors are the cheapest codes; (c) $\beta \to \infty$ forces $D_{\mathrm{KL}} \to 0$ regardless of reconstruction: total posterior collapse, the encoder ignores $\boldsymbol{x}$.
- **(13)** (i) Prior holes: the aggregate posterior does not fill the prior, so a grid point can sit in a region no training posterior used (Section 4.6.4). (ii) Interpolation across class boundaries: a straight-line walk can cross the boundary between two far-apart digit clusters, where decodes are transitional artifacts rather than data.
- **(14)** With $e(\boldsymbol{x}) = \boldsymbol{W}_e\boldsymbol{x}$, $d(\boldsymbol{z}) = \boldsymbol{W}_d\boldsymbol{z}$, the composition $\boldsymbol{W}_d\boldsymbol{W}_e$ is an arbitrary rank-$\leq K$ linear map, so Equation (4.2) asks for the best rank-K approximation of the data in least squares. By Eckart–Young–Mirsky the optimum is the orthogonal projector onto the top-K eigenspace of the covariance—the PCA subspace—and the encoder's codes

are the principal-component scores up to an invertible mixing of the coordinates. Any nonlinearity inside e or d breaks the equivalence by freeing the model from the rank-K subspace constraint.

- **(15)** One dimension limits the *number of distinct messages* the code can carry, not the *information the decoder can invoke per message*: the encoder can classify each training input into one of N disjoint latent intervals, and for each interval the decoder stores a full image. Memorization lives in the weights, not the code. The separating measurement is held-out (or perturbed-code) reconstruction error, which is near zero only for a genuine compressor.

Concept Map — how this chapter connects

Backward links. Chapter 1's dogma—structure in the data is free supervision—is instantiated here as *reconstruction*: the plain AE's target is its own input (Section 4.1). Chapter 2 supplies every tool this chapter ran on: KL divergence and mutual information for the IB view (4.1) and the decomposition (4.5), Jensen's inequality for Step 3 of the derivation, Gaussian densities for Equations (4.7) and (4.12), and mini-batch statistics for BatchNorm (4.3). Chapter 3's clustering codebook is the memorization failure mode of Section 4.2 with the learning removed. **Sideways link.** Chapter 6 re-reads the VAE as the first generative model of the taxonomy: normalizing flows replace its approximate posterior with exact invertibility, GANs replace its likelihood with a game. **Forward links.** Chapter 5 replaces the probabilistic bottleneck with a spatial one: the masked autoencoder hides 75% of patches and learns *representations* rather than densities, settling the "generative vs. representation" tension this chapter leaves open. Chapter 7 turns Section 4.6.2's single noise injection into a schedule: denoising autoencoders (already met as the DAE of the Common Confusion box) become learned diffusion. Chapter 15 asks whether the ELBO's success here is evidence about data geometry or about exchange rates.

CHAPTER 5

Masked Autoencoders and Masked Image Modeling

"I can't make an image, but I can tell you what is missing in one."
— paraphrasing the spirit of every mask-and-predict pretext task

In this chapter

5.1 The pretext task: masking as generation ..... 176 • From reconstruct-everything to reconstruct-the-missing • Why images tolerate extreme masking
5.2 The MAE machinery, step by step .... 178 • The full pipeline at a glance • Formalizing the pipeline • The four-step reveal • Reading the reconstructions
5.3 The asymmetric design 186
5.4 What to reconstruct: pixels, not tokens 189 • BEiT: discrete visual tokens • The pixel target, concretely: per-patch normalization • An MAE training step in code
5.5 Why does this actually work? 194
5.6 What MAE is for: representation learning ..... 195 • Pretrain, then adapt
5.7 Headline results 196 • ImageNet classification • Transfer to detection and segmentation • The masking ratio, one more time
5.8 Failure modes and boundaries 199
5.9 Beyond still images 200
5.10 Check Your Understanding 200
5.11 Concept Map 203

Chapter roadmap

Before you start. You need the autoencoder idea from Chapter 4 (squeeze and rebuild) and the vision transformer from Section 2.2.3 (an image cut into patches, each patch becoming a token). You do *not* need the VAE mathematics — this chapter drops probability almost entirely.

The idea in one sentence. Delete three quarters of an image, run only the surviving quarter through a big network, and make a small network guess the deleted pixels.

Why it is a big deal. Three wins arrive at once. *Pedagogically*: the task is so hard the network cannot cheat by copying — the hidden patches never reach the encoder, and most of their neighbours are hidden too. *Computationally*: the expensive part of the network only ever sees 25% of the tokens, so pretraining runs several times faster than it looks like it should. *Empirically*: the encoder you keep afterwards transfers better than anything built so far in this book.

What we cover.

- Why an image can lose 75% of its content and still be guessable, while a sentence cannot lose 75% of its words.
- The pipeline in four steps, with a diagram for each.
- The *asymmetric* design — a big encoder and a deliberately small decoder — and why the asymmetry is the point rather than an optimisation.
- What to predict: raw pixels versus discrete tokens (BEiT), and why the simpler answer won.
- The results, and the masking-ratio curve everybody quotes.

Afterwards you will be able to explain why MAE's decoder is thrown away, why the masking ratio for images is so much higher than for text, and why MAE features win on segmentation while losing on nearest-neighbour classification — a trade-off Chapters 11 and 16 return to repeatedly.

Chapter 4 closed with a tension. The variational autoencoder turned the hourglass into a generative model — a device for sampling *new* images — but the sample quality was visibly blurry, and the whole probabilistic machinery (priors, posteriors, ELBO) was heavy. Meanwhile the plain deterministic autoencoder, given enough capacity, happily memorized a codebook of the training set and learned nothing transferable. This chapter resolves that tension with a move that looks deceptively modest: *do not feed the encoder the whole image.* Show it a quarter of the pixels, ask the decoder to guess the missing three quarters, and throw the decoder away afterwards. What remains is an encoder pretrained on a generative pretext task that transfers better than anything we have built so far — this is the *Masked Autoencoder* (MAE) of He et al. (2022), the subject of the lecture's act III (Lecture 03, slides 23–39).

The chapter first develops the pretext task itself (random patch masking as a generative task, Section 5.1), then walks through the MAE pipeline step by step exactly as the lecture reveals it (Section 5.2), analyzes the two design decisions that make it fast — asymmetric compute and raw-pixel targets (Sections 5.3 and 5.4) — and closes with what the reconstruction is actually *for*: representation learning, its evaluation protocols, and the headline transfer numbers (Sections 5.6 and 5.7).

5.1 The pretext task: masking as generation

5.1.1 From reconstruct-everything to reconstruct-the-missing

Recall the plain autoencoder objective from Chapter 4: feed $\boldsymbol{x}$, compress to a latent $\boldsymbol{z}$, reconstruct $\hat{\boldsymbol{x}}$, and minimize $\mathcal{L} = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|_2^2$. The failure mode we identified — a large enough encoder–decoder pair can memorize an index into the dataset rather than generalize — was fixed in the classical toolbox only by shrinking the bottleneck or adding regularization, both of which fight the capacity we actually want to use. The VAE attacked the same problem probabilistically, at the price of a recognizably complicated objective.

Masked image modeling takes a different route: *corrupt the input* so that reconstruction can no longer be solved by memorizing or copying. Split an image into a grid of non-overlapping patches, hide most of them, and ask the model to predict the hidden content from the visible remainder. Now the encoder never sees what it has to reconstruct, so the identity shortcut is destroyed *by construction* rather than suppressed by a penalty term. The idea is directly ancestral to BERT's masked language modeling (Devlin et al., 2019), where a fraction of word tokens is hidden and predicted from context.

Key Concept — MAE is a generative pretext task

Masked image modeling is *generative* in the formal sense of Chapter 6: it learns a conditional model $p(\text{missing pixels} \mid \text{visible pixels})$ of the data distribution. But unlike the VAE of Chapter 4, the *generation is a means, not an end*. The generative decoder is discarded after pretraining; the product that ships is the encoder, used as a representation extractor. The same loss therefore plays two roles depending on your framing — density estimation for a generative model, pretext task for a representation learner — and the MAE paper is emphatic about the second framing (He et al., 2022).

5.1.2 Why images tolerate extreme masking

The quantitative heart of the pretext task is a single design choice: the *masking ratio*, the fraction of patches hidden from the encoder. The lecture's mechanics slides show a flamingo photograph masked at 75% (slide 25), and the closing reconstruction gallery pushes the ratio to 95% while the output remains recognizable, if heavily blurred (slide 35; Section 5.2.4 reads those galleries in detail). Why does a task this aggressive even have a solution, let alone teach good features?

Because natural images are *spatially redundant*. Neighboring patches are statistically near-duplicates: sky patches resemble sky patches, a hidden patch on a dog's flank is strongly predicted by its visible neighbors. Removing three quarters of the patches removes far less than three quarters of the *information*. Language has far less redundancy — words carry much higher semantic density, which is precisely why BERT masks only about 15% of tokens (Devlin et al., 2019). The 75% figure is not an accident; it sits at the sweet spot where the

task is hard enough that the encoder must understand scene structure, yet solvable enough to give a consistent gradient. Too low a ratio and the task degenerates to interpolation between neighboring patches, learnable with low-level cues and no semantics — the close cousin of the "copy through the bottleneck" failure from Chapter 4.

Everyday analogy — a jigsaw with most pieces missing

Throw away three quarters of a jigsaw's pieces and hand someone the rest. Finishing the sky from a few blue pieces is easy, because sky is predictable. Rebuilding a cat's ear that touched no remaining piece requires knowing what cats look like.

Notice where the effort goes. The solver studies the *real* pieces hard. Once they understand the picture, filling the gaps is mechanical. And after training, you only need the solver's trained eye — not the gap-filling. That is MAE: the encoder studies only the visible patches; the decoder does the cheap filling-in and is then thrown away.

Where the analogy breaks. A human solver fills gaps with one confident guess. MAE's decoder, trained with squared error, fills them with the *average* of plausible guesses — hence the blur.

Why images need far more masking than text.

- **Text is dense.** Hide 15% of a sentence — "The cat sat on the _____" — and grammar alone fills the gap. Hide 75% and nothing recoverable is left.
- **Images are redundant.** Hide 15% of an image and each gap can be filled from its neighbours — the network learns texture, not objects. You have to hide much more before the task requires understanding.

Same question in both cases — "how much must I hide to make the task interesting but still solvable?" — with very different answers, because the two kinds of data are redundant to very different degrees.

Both extremes fail.

- **Too little masking ($r \to 0$).** Gaps can be filled by smoothing from neighbours, so the network learns local smoothness, not scenes. And the speed-up disappears, because the encoder processes almost the whole image.
- **Too much masking ($r \to 1$).** Ten scattered patches cannot tell you what the scene was. The decoder averages over possibilities, reconstructions blur, and the features get worse too.

In between lies a surprisingly wide range that works well, and 75% sits comfortably inside it. Section 5.7.3 shows the measured curves.

Common Confusion

MAE masking ratio vs. BERT's 15%. A natural first question is: if masking 75% of an image works, why did the inventors of masked language modeling settle for 15% (Devlin et al., 2019)? The answer is the information-density asymmetry above, not a bug or an oversight. Mask 75% of a sentence and the residue is generally not a sentence at all — the pretraining task becomes nonsense and the model is forced to hallucinate. Mask 15% of an image and the residue is essentially the whole image: the missing patches are trivially interpolated from their neighbors, and the encoder learns local texture statistics rather than scene semantics. The two numbers are two answers to the same calibration question — *make the task as hard as possible while it still has a unique, learnable answer* — on substrates with very different redundancy. The evaluation curves on the image side (Figure 5.9) confirm that this calibration is real: accuracy is strongly non-monotonic in the masking ratio.

5.2 The MAE machinery, step by step

Model at a glance: Masked Autoencoder (MAE) He, Chen, Xie, Li, Dollár & Girshick, 2022

In one line	Hide 75% of an image's patches, encode only the visible ones with a large transformer, and let a small transformer paint the missing patches back in.
Problem it solves	Plain autoencoders can cheat by copying; VAEs are heavy and blurry. MAE makes the task hard enough that cheating is impossible and cheap enough that it scales.
Key intuition	Images are enormously redundant — a missing patch of sky is guessable from other sky. At 75% masking most of a hidden patch's neighbours are hidden too, so simple interpolation no longer works; you have to understand the whole scene. The difficulty of the task *is* the feature.
Inputs	Unlabelled images. No augmentation pipeline is needed, which is itself notable — contrast Chapters 8 and 9, where augmentation design is the whole game.
Architecture	A **large** ViT encoder that sees only the 25% visible patches; a shared learned *mask token* inserted at every hidden position; a **small** ViT decoder that sees the full sequence.
Loss	Mean squared error on the pixels of the *masked patches only*, with per-patch normalisation. Visible patches are not scored.
Why asymmetric	The encoder is the product; the decoder is scaffolding. A small decoder costs little accuracy, saves a great deal of compute, and — the real reason — forces the scene understanding to live in the encoder rather than the decoder.
What you keep	The encoder. At fine-tuning time it is fed complete images with no masking, a train/test mismatch that in practice does not hurt.

Best at	Fine-tuning, and dense tasks — detection, segmentation, depth — where patch-level layout matters.
Weak at	Frozen-feature evaluation. Linear probes and kNN on raw MAE features lag joint-embedding methods badly, because nothing in the loss ever asked for a linearly separable global descriptor.
Relatives	BEiT (predict discrete tokens instead of pixels), SimMIM (same idea, simpler decoder), iBOT and DINOv2 (masked prediction *plus* a joint-embedding loss — Chapter 11).

In plain words — why the decoder is small on purpose

It is tempting to read "small decoder" as a cost-saving compromise. It is closer to the opposite. Whatever knowledge is needed to fill in the missing patches has to live *somewhere*. If the decoder is large, the decoder can hold it — and the decoder is the part you throw away. Shrinking the decoder is a way of forcing the knowledge into the encoder, which is the part you keep. The speed-up is a bonus, not the motivation.

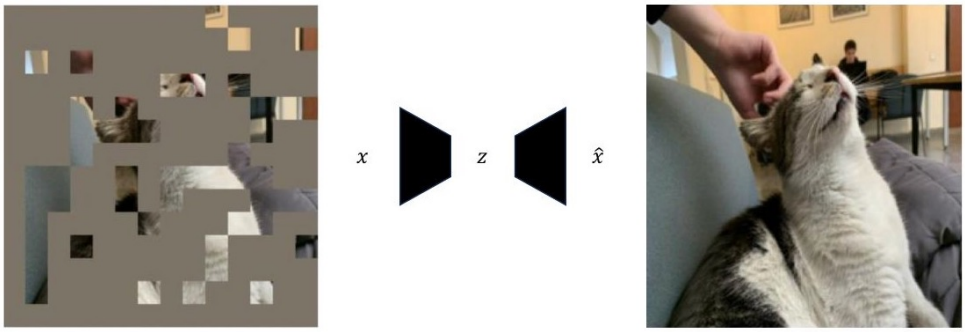


Figure 5.1. The demonstration that makes masked autoencoding believable. *Left:* the input — roughly 80% of the patches replaced by flat grey. *Middle:* the familiar hourglass. *Right:* the reconstruction. Spend a moment on the left panel before looking right: what is visible is a scatter of fur, an ear, part of a hand, and nothing else.

How to read it: The point is not that the reconstruction is beautiful — it is soft and imperfect. The point is that it is correct about what the scene contains*, which cannot be done by local interpolation from the visible patches.*

Source: Lecture 03, slide 24 (He et al., 2022).

5.2.1 The full pipeline at a glance

Figure 5.2 shows the complete pretraining pipeline, which the lecture reveals in four steps (slides 25–28): (1) *patchify* the image and *randomly mask* 75% of the patches; (2) *encode* only the visible patches with a large ViT encoder; (3) *insert learned mask tokens* at the positions

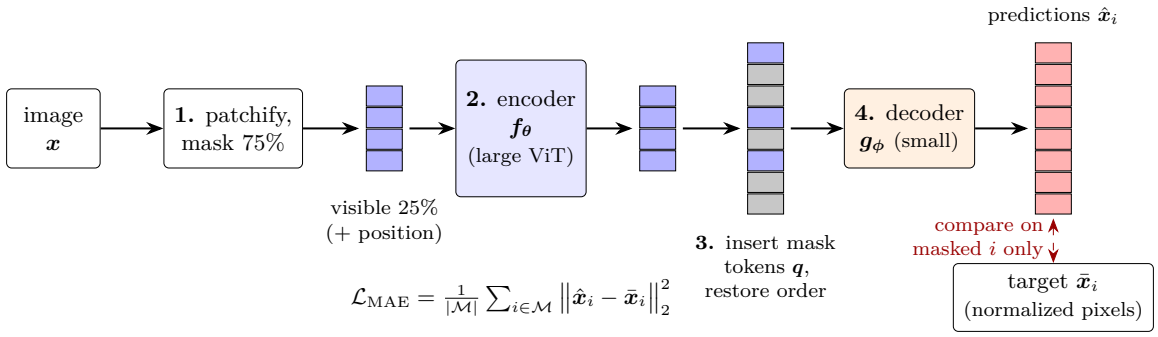


Figure 5.2. The Masked Autoencoder pretraining pipeline (Lecture 03, slides 24–31). The input image is split into non-overlapping patches; a uniform random subset (75%) is masked, and the remaining visible patches are embedded by a linear projection with positional embeddings. **Only the visible tokens** enter the large ViT encoder $\boldsymbol{f_\theta}$ — the encoder never sees a mask token. Afterwards the sequence is unshuffled back to the original patch order and a shared, learned mask token $\boldsymbol{q}$ is inserted at every masked position. A lightweight transformer decoder $\boldsymbol{g_\phi}$ processes the full sequence and regresses the pixels of each masked patch; the mean-squared error is computed only on the masked positions $\mathcal{M}$ against the (per-patch normalized, Section 5.4) pixel target.

of the removed patches to restore the full sequence; (4) *decode* the full sequence with a small transformer and *reconstruct* the missing pixels.

5.2.2 Formalizing the pipeline

Let the input image be $\boldsymbol{x} \in \mathbb{R}^{H \times W \times C}$, where H and W are the spatial dimensions and $C = 3$ the color channels. Following the Vision Transformer (Dosovitskiy et al., 2021), MAE divides $\boldsymbol{x}$ into N non-overlapping patches $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, each of size $p \times p$ pixels, where p is the patch size (typically $p = 16$) and $N = HW/p^2$ (for the standard 224×224 input this gives $N = 196$ patches; think of $p^2C = 768$ raw pixels per patch).

Masking. A *masking ratio* $r \in (0, 1)$ is chosen — the default is $r = 0.75$. A uniform random subset $\mathcal{M} \subset \{1, \ldots, N\}$ of $|\mathcal{M}| = rN$ patch indices is drawn per image and *masked out*; the complementary set $\mathcal{V} = \{1, \ldots, N\} \setminus \mathcal{M}$ of visible patches has size $(1 - r)N$. Two properties of this sampling scheme are doing quiet work. First, the sampling is *uniform* over positions, so the visible subset carries no spatial prior: the model cannot game the task by, say, always keeping the image center, and no patch location is systematically easier than any other. Second, every image gets a *fresh* random subset drawn at every training step, so nothing about the visible/masked assignment can itself be memorized — the split is not a constant of the dataset, and there is no "this patch is always hidden" shortcut to discover. Concretely, at $r = 0.75$ an input with $N = 100$ patches reduces to 25 visible tokens: three quarters of the evidence vanishes *before* any computation happens, and everything the network will ever know about the image on this forward pass is in those 25 tokens.

Encoder. Each visible patch $\boldsymbol{x}_i$, $i \in \mathcal{V}$, is flattened, mapped to a d-dimensional vector by a shared linear projection (the same patch embedding as in a ViT (Dosovitskiy et al., 2021)), and a learned positional embedding for position i is added so that the model knows *where* the patch came from. The sequence of embedded visible tokens is processed by the encoder $\boldsymbol{f}_{\boldsymbol{\theta}}$, a stack of transformer blocks (multi-head self-attention (Vaswani et al., 2017) plus feed-forward layers) with parameters $\boldsymbol{\theta}$:

$$\boldsymbol{h}_i \;=\; \boldsymbol{f}_{\boldsymbol{\theta}}\big(\{\boldsymbol{x}_j\}_{j\in\mathcal{V}}\big)_i, \qquad i \in \mathcal{V}, \tag{5.1}$$

where $\boldsymbol{h}_i \in \mathbb{R}^d$ is the encoding of visible patch i. Both parts of this step matter:

- **The shared linear projection** turns each flattened patch into a vector, exactly as in a standard ViT (Dosovitskiy et al., 2021) — a transformer needs a *list* of vectors, not a 2-D grid.
- **The position embedding** tells the network *where* each patch came from. Attention alone ignores order, so without it the encoder would see a bag of patches, not a layout: it could not tell the flamingo's head from the same pixels at its feet.

The key point about (5.1) is what is *missing*: no mask token enters the encoder. BERT puts a special `[MASK]` token at every hidden position and runs it through the encoder (Devlin et al., 2019). MAE simply *removes* those positions — and that is what makes the asymmetric design pay off.

Key Concept — The encoder never sees a mask token

Hidden positions are *removed* from the encoder's input, not replaced by placeholders. The reason: the encoder is the expensive part, so it should spend its computation only on real image content. A mask token carries no information — it is the same learned vector at every hidden position of every image — so processing it in the encoder is pure waste. Placeholders are only needed where a prediction has to be *placed*, which is the decoder's job. So the encoder sees $(1-r)N$ tokens instead of N: that is the whole speed story of Section 5.3.

Mask tokens and unshuffle. The decoder must predict content at *specific spatial positions*, and a transformer's only carriers of position and capacity are its tokens. MAE therefore re-expands the sequence to full length: at every masked position $i \in \mathcal{M}$ it inserts a shared, *learned* mask token $\boldsymbol{q} \in \mathbb{R}^d$ (a single parameter vector, the same at every masked position of every image), keeps the encoded $\boldsymbol{h}_i$ at every visible position, and adds positional embeddings again, so the decoder sees a complete length-N sequence in the *original* spatial order. The mask token $\boldsymbol{q}$ is trained jointly with everything else; it is the decoder's learned placeholder for "a patch used to be here — imagine it." Notice that the scheme needs *two* rounds of positional information, and that this is not redundancy. The encoder must remember *which observed content* each embedded token refers to, so encoded tokens should carry their source coordinates through the transformer blocks. The decoder faces the dual problem: the

mask token $\boldsymbol{q}$ is content-free — bit for bit the same vector at every masked position — so after the unshuffle the decoder learns *where each prediction has to land* exclusively from the second, decoder-side positional embedding. The encoder's embeddings say "this evidence came from here"; the decoder's say "your guess here is due at this grid location." Removing either round breaks a different half of the pipeline.

Decoder and loss. The full sequence is processed by the decoder $\boldsymbol{g}_{\boldsymbol{\phi}}$, a second transformer with parameters $\boldsymbol{\phi}$, followed by a linear layer that maps each output token to a p^2C-dimensional patch prediction. The prediction at position i is denoted $\hat{\boldsymbol{x}}_i \in \mathbb{R}^{p^2C}$. Training minimizes the reconstruction error on the masked patches only:

$$\mathcal{L}_{\text{MAE}}(\boldsymbol{\theta}, \boldsymbol{\phi}) \;=\; \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \|\hat{\boldsymbol{x}}_i - \bar{\boldsymbol{x}}_i\|_2^2, \tag{5.2}$$

where $\bar{\boldsymbol{x}}_i$ is the target for masked patch i: its raw pixels, optionally normalised per patch (Section 5.4).

This loss blocks both of the cheap ways out:

- **Copying the input.** The encoder never sees the hidden patches, so there is nothing to copy. (Chapter 4 hoped a small bottleneck would prevent copying; masking prevents it outright.)
- **Always predicting the average patch.** Every image and every position has a different target, so a constant guess is wrong everywhere. The only way down is to predict *this* image's missing content.

Why only the hidden patches are scored. Suppose the loss also counted the visible patches. Those arrive at the decoder already encoded, so copying them back is easy and teaches nothing. A quarter of the loss would then reward copying — effort and gradient spent on a task that is already solved — diluting the three quarters that reward the real task: predicting what was hidden. Scoring *only* the masked positions $\mathcal{M}$ puts all of the learning pressure on guessing unseen content from context. It is the difference between training the network to *infer* and training it to *transcribe*.

After pretraining, the decoder $\boldsymbol{g}_{\boldsymbol{\phi}}$ is *discarded*. Transfer (Section 5.6) uses the encoder $\boldsymbol{f}_{\boldsymbol{\theta}}$ alone — now fed *all* patches, unmasked.

5.2.3 The four-step reveal

The lecture reveals the pipeline in four steps, one slide each (25–28); Figure 5.3 puts them in one strip. Two slides around it are worth keeping in mind:

- **Slide 23** shows the paper itself. Its main claim is not a new task but *scalability*: a masking task cheap enough that vision encoders can grow the way language models did — delivered by the asymmetric design of Section 5.3.

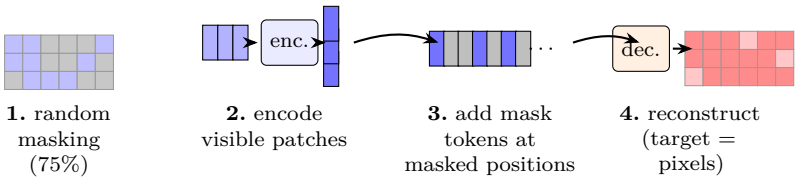


Figure 5.3. The lecture's four-step reveal of the MAE mechanics (Lecture 03, slides 25–28, after the ECCV'22 tutorial graphics). (1) An image is divided into a patch grid and 75% of the patches are masked at random (dark cells). (2) Only the visible patches are extracted and pushed through the encoder, yielding latent tokens. (3) The sequence is re-expanded to full length by placing shared learned mask tokens (grey) at the masked positions, restoring the original spatial order. (4) A lightweight decoder maps the full sequence to pixel predictions, which are compared against the original (unmasked) image — the target is the input itself.

- **Slide 24** makes the boldest claim first: a cat photo with about 80% of it hidden is reconstructed recognisably. Not pixel-exact — a *plausible* cat rather than a copy — which already hints that the network learned the idea "cat", not a lookup of nearby pixels.

Sections 5.2.4 and 5.5 come back to both points.

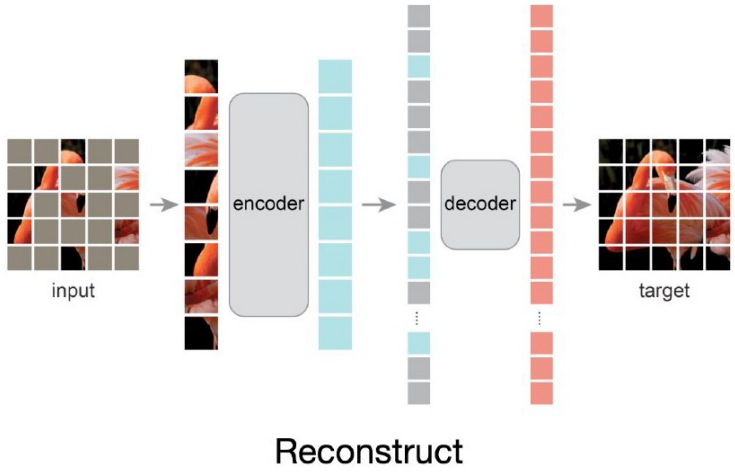


Figure 5.4. The MAE pipeline end to end. Left to right: the masked input; the visible patches pulled out and fed to the encoder; the encoder's output tokens (blue); mask tokens (grey) inserted at the positions that were hidden, restoring full sequence length; the decoder; and the reconstructed target.

How to read it: Count what enters the encoder against what enters the decoder. The encoder column is short — only the visible patches — while the decoder column is full length. That asymmetry is the source of both the speed-up and the design's central idea.

Source: Lecture 03, slide 28; graphic after the ECCV'22 MAE tutorial (Xinlei Chen).

5.2.4 Reading the reconstructions

Before the mechanics settle in too comfortably, it is worth looking at the artifact the lecture interposes between the walkthrough and the results: slides 32–35 are a small gallery of what the trained decoder actually produces, and it is pedagogically load-bearing, since everything else in this chapter hinges on believing that the reconstructions mean anything at all.

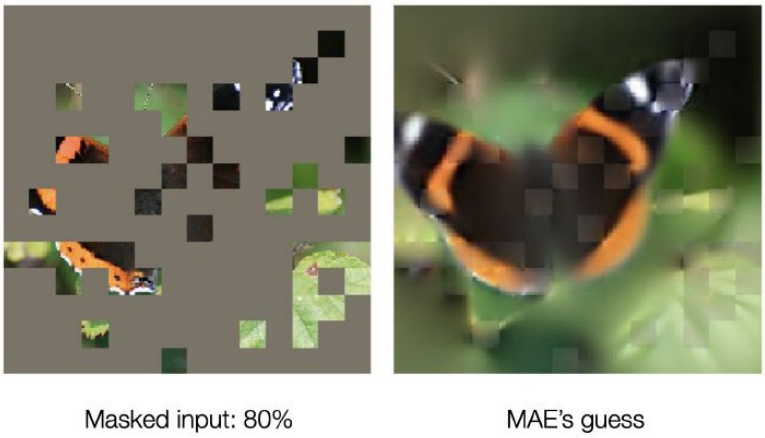


Figure 5.5. The lecture's guessing game, with the answer. At 80% masking a human can usually say roughly what the scene is, and so can the model. The two abilities agreeing is the entire premise of the pretext task: if the content were genuinely unrecoverable, predicting it would teach nothing.

How to read it: Try the left panel yourself before reading the right one. Notice what you used: not the neighbouring pixels of any gap, but the global arrangement of colour and texture across the whole frame. That is exactly what the network is forced to learn.

Source: Lecture 03, slides 32–33; ECCV'22 MAE tutorial (Xinlei Chen).

Slide 32 is staged as an interactive guessing game: an ImageNet photograph masked at 80% is shown next to an empty dashed box labeled "You guess?", and slide 33 delivers the reveal — the MAE decoder's reconstruction is a blurred but unmistakable red-admiral butterfly resting on green foliage. The virtue of this pairing is that it separates two questions the pipeline figure conflates: *can a human* infer the missing content from the visible 20% (yes, approximately), and *can the model* (also yes, approximately). The two agreeing is the entire premise of the pretext task.

Slide 34 shows eight more examples — a vintage truck, a show-jumping horse, a surfer, a baseball batter, a dog behind a fence, two people against a pink wall, a busy scooter street, highland cattle — each as masked input, reconstruction and original. Two things stand out:

- **Right in content, soft in detail.** What the object is, its pose, the layout and the colours are all correct; fine texture is smoothed. That is what squared error predicts:

when several completions are possible, it favours their *average* — the same effect that made Chapter 4's VAE samples blurry.
- **Not cherry-picked.** The blur is plainly visible, and the lecture leaves it on screen because it tells you something.

How much masking is too much? Slide 35 masks the same mushroom photo at three levels:

- **75%** — a plausible reconstruction;
- **85%** — still clearly a mushroom;
- **95%** — a single blurred blob.

At 95% the few visible patches fit many different scenes, so the decoder's best guess is the average of them all — a blur. The lesson: the task that teaches the best features is not the hardest task you can set. Figure 5.9 puts numbers on this.

Learning note from study sessions — A useful way to reread the gallery

A study-session habit worth adopting: when you look at any MAE reconstruction, resist asking "is it sharp?" and instead ask "which choices did the model have to commit to?" Sharpness is a low-level budget — the pixel target capped it on day one; commitment to object identity, pose, and scene layout is a semantic budget, and it is the one pretraining spends. The blurred butterfly is more informative than a photorealistic one would be, exactly because its blur tells you which axis of difficulty the loss left unsolved.

Worked Example — Patch counts on a 224×224 image

Take the standard ViT configuration: input $H = W = 224$ pixels, patch size $p = 16$, three channels.

- Patches per side: $224/16 = 14$, so $N = 14 \times 14 = 196$ patches.
- Raw numbers per patch: $16 \times 16 \times 3 = 768$ pixel values — so each *prediction* $\hat{\boldsymbol{x}}_i$ in (5.2) is a 768-vector.
- At masking ratio $r = 0.75$: $|\mathcal{M}| = 0.75 \times 196 = 147$ masked patches, $|\mathcal{V}| = 196 - 147 = 49$ visible patches reach the encoder. The pairwise attention cost of a transformer block scales quadratically in the sequence length, so the encoder's attention term drops roughly by a factor $(196/49)^2 = 16$; even with the linear-in-length terms mixed in, the encoder wall-clock saving is several-fold (the paper reports overall pretraining speedups of roughly $3\times$ or more from the asymmetric design, because the decoder covers the remaining positions with far less compute — see Section 5.3).
- Loss reduction: $\mathcal{L}_{\text{MAE}}$ averages the squared error over the 147 masked 768-vectors — 112,896 scalar residuals per image, all about content the encoder never saw.

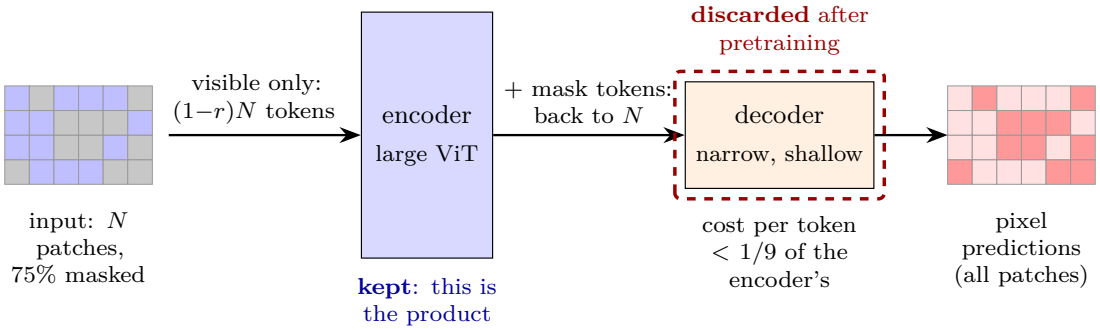


Figure 5.6. The asymmetric autoencoder (Lecture 03, slides 30–31). The encoder (tall block, left) processes *only the visible quarter* of the tokens and is deliberately made very large. The decoder (short block, right) processes the full-length sequence but is narrow and shallow — the slides quantify this as less than one ninth of the encoder's computation per token. Because the decoder exists solely to produce the reconstruction *loss*, it is discarded after pretraining; the encoder — which never processed a mask token — transfers cleanly to tasks that present complete images. Note the unconventional topology relative to Chapter 4: the "hourglass" is not symmetric, and the bottleneck is not a dimensionality constraint but the *absence of 75% of the input*.

- Note also what is *not* in this count: the visible 49 patches never appear as targets, the mask token $\boldsymbol{q}$ is learned only through the decoder's predictions at masked positions, and the decoder's output at visible positions is computed but discarded. The whole 112,896-number pile of residuals is about content neither half of the network was shown. That is the pretext task in a single sentence.

5.3 The asymmetric design

Figure 5.2 already hints at the load imbalance between the two halves of MAE. Figure 5.6 makes it explicit, because this asymmetry is the design decision that turns masked image modeling from a nice idea into a *scalable* one — the lecture states the point bluntly: the encoder carries over 9× the computation per token of the decoder (Lecture 03, slide 30).

Three properties fall out of this design.

(i) Efficiency. Self-attention cost grows quadratically in sequence length, so feeding $(1-r)N$ instead of N tokens to the big encoder cuts its attention cost by a factor close to $1/(1-r)^2 = 16$ at $r = 0.75$. The cheap decoder restores sequence length wherever that length actually matters (the prediction needs position), at a fraction of the price. Without this trick, masked image modeling of large ViTs would be compute-prohibitive; with it, pretraining a ViT-H becomes feasible on ImageNet alone.

(ii) The encoder is deployable as-is. A subtle pitfall in BERT-style pretraining is a *distribution shift*: the encoder is trained on sequences containing `[MASK]` tokens that never appear downstream. MAE sidesteps this entirely — the encoder sees real image patches

during pretraining and real image patches during transfer, in both cases without any mask tokens. The asymmetry is therefore not just a speed hack; it removes a train/test mismatch.

(iii) The decoder is flexible. Because the decoder's only job is to generate a *loss*, its architecture is decoupled from the encoder's interface: it does not need the skip connections of a U-Net, does not need to be invertible, and can be widened, narrowed, or even replaced (linear decoder, single transformer block) without touching the encoder that will actually be used downstream. The slides emphasize this independence explicitly (slide 31). This is the "asymmetrical autoencoder design" — the word printed in red on slide 31 of the lecture.

Worked Example — The 16× attention saving, from first principles

Self-attention is the dominant term in a transformer block's compute profile, and its cost at layer ℓ scales quadratically in the sequence length n: each of the n tokens attends to all n tokens, so the attention matrix has n^2 entries. Suppose an image is patchified into $N = 196$ tokens and masked at $r = 0.75$.

- If mask tokens were *fed to the encoder* (the naive BERT-for-images design), the encoder's attention matrix would have $N^2 = 196^2 = 38{,}416$ pairwise interactions.
- MAE deletes mask positions from the encoder input, so the encoder's attention matrix has only $|\mathcal{V}|^2 = 49^2 = 2{,}401$ entries — an attention-layer saving of exactly $38{,}416/2{,}401 = 16$.
- The decoder restores the full 196-token sequence, so its attention term *is* quadratic in N — but each decoder token is cheap: the lecture puts the per-token computation gap at over 9× in the encoder's favor (slide 30), and the decoder's job ends at a single linear projection to $p^2C = 768$ predicted pixel values per patch.

The 16× figure is a property of the attention term only; once the linear-in-length terms (the feed-forward layers, the embeddings) are mixed back in, the end-to-end saving drops to several-fold. He et al. (2022) report total pretraining speedups of roughly 3× or more, and much lower memory use, from the asymmetric design — enough to make pretraining a ViT-Huge on ImageNet alone feasible on a standard academic cluster budget, which is why slide 29 calls the exclusion of mask tokens from the encoder the enabler of scale.

The same design, as a budget. The lecture's quiz (Q5 and Q17) frames it as spending a fixed compute budget. Every unit spent on the decoder goes into a part you will throw away; every unit spent on the encoder goes into the part you keep. Putting mask tokens into the encoder would spend *encoder* compute on placeholders that are identical everywhere and carry no information — buying nothing that survives pretraining. So the placeholders are handled only by the small decoder, where they cost little.

Section 5.6 shows the payoff: MAE pretrained on ImageNet *without labels* competes in Table 5.1 with methods that used labels or extra tokenizers, because almost all of its pretraining compute went into the part that is kept.

Why MAE is (almost) always a transformer. The whole recipe assumes the image is a *set of tokens* that you can simply leave out. A transformer handles that naturally: drop 75% of the tokens and it does at least 75% less work.

A convolutional network cannot. It slides its filters over a continuous grid, so a masked image is just an image with grey holes — the filters still run over the holes, and the speed-up disappears. There is also no natural convolutional equivalent of a "mask token". That is why masked image modelling and Vision Transformers arrived together and stayed together.

Simplified intuition vs. more precise view

Simplified story. "MAE hides 75% of the image, guesses the rest, and the guessing teaches the encoder vision."

More precise view. The guessing itself is *never evaluated as generation.* MAE minimizes a pixel-level mean-squared error, and — as the same phenomenon in Chapter 4's VAE decoder showed — squared-error losses predict conditional *means*, which is why all MAE reconstructions are blurrier than real images. What is being learned is not a pixel-perfect generative model but a set of encoder features of which a small linear layer (on top of a shallow decoder) can linearly read out missing low-level content. Being forced to enable that readout, at high masking ratios, *requires* the features to encode scene layout, object shape, and texture statistics. The generation is scaffolding; the learned conditional structure is the building.

Paper Analysis: Masked Autoencoders Are Scalable Vision Learners (He et al., 2022)

Problem. Masked language modeling made BERT-scale pretraining canonical in NLP by 2019 (Devlin et al., 2019), but the analogous recipe for images lagged for two disjoint reasons: image redundancies make low masking ratios degenerate into local interpolation, and full-sequence encoders make high masking ratios compute-prohibitive at ViT-L/H scale.

Idea. Solve both problems with one asymmetry: mask 75% of the patches, run the large encoder only on the visible 25%, then let a lightweight decoder reconstruct the missing pixels. The encoder is the product; the decoder is scaffolding.

Evidence. ImageNet fine-tuning: 84.9% top-1 at ViT-L (vs. 82.5% from scratch; slide 36). Transfer: COCO detection/segmentation and ADE20K semantic segmentation results in Table 5.1; the curves of Figure 5.9 show fine-tuning is robust to masking ratio while linear probing demands high masking. At larger scale the same recipe reaches 87.8% top-1 with ViT-H on ImageNet-1K — the number that made the "scalable" of the title concrete.

One ablation to remember. Excluding mask tokens from the *encoder*: this single move is what buys the 3×-or-more wall-clock speedup and the memory saving that let the recipe scale. The pixel target, the 75% ratio, and the discarded decoder all matter, but the asymmetry is the load-bearing wall — remove it and the whole design folds back into the compute regime that made masked image modeling impractical before He et al. (2022).

5.4 What to reconstruct: pixels, not tokens

MAE is not the first attempt at BERT-style pretraining for images; it is the one that made the simplest possible target work. Understanding *what the decoder is asked to predict* is the difference between MAE and its direct predecessor BEiT (Bao et al., 2022), and Figure 5.7 lays the two side by side.

5.4.1 BEiT: discrete visual tokens

BEiT: BERT for images, taken literally. BEiT (Bao et al., 2022) came out months before MAE and copies BERT as closely as possible. The difficulty: text already comes as a list of words from a fixed vocabulary; images do not. So BEiT first *creates* a vocabulary:

1. A separately trained discrete VAE (the one from DALL-E) turns each image patch into one of a fixed set of *visual words*.
2. A transformer sees the image with some patches hidden, and for each hidden patch predicts *which visual word* it was — a classification with a softmax and cross-entropy, exactly as BERT predicts a hidden word.

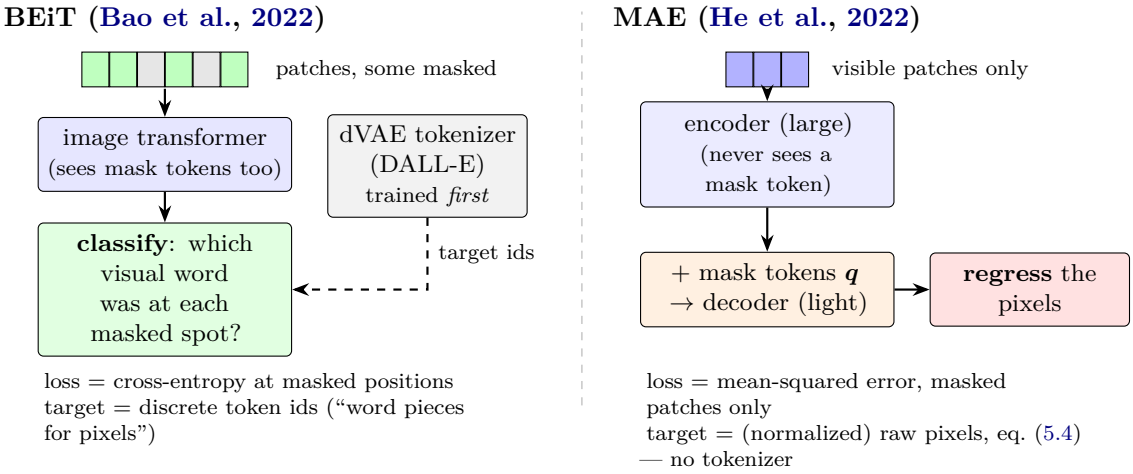


Figure 5.7. Two answers to "what should a masked image model predict?" **Left:** BEiT (Bao et al., 2022) transfers the BERT recipe literally — it first trains a discrete VAE tokenizer (from DALL-E) that turns patches into a vocabulary of visual tokens, then trains a transformer to *classify* each masked position's token id. **Right:** MAE (He et al., 2022) skips the tokenizer entirely: the decoder *regresses* raw (per-patch normalized) pixel values at masked positions, and the encoder sees visible tokens only. MAE's choice removes a whole pretrained subsystem and, paired with the asymmetric encoder, is what makes the method cheap enough to scale.

The lecture's table lists BEiT's pretraining data as "IN1K+DALLE" (slide 36): the tokenizer is another trained model, with its own data needs.

The real question: what should a masked patch be predicted *as*? This comparison is the first place where the choice of *target* matters.

- **BEiT: predict a codeword.** Each patch is first turned into one of 8192 discrete "visual words" by a separate pretrained network (a dVAE, itself trained on DALL-E data). The masked patch is then predicted by classification, exactly like BERT predicts a masked word. Cost: a whole second model has to exist first.
- **MAE: predict the pixels.** No vocabulary, no second model, no classifier — just regress the raw pixel values.

The lecture's transfer table shows the simpler option is at least as good. And the choice of target is a deep design decision in its own right: it decides what "a correct guess" means.

Common Confusion

MAE vs. contrastive learning: what is a "target"? In the contrastive methods of Chapter 9 — MoCo, SimCLR and their descendants — the target is *another view of the image*: the representation is asked to be invariant to whatever augmentation produced that view, and the loss is a softmax contrast over a batch of other images. In MAE the target is *content*, not identity: the decoder must produce the actual missing pixels,

comparing position by position, and no other image is involved. This difference is not cosmetic. Contrastive invariance is what makes those features linearly separable. MAE makes no invariance demand at all — and it shows: its linear-probe numbers are far below the contrastive state of the art (subsection 5.7.3), while its fine-tuned numbers match or beat it. "Masked modeling" and "contrastive learning" both pretrain vision transformers, but they answer different questions about what makes two representations equal. Neither pretext is a special case of the other.

5.4.2 The pixel target, concretely: per-patch normalization

The MAE paper studies several reconstruction targets; the default that survives is each masked patch's pixels, *normalized by that patch's own mean and standard deviation*. For masked patch i with raw pixel vector $\boldsymbol{x}_i \in \mathbb{R}^{p^2C}$, let

$$\mu_i = \frac{1}{p^2C}\sum_{d=1}^{p^2C} x_{i,d}, \qquad \sigma_i = \sqrt{\frac{1}{p^2C}\sum_{d=1}^{p^2C}(x_{i,d}-\mu_i)^2}\,, \tag{5.3}$$

and define the target in (5.2) component-wise as

$$\bar{x}_{i,d} = \frac{x_{i,d}-\mu_i}{\sigma_i}\,, \qquad d = 1,\ldots,p^2C, \tag{5.4}$$

where d indexes the individual pixels (and channels) of patch i. Two motivations: first, normalized patches put every patch target on the same scale regardless of local brightness, which stabilizes the squared-error loss; second, because each target has its patch's own brightness and contrast *removed*, the decoder cannot earn loss reduction just by getting the local brightness right; the error is spent on the patch's internal structure (edges, texture, shape), which nudges the encoder away from global lighting. The paper finds normalized pixels slightly better than raw pixels on downstream fine-tuning, and about as good as (slightly better than) BEiT-style dVAE tokens — with no tokenizer at all (He et al., 2022).

Figure 5.8 computes (5.4) on a toy 2×2 patch so the arithmetic is checkable by hand.

Simplified intuition vs. more precise view

Simplified story. "MAE predicts pixel values; BEiT predicts categories."
More precise view. A per-patch softmax over a learned tokenizer code is still a model of the *conditional distribution* of patches, with a built-in inductive bias that similar patches collapse to one token; pixel regression with squared error is a conditional *mean*, which by construction blurs over uncertainty (the same averaging that makes Chapter 4's VAE samples soft). The distinction matters where uncertainty is irreducible: at high masking, many completions are plausible, and the L2 target deliberately averages them. Tokenized classification, in contrast, can spread mass over several discrete modes ("dog patch *or*

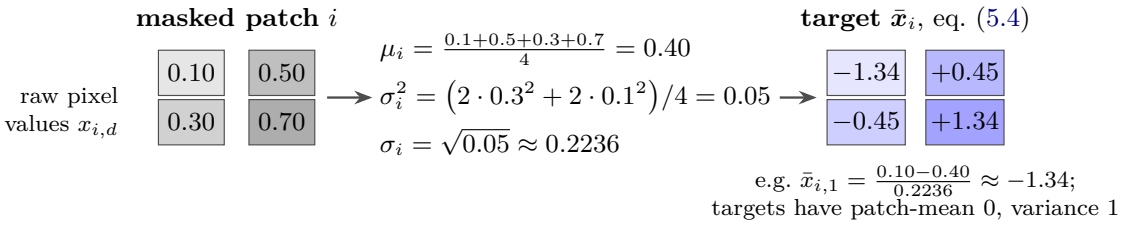


Figure 5.8. Per-patch normalization on a toy 2×2 patch ($d = 1, \ldots, 4$; a one-channel patch for readability — the real patch has $p^2C = 768$ numbers). The patch mean $\mu_i = 0.40$ and standard deviation $\sigma_i \approx 0.2236$ are computed from the patch's own pixels by (5.3); the decoder's target $\bar{\boldsymbol{x}}_i$ is each raw value standardized by (5.4). The squared error in (5.2) is then measured in units of *patch contrast*, not absolute brightness.

cat patch"), which is one reason BEiT's objectives resemble BERT's so faithfully. The pixel target is not the statistically richer model — it is the *cheaper* and, empirically, better-transferring teacher signal.

5.4.3 An MAE training step in code

The pipeline of Section 5.2 compresses into about thirty lines of idiomatic PyTorch. The listing below is faithful to the official implementation structure; the annotations point each block back at the abstraction it implements.

```python
import torch
import torch.nn as nn

class MaskedAutoencoder(nn.Module):
    def __init__(self, encoder, decoder, patch_size=16,
                 embed_dim=1024, mask_ratio=0.75):
        super().__init__()
        self.patch_size = patch_size
        self.mask_ratio = mask_ratio
        self.encoder = encoder        # deep ViT; sees visible tokens only
        self.decoder = decoder        # lightweight; sees full sequence
        # one shared learned placeholder for every masked position
        self.mask_token = nn.Parameter(torch.zeros(1, 1, embed_dim))

    def patchify(self, imgs):
        # (B, C, H, W) -> (B, N, p*p*C); N = (H/p)*(W/p)
        p = self.patch_size
        h, w = imgs.shape[2] // p, imgs.shape[3] // p
        x = imgs.reshape(imgs.shape[0], 3, h, p, w, p)
        x = torch.einsum('nchpwq->nhwpqc', x)
        return x.reshape(imgs.shape[0], h * w, p**2 * 3)

    def random_masking(self, x):
        B, N, L = x.shape
        len_keep = int(N * (1 - self.mask_ratio))  # visible 25%
        noise = torch.rand(B, N, device=x.device)
        ids_shuffle = torch.argsort(noise, dim=1)
        ids_restore = torch.argsort(ids_shuffle, dim=1)
        ids_keep = ids_shuffle[:, :len_keep]
        x_keep = torch.gather(
            x, dim=1,
            index=ids_keep.unsqueeze(-1).repeat(1, 1, L))
        # binary mask: 1 where the original patch is masked
        mask = torch.ones(B, N, device=x.device)
        mask[:, :len_keep] = 0
        mask = torch.gather(mask, dim=1, index=ids_restore)
        return x_keep, mask, ids_restore

    def forward(self, imgs):
        patches = self.patchify(imgs)                # step 1 (slide 25)
        x_keep, mask, ids_restore = self.random_masking(patches)
        latent = self.encoder(x_keep)                # step 2 (slide 26)
        B, N, _ = patches.shape
        pad = self.mask_token.repeat(
            B, N - latent.shape[1], 1)               # step 3 (slide 27)
        x_all = torch.cat([latent, pad], dim=1)
        x_all = torch.gather(
            x_all, dim=1,
            index=ids_restore.unsqueeze(-1).repeat(1, 1, latent.shape[-1]))
        pred = self.decoder(x_all)                   # step 4 (slide 28)
        # masked-patch-only mean-squared error; see the loss equation
        # L = (1/|M|) * sum_{i in M} |pred_i - target_i|^2
        target = patches
        loss = (pred - target) ** 2
        loss = loss.sum(-1)                          # per-patch squared L2
        loss = (loss * mask).sum() / mask.sum()      # masked only
        return loss
```

Listing 5.1. One pretraining step of MAE, mirroring the four stages of Figure 5.2: patchify, mask, encode visible positions only, re-expand with mask tokens, decode, and supervise only the masked predictions. The `ids_shuffle`/`ids_restore` bookkeeping is how the decoder's output gets mapped back to the original spatial order for the loss.

Three details in the code that the slides skip:

1. **How patches are chosen.** Each patch gets a random number, and the lowest 25% are kept. That gives a genuinely uniform random subset, with no spatial pattern.
2. **The encoder never sees a mask token.** It is called only on the visible patches, `self.encoder(x_keep)`.
3. **Only masked patches are scored.** The loss line multiplies by `mask` and divides by `mask.sum()`: visible patches contribute nothing. The network gets no credit for copying what it could already see.

5.5 Why does this actually work?

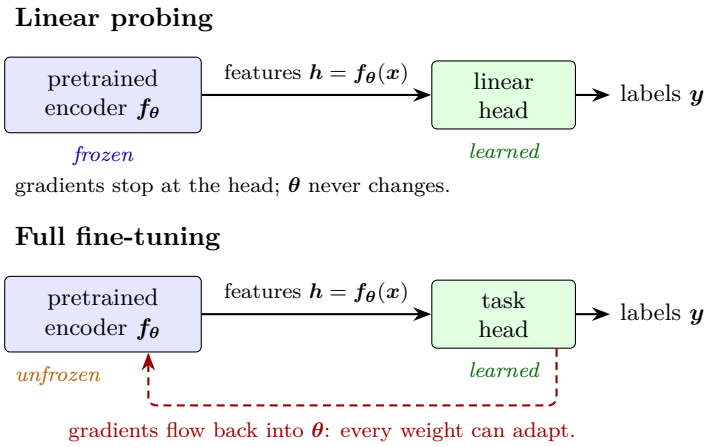
Three candidate accounts, none of them wholly settled. All three lean on the same family of collapse-avoidance arguments, and the two collapse mechanisms are worth stating separately before we attribute any specific representation to them.

Common Confusion

Two distinct shortcuts the recipe heads off. The *identity-mapping collapse* is what Chapter 4's autoencoders suffer from in limitless-capacity regimes: if the encoder is ever shown the content it is asked to reconstruct, a sufficiently large model learns to copy rather than to understand. MAE prevents this structurally, not by regularization — *the encoder is never shown the masked pixels*, so there is nothing to copy through. The masking ratio is the choke. The second shortcut is *constant output*: a decoder that ignores the visible context and always predicts the dataset-average patch. It never reaches low loss, because the targets differ from image to image and position to position — a constant predictor pays a high MSE on every masked position. And because the sum in (5.2) runs over $\mathcal{M}$ only, there are no easy visible positions where copying could earn cheap loss reduction instead. A prediction that varies with the visible context is forced. Neither safeguard is a supplemental trick; each falls out of a decision (masking, masked-only loss) that also serves the compute story.

First, *masking as corruption* is the closest modern relative of the denoising autoencoder from Chapter 7: a denoising task forces the encoder to learn the image manifold well enough to move off-manifold corruptions back onto it. Masking can hence be read as "denoising with 75% of the input replaced by nothing at all"; Chapter 7 makes this connection explicit when it treats diffusion as iterated denoising.

Second, *a high masking ratio removes the shortcut.* Chapter 4's autoencoders failed when the network had enough capacity to memorise instead of learn. Hiding three quarters of the input fixes that structurally: nothing can copy what was never shown. The only way to lower the loss is to learn how the world works — fur continues as fur, sky continues as sky, objects have shapes. With a low masking ratio, a gap can be filled from its immediate neighbours.



With a high one, the model must relate distant parts of the same object — the flamingo's beak to its tail — which is the kind of understanding we want.

Third, *the task is already semantic.* Because the target is low-level (pixels) but the task at 75% masking is not solvable by low-level cues alone, the gradients flowing into the encoder have to reward whatever internal features encode "what is depicted". The encoder ends up encoding high-level content not because that was asked for directly, but because it is the most economical way to enable the decoder's job. The same mechanism predicts the linear-probe/fine-tune gap measured in Figure 5.9: the features are there, they are just not linearly arranged.

5.6 What MAE is for: representation learning

5.6.1 Pretrain, then adapt

Lecture 03 devotes its last act to re-framing MAE away from generation (slides 36–39): the decoder is discarded, the *encoder* is the product, and the question becomes how well its representations transfer. The two-stage logic — pretrain on an unlabeled task, then adapt to a labeled one — is the same pretrain/finetune paradigm the slides draw as two clouds of images feeding one network (slide 37): first a large unlabeled cloud (ImageNet *without* labels) and the MAE loss; then the task you actually care about.

Key Concept — Linear probing vs. full fine-tuning

Linear probing freezes the pretrained encoder and trains only a single linear classifier on top of its features. *Full fine-tuning* continues training the encoder itself (plus any task-specific layers) on the downstream task. The first is a diagnostic: it asks how much task-relevant information the representation *already exposes linearly.* The second is a practical measurement: it asks how well the representation serves as a *starting point* once all parameters are allowed to move. A method can excel at one and lag at the other —

the contrastive methods of Chapter 9 win linear probing; MAE wins fine-tuning (slide 39 shows a roughly 11–point gap between the two protocols at their peaks, exactly because MAE features are not linearly separated out of the box but are an excellent initialization).

5.7 Headline results

5.7.1 ImageNet classification

The headline result (slide 36). A ViT-L trained *from scratch* on ImageNet (Deng et al., 2009) reaches 82.5% top-1. The same network, first pretrained with MAE and then fine-tuned, reaches **84.9**% — a gain of 2.4 points, with no extra data. The pretraining used ImageNet *without its labels*, and — unlike BEiT — without an extra tokenizer.

It keeps improving with size: a ViT-H pretrained with MAE and fine-tuned on ImageNet-1K reaches 87.8%, well above the same network trained from scratch (He et al., 2022). That trend — bigger encoder, better result — is the "scalable" in the paper's title. It gave vision what language models already had: pretrain one very large encoder on unlabelled data, and start everything else from there.

5.7.2 Transfer to detection and segmentation

Table 5.1 reproduces the lecture's transfer results (slide 36; Tables 4–5 of He et al. (2022)). How to read it:

- **COCO** (Lin et al., 2014): the pretrained ViT is plugged into a Mask R-CNN object detector. AP^{box} scores the boxes, AP^{mask} the object outlines. Higher is better.
- **ADE20K** (Zhou et al., 2017): semantic segmentation with a UperNet head, scored by mIoU (how well predicted and true regions overlap). Higher is better.
- In each cell, the first number is ViT-B and the second ViT-L.
- "IN1K" means pretrained on ImageNet-1K images; the self-supervised rows never use its labels.

Reading the table row by row: MAE, pretrained on nothing but unlabeled ImageNet, beats supervised IN1K pretraining by 2.4 AP^{box} at ViT-B (50.3 vs. 47.9) and 4.0 at ViT-L scale (53.3 vs. 49.3) on COCO detection; it beats the contrastive MoCo v3 across the board; and it is competitive with BEiT despite BEiT's extra tokenizer pretraining — BEiT trains a whole second model just to decide what the targets are. The ADE20K segmentation column tells the same story: 48.1 mIoU (ViT-B) and 53.6 mIoU (ViT-L), the best in the table at both scales.

Table 5.1. Transfer results as reported in the lecture (slide 36; Tables 4–5 of He et al. (2022)). Each cell lists ViT-B / ViT-L. Self-supervised entries pretrain on ImageNet-1K *without* labels; BEiT additionally uses DALL-E dVAE tokenizer training. Bold marks entries the slide highlights as best (with ties left bold on both sides).

	COCO (Mask R-CNN)		ADE20K (UperNet)
pretraining	AP^{box}	AP^{mask}	mIoU
supervised IN1K (with labels)	47.9 / 49.3	42.9 / 43.9	47.4 / 49.9
MoCo v3, IN1K (Chen et al., 2021)	47.9 / 49.3	42.7 / 44.0	47.3 / 49.1
BEiT, IN1K+DALLE (Bao et al., 2022)	49.8 / **53.3**	44.4 / 47.1	47.1 / 53.3
MAE, IN1K (He et al., 2022)	**50.3** / **53.3**	**44.9** / **47.2**	**48.1** / **53.6**

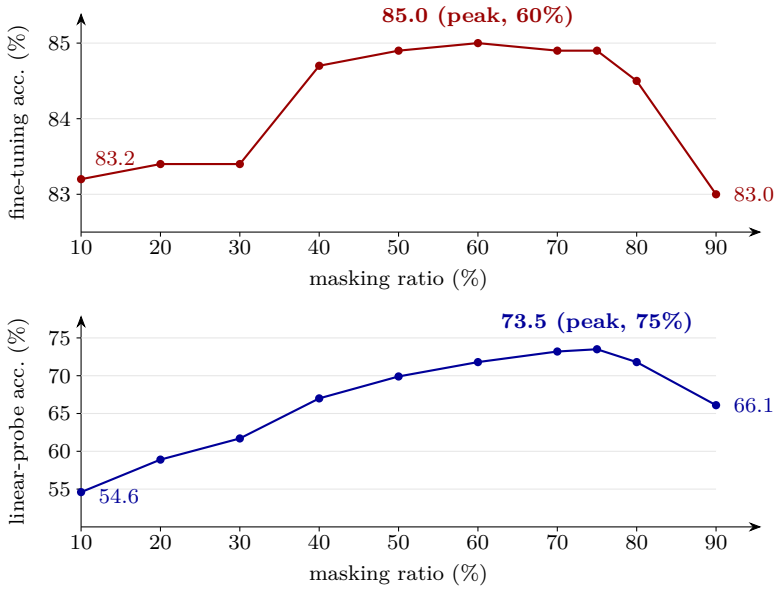


Figure 5.9. ImageNet accuracy vs. masking ratio, as plotted on the lecture's final slide (Lecture 03, slide 39; the ten data-point values of each curve, including one at 75%, are copied from the slide; lines connect them). **Top, fine-tuning:** a broad, flat plateau — 83.2, 83.4, 83.4, 84.7, 84.9, **85.0**, 84.9, 84.9, 84.5 across 10–80%, falling to 83.0 at 90%; the **peak is at 60% masking**. **Bottom, linear probing:** a steep rise from 54.6 at 10% through 67.0 at 40% to a high plateau of 73.2–73.5 at 70–75% — the slide's highest measured point is **73.5** — then a decline to 71.8 (near 80%) and 66.1 at 90%. The protocols disagree about the best masking ratio (60 vs. ∼70–75) and, more sharply, about how much the ratio matters at all: the fine-tuning curve spans barely two points top to bottom, the linear-probe curve nearly nineteen.

5.7.3 The masking ratio, one more time

The lecture closes on an ablation rather than a victory lap (slide 39): sweep the masking ratio from 10% to 90%, pretrained each time, then evaluate *both* protocols. Figure 5.9 reproduces the two curves faithfully from the slide's ten data points each.

The two curves are the chapter's empirical punchline, and they deserve a slow reading. Under fine-tuning MAE is astonishingly tolerant: anywhere between roughly 40% and 75% masking is near-optimal, and even 90% masking costs only about two points. Once the task is hard enough, it stays hard enough. Under linear probing the picture is completely different: accuracy climbs steeply with the masking ratio, plateaus at 73.2–73.5 around 70–75% masking, then drops sharply at 90%. The persistent *gap* between the curves — about 11.5 points between their peaks (85.0 vs. 73.5), more elsewhere — is the signature, in data, of the distinction from the key concept: MAE features are highly informative but not linearly separated, and they unlock under fine-tuning. The lecture states the two protocols' roles in matching terms (slide 38): linear probing "measures representation quality under restricted conditions"; fine-tuning "exploits the model's near-true potential to adopt for new tasks."

Common Confusion

"MAE learns worse features than contrastive methods" — under which ruler? It is tempting to read the 73.5% linear-probe peak of Figure 5.9 and conclude that MAE is simply inferior to MoCo v3-style contrastive pretraining (Chen et al., 2021), whose frozen features are far stronger. That reading confuses two different measurements of the same object. Frozen-feature evaluation (linear probe, kNN) measures what the representation makes *linearly available*; full fine-tuning measures what the representation can *become*. By the second ruler MAE wins the transfer comparisons of Table 5.1. Neither is the "true" quality — they answer different deployment questions. If you will fine-tune (detection, segmentation, any end-to-end task), MAE's profile is the attractive one; if the encoder must stay frozen (retrieval, cheap linear heads), a contrastive or distilled backbone of Chapters 9 or 11 is the safer choice.

Historical Note

The conceptual lineage of masking runs through denoising, not only through BERT. Inpainting and context encoders (Pathak et al., 2016) had already argued that predicting removed image regions teaches scene structure; BERT (Devlin et al., 2019) made the corruption discrete and bidirectional in language; BEiT (Bao et al., 2022) freighted the recipe with a learned image vocabulary to make it literally BERT-like; MAE stripped it back to pixels, removed the mask tokens from the encoder, and made it cheap enough to scale. Each step deleted a component — the bottleneck dimension, the tokenizer, the symmetric hourglass of Chapter 4 — rather than adding one. The lecture's own closing slide underlines the lesson: the last word of Lecture 03 is not a qualitative demo but the two evaluation curves of Figure 5.9, as if to say that the "deep" contribution of MAE is measured, not pictured.

5.8 Failure modes and boundaries

Failure Modes

(i) Small ViTs gain less. MAE's advantage over supervised and contrastive pretraining grows with scale (He et al., 2022): the lectures and the paper concentrate on ViT-L/H, and the lead shrinks for smaller models — in Table 5.1 MAE's COCO box-AP lead over supervised pretraining is 4.0 points at ViT-L but only 2.4 at ViT-B, and below ViT-B the paper gives no evidence that it survives. If your budget buys a small model, MAE is not automatically the right pretraining recipe.

(ii) The fine-tune/linear-probe gap cuts both ways. MAE's frozen features are conspicuously weak: Figure 5.9's bottom curve peaks at 73.5% while contrastive ViTs exceed that comfortably under the same frozen evaluation. Two practical hazards follow. Anyone *benchmarking* an MAE backbone with a linear head only — a common convention — dramatically underestimates it; and anyone *choosing* a backbone for genuinely frozen use (kNN retrieval, semantic indexing, edge deployment) may prefer a contrastive model and should not be seduced by the 84.9%/fine-tune numbers.

(iii) What MAE does not learn: invariance. Nothing in (5.2) asks the encoder to give the *same* representation to two augmented views of one image. MAE features therefore carry no built-in invariance to crops, colors, or viewpoints — the property the whole contrastive family of Chapter 9 optimizes directly — and no emergent part-level attention organization of the kind self-distilled ViTs exhibit (Chapter 11). MAE learns the *conditional structure* of images — what is likely to be where, given what is visible — which is a different competence from invariance, and tasks that lean on invariance in frozen features will not be well served by it.

(iv) Long pretraining is part of the recipe. He et al. (2022) report that accuracy improves with pretraining budgets of hundreds to thousands of epochs; the gains cited in this chapter assume those schedules. Short-schedule MAE is markedly less compelling.

(v) The decoder is not a product. Because the pixel loss is a conditional mean, reading out the decoder's predictions yields the blur visible in the lecture's reconstruction galleries (slides 33–35), which also repeat high-frequency mistakes. MAE is pretraining machinery whose generated images are diagnostic artifacts, not deliverables — photorealistic generation belongs to the diffusion chapter (Chapter 7).

(vi) The recipe assumes a transformer-shaped backbone. The masking dividend of Section 5.3 is paid in units of *sequence length*: the encoder is fast at 25% tokens because attention cost is quadratic in the number of tokens and it receives a quarter of them. A convolutional backbone has no such lever: convolutions run over the full spatial lattice regardless of which cells you have conceptually "masked," and kernels cannot skip empty cells without bespoke sparse machinery. The same is the reason the mask-token device exists in MAE at all — a Transformer consumes a set of tokens, so a placeholder can simply be included at position i — whereas a CNN has no token vocabulary in which to

write one. Mask-and-reconstruct is a generative pretext analogue of Chapter 7's denoising autoencoder, but its *compute* story is a Transformer story.

5.9 Beyond still images

Research Frontier

Masked autoencoding leaves the image domain. The recipe — mask aggressively, encode only the visible remainder, decode cheaply — transfers to any modality whose raw signal is strongly redundant along chosen axes. Two active directions to be aware of (qualitative pointers, not verified results): *video MAE* work masks spatio-temporal "tubes" of video rather than spatial patches and pushes masking ratios even higher (around 90%), because time adds another axis of redundancy on top of the spatial ones; *audio MAE* work applies the same design to audio spectrograms, masking time–frequency patches and reconstructing the masked spectrogram content. The family resemblance is the thing to take away: each of these variants re-runs the same decision sequence you have now walked through for images — what is a patch? what ratio makes the task hard but solvable? what does the decoder regress, and with what normalization? — which is why the design vocabulary of this chapter has become a portable template rather than a one-off architecture.

5.10 Check Your Understanding

Check Your Understanding

Level 1 (recall).

1.1 What fraction of patches does MAE mask by default, and what does the decoder regress at masked positions?

1.2 Name the two asymmetries in MAE's encoder–decoder design and the quantitative claim the slides make about their per-token cost gap.

1.3 What is the shared mask token $\boldsymbol{q}$, and at which stage of the pipeline does it enter?

Level 2 (mechanism).

2.1 A 384×384 image is split into 16×16 patches and masked at $r = 0.75$. How many patches reach the encoder, and how many scalar values does the decoder output for the masked patches?

2.2 Why does MAE add positional embeddings *twice*, once into the encoder input and again into the decoder input?

2.3 In (5.2), why is the sum over $i \in \mathcal{M}$ rather than over all N patches?

Level 3 (design).

3.1 Why is a 15% masking ratio sensible for text but wasteful for images? Tie your answer to the shape of the curves in Figure 5.9.

3.2 Colleague A proposes swapping MAE's pixel target for a BEiT-style tokenizer while keeping the asymmetric encoder; colleague B proposes restoring a symmetric BEiT-style encoder while keeping pixels. Which change sacrifices more of MAE's compute advantage, and why?

3.3 Suppose per-patch normalization (5.4) is dropped and raw pixels are the target. Describe one shortcut the decoder could now exploit that normalization forecloses.

Level 4 (analysis).

4.1 The linear-probe curve in Figure 5.9 collapses at 90% masking while fine-tuning only falls about two points. Give two distinct hypotheses for this asymmetry, and one extra experiment that would discriminate between them.

4.2 MAE pretraining never compares two images; contrastive pretraining always does. Explain why this difference alone predicts MAE's weak linear-probe/kNN profile, and name a downstream setting from this chapter where the disadvantage should essentially vanish.

4.3 Using the toy patch of Figure 5.8, suppose the decoder outputs $\hat{\boldsymbol{x}}_i = (-1.0, 0.5, -0.5, 1.0)$. Compute its contribution to $\mathcal{L}_{\text{MAE}}$ assuming i is the only masked patch.

Level 5 (synthesis).

5.1 Chapter 7 frames MAE as a special case of denoising. Make the correspondence precise — what plays the role of the corruption, and what of the noise level? — and name one respect in which the analogy breaks down.

5.2 Design a masked-autoencoding pretraining for a robot's proprioceptive time series (Chapter 14 asks whether such signal is "labels-free"; here answer only the masking questions): What is a "patch"? What masking ratio would you start from, and why? What replaces the per-pixel decoder?

Solution sketches

1.1. 75%; the (per-patch normalized) pixel values of the masked patches, eqs. (5.2)–(5.4). **1.2.** The encoder sees only the visible tokens (25% at $r = 0.75$), and it is much larger than the decoder: the slides state the encoder carries over 9× the decoder's computation per token (slide 30). **1.3.** $\boldsymbol{q}$ is a single learned d-dimensional parameter vector, shared across all masked positions of all images; it enters at the *decoder* input, after the visible tokens have been encoded and unshuffled back into spatial order. **2.1.** $N = (384/16)^2 = 576$ patches; visible = 144; masked = 432; each masked-patch prediction has $16 \cdot 16 \cdot 3 = 768$ scalars, so $432 \times 768 = 331{,}776$ outputs. **2.2.** The two modules use position for different jobs: the encoder must know *which observed content* it is encoding (its output tokens should remember

where their evidence came from); the decoder, whose mask tokens $\boldsymbol{q}$ are content-free, must know *where* each prediction has to land in the grid. **2.3.** Visible-patch tokens are encoder outputs the decoder received for free; supervising them rewards copying, spends decoder capacity on an easy sub-task, and measures nothing about reconstructing unseen content. **3.1.** Text carries high semantic density per token — at 75% masking the residue is not a sentence, so the task becomes unlearnable; images are spatially redundant — at 15% masking the missing patches are interpolated from neighbors, an uninformative task landing in the low-accuracy left end of the linear-probe curve in Figure 5.9. **3.2.** B sacrifices more. The compute advantage lives in keeping the *big* encoder off most of the sequence (quadratic attention over $N/4$ vs. N wide tokens); A's tokenizer adds a training stage but leaves the asymmetry intact, so most of the speed survives. **3.3.** Brightness/exposure degrees of freedom become always-easy targets: the decoder can fit absolute levels with local constants, letting the encoder spend capacity on lighting instead of content. Per-patch normalization erases exactly that degree of freedom from the target. **4.1.** H1 — *information*: 90% masking leaves too little visible evidence to pin down object identity, so frozen features lose linearly-decodable class information. H2 — *optimization*: the task is so hard that training stalls in a worse basin. Discriminator: at 90% masking, test whether reconstructions are confident-but-wrong (favors H1) or noisy/underfit — e.g. train longer or with a wider decoder and check whether frozen-probe accuracy recovers (H2) or stays put (H1). **4.2.** Linear probe and kNN read frozen geometry; nothing in MAE's loss pushes different augmentations of one image together, so nuisance directions (pose, illumination, crop) survive in the representation and a linear read-out mixes them with class signal. Fine-tuning rewires those directions within a few epochs, so detection and segmentation (Table 5.1) see essentially no penalty. **4.3.** Errors: $(-1.00-(-1.34),\ 0.50-0.45,\ -0.50-(-0.45),\ 1.00-1.34) = (0.34, 0.05, -0.05, -0.34)$; squared sum $= 0.1156+0.0025+0.0025+0.1156 = 0.2362$; one masked patch, so $\mathcal{L}_{\text{MAE}} = 0.2362$. **5.1.** MAE = denoising autoencoder with corruption "zero out 75% of the input" — the masking ratio plays the role of the noise level, the visible patches are the observed signal, the decoder is the denoiser. Breakdown: the diffusion family of Chapter 7 uses *continuous* Gaussian noise at *many levels* and iterates the denoiser to sample; MAE's corruption is binary, applied once, and its training stops at a deterministic conditional mean rather than learning a score or a sampler. **5.2.** Patch = short time window (possibly per joint/channel) of the trajectory. Because 1-D proprioception is even more redundant along time than images are along space, start from a *higher* ratio (cf. the $\sim 90\%$ of video MAE in the Research Frontier box) and sweep. The decoder regresses per-window normalized sensor readings — the analogue of (5.4) with mean/variance over the window — and the same heavy-encoder/light-decoder asymmetry applies verbatim.

5.11 Concept Map

Concept Map — how this chapter connects

Backward. *Chapter 4 (Ch. 4, autoencoders and VAEs)* supplied the hourglass, the reconstruction loss, and — through the Gaussian-observation argument — the reason a pixel MSE target yields blurred conditional means rather than sharp samples; the codebook memorization failure there is precisely the shortcut that 75% masking rules out by construction here. *Chapter 1* defined the pretext-vs-downstream distinction and the evaluation protocols that Section 5.6 operationalizes.

Sideways. *Chapter 8 (pretext tasks)* places masked image modeling in the corruption–reconstruction family alongside inpainting and context encoders (Pathak et al., 2016). *Chapter 9 (contrastive learning)* is the rival account of what a representation should be — invariance to augmentation vs. the conditional pixel structure learned here; the complementary linear-probe/fine-tune profiles of the two families are the running case study of *Chapter 16*.

Forward. *Chapter 7 (Ch. 7, diffusion and denoising)* is the closest cousin in this book: MAE is denoising with a binary 75% mask where diffusion uses continuous noise at a continuum of levels, and reading the two against each other sharpens what "generative pretext" means across the book. *Chapter 11 (DINO and self-distillation)* takes the masking idea into the distillation family — iBOT combines MAE-style masking with DINO-style self-distillation — and supplies the emergent-attention properties this chapter noted MAE lacks. *Chapters 12–14* reuse the pretrain-then-adapt logic of Section 5.6 in distillation, world-model, and robotics settings.

CHAPTER 6

Generative Models as SSL I: KDE, Flows, GANs

They sample from a learned P, / A distribution—structured, free. / Each token comes conditionally, / On all the ones that used to be.

Poem produced by a language model in response to the lecture's on-slide prompt (Lecture 04, slide 41)

In this chapter

6.1 Modeling $p(X)$ Is Self-Supervision 206
6.2 Kernel Density Estimation: The Simplest Generative Model . . 207 • Problem: from points to distributions • From points to densities: the estimator • Worked example: five points in the plane • KDE generates—badly
6.3 Normalizing Flows: Exact Likelihoods by Inversion 216 • Replacing the encoder–decoder pair with one bijection • Change of variables: the mathematics of flows • Composing many flows • The design constraint: triangular Jacobians • NICE: the first coupling flow • Real NVP: affine coupling layers • Multi-scale architectures: factoring out latents • Glow: invertible 1×1 convolutions • MAF: letting each coordinate see all of its predecessors • IAF: the same equation, solved the other way round • The five flows side by side • Conditional flows and super-resolution • Where flows fail
6.4 Generative Adversarial Networks: From Likelihood to Game . . . . 259 • Setup: a generator and a critic • Reading the minimax game one player at a time • Saturating vs. non-saturating loss: a look at the gradients • Training dynamics and failure modes • Why GAN training is hard, and what fixes each problem • Before StyleGAN: DCGAN and progressive growing • StyleGAN: a generator that composes styles • StyleGAN, one component at a time • StyleGAN2 and StyleGAN3: fixing the artifacts • BigGAN: the same game at ImageNet scale • The StyleGAN family and its relatives, side by side • Conditional GANs: telling the generator what to make • Pix2Pix: a conditional GAN for image-to-image translation • Why generative models matter beyond the course
6.5 Taxonomy of Generative Models 303
6.6 Summary 303

Chapter roadmap

Before you start. You need the autoencoder and VAE of Chapter 4 — this whole chapter is a reply to the VAE's weaknesses — plus the Gaussian distribution and the idea of a determinant as "how much a linear map scales volume" (Appendix A.2 if that is unfamiliar). You do not need measure theory.

The question this chapter answers. Chapter 4 built a model that can generate, but can only put a *bound* on the probability it assigns to data. Can we have the real number — the exact likelihood — and still generate? Three answers follow, in increasing order of how much they give up.

The three families, and what each trades away.

- *Kernel density estimation*: exact, trivial, needs no training at all — and useless above a handful of dimensions. Read it as the baseline that shows why the rest of the chapter is hard.
- *Normalizing flows*: exact likelihood, exact sampling and exact inference, paid for with a severe restriction on what layers you are allowed to build. Five models are developed here in full: **NICE**, **Real NVP**, **Glow**, **MAF** and **IAF**. Each changes exactly one thing about the one before it, and the chapter is arranged so you can see what.
- *Generative adversarial networks*: total architectural freedom and sharp samples, paid for by abandoning likelihood entirely in favour of a two-player game that may never converge.

Afterwards you will be able to state the change-of-variables formula and say in plain words what each term does; draw a coupling layer and invert it on paper; explain why MAF is fast to train and slow to sample while IAF is the exact opposite, and why that is not a coincidence; and place any generative model you meet onto the taxonomy of Section 6.5.

One promise about notation. Two directions get confused constantly. Throughout this chapter g maps **data to latent** (the *normalizing* direction, used for training) and g^{-1} maps **latent to data** (the *generating* direction, used for sampling). Whenever a formula appears, check which direction it is in before anything else.

The last two chapters learned by *rebuilding* the input: an autoencoder squeezes $\boldsymbol{x}$ into $\boldsymbol{z}$ and back, and the masked autoencoder of Chapter 5 fills in missing patches. This chapter asks a different question: *can we learn the probability distribution the data came from?*

That question is self-supervised too — the data is its own target — and a model that can say which images are likely must have learned a lot about what images look like.

We follow the lecture through three families (Dinh et al., 2017; Goodfellow et al., 2014; Karras et al., 2019):

1. **Kernel density estimation** — the simplest possible model: put a bump on each data point.
2. **Normalizing flows** — one invertible network that replaces the VAE's encoder–decoder pair and gives exact probabilities.
3. **GANs** — give up on probabilities entirely, and learn by playing a game between two networks.

The chapter ends with a map of all generative models (a tree, and a table) that Chapter 7 completes.

6.1 Modeling $p(X)$ Is Self-Supervision

Key Concept — Generative model

Given i.i.d. samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \sim p_{\text{data}}(\boldsymbol{x})$ from an unknown data distribution, a *generative model* is an approximation $q_{\boldsymbol{\theta}}(\boldsymbol{x})$ from which we can either (i) evaluate the density of a new point, (ii) draw new samples resembling the training data, or ideally both. Training typically maximizes the likelihood of the observed data under $q_{\boldsymbol{\theta}}$.

The lecture opens (slides 2–3) with the generative recipe in four steps: (a) we have data samples; (b) they come from the same underlying distribution; (c) we fit an approximate model $q(X)$; (d) we sample new points from $q(X)$. Slide 3 then answers a natural objection: why does this belong in a self-supervised learning course at all? Three reasons, each worth holding onto throughout this book.

Data. Only the raw data X is needed. No annotator ever touches the pipeline. The supervision signal is the input itself, in the same sense that the autoencoder of Chapter 4 supervises itself with its own input.

Learning outcome. To assign high density to real images of bedrooms and low density to random pixel noise, a model cannot avoid learning *hidden structure*: what edges are, how walls meet ceilings, the statistics of light on fabric. Density estimation on structured data forces the discovery of that structure.

Transfer. Structure learned for generation is exploitable beyond generation. The lecture makes the point with an explicit callback: "e.g. remember MAEs"—the masked autoencoder of Chapter 5 is trained purely to reconstruct pixels, yet its encoder transfers to classification, detection, and segmentation. Generative pretraining is an SSL subroutine, not an end in itself.

Simplified intuition vs. more precise view

Simple story: if you can generate data, you must have "understood" it. **More precise view:** generation quality and representation quality are *correlated but not identified*. A model can memorize samples and produce low-quality novel output (Section 6.2.4),

and a model can assign high likelihood while learning features that transfer poorly (Section 6.3.13). The bridge from "models $p(X)$" to "has learned structure" is empirical, not theorems-backed; Chapter 16 returns to it directly.

6.2 Kernel Density Estimation: The Simplest Generative Model

6.2.1 Problem: from points to distributions

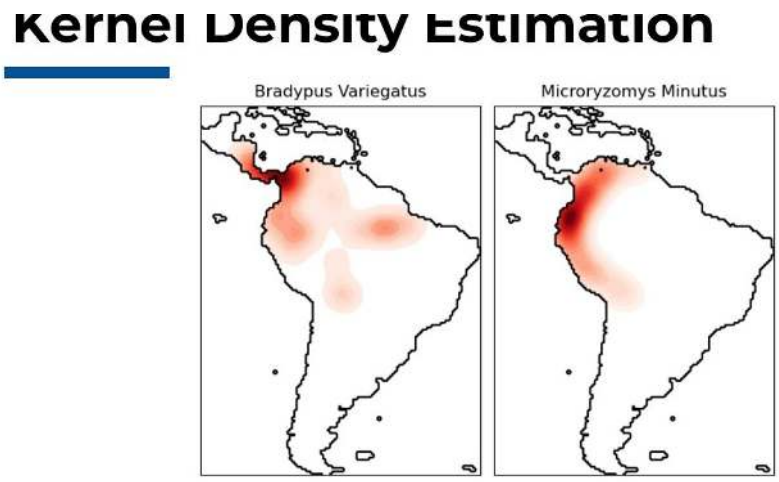


Figure 6.1. What kernel density estimation is *for*. Each map shows recorded sightings of one species in South America, turned into a continuous density by placing a kernel on every sighting and summing. Notice that the sloth's map develops several distinct peaks along different mountain ranges — KDE recovers a multi-modal distribution without ever being told how many modes to look for, which a single Gaussian could never do.

How to read it: The input was nothing but a scatter of dots. Everything smooth in these maps was produced by the bandwidth choice.

Source: Lecture 04, slides 4–5; data from Phillips et al., Ecological Modelling*, 2006.*

The lecture motivates KDE with an ecological example (slides 4–5, data from Phillips et al., *Ecological Modelling*, 2006). Two maps of South America show raw occurrence records—individual sightings—of the brown-throated sloth (*Bradypus variegatus*) and the forest small rice rat (*Microryzomys minutus*). The raw data is a scatter of points. What a biologist wants is a *density*: where does this species live, continuously over the continent, and with what concentration? No deep learning, no labels—just the question "model $p(X)$" made concrete in two dimensions.

It helps to pin down why the raw scatter is *not* already an answer. Suppose you are asked to map the risk of a rare disease across a city, and all you have is a map with fifty pins: one per recorded case. If a new resident asks about an address lying exactly between two

pins, what is their risk? An *overly localized* view replies "zero, because no pin sits on that coordinate." An *overly smoothed* view replies "the same as everywhere else in the city." Both are clearly wrong, and they are wrong in opposite directions. The first view memorizes the sample; the second ignores it. A third view—the kernel view—places a small hill of probability on each pin. Where pins cluster, the hills stack into mountains; where pins thin out, the hills subside toward zero. The width of the hills turns out to be the single most important design choice of the whole method, and it will occupy us for the rest of the section.

Why not just use the sightings? The sloth sightings on slide 4 are single dots. Treat only those exact dots as habitat and you would claim the Amazon forest *between* two sightings a hundred kilometres apart is impossible sloth territory — obviously false. What we want is a smooth answer to "how likely is a sighting *anywhere*?", built from a finite set of dots. That is density estimation.

Slide 5 shows the result: a smooth heatmap, dark where sightings are dense and pale where sparse. Notice that the sloth's map has *several separate peaks*, and nobody told the method how many to find. A mixture model needs you to fix the number of clusters K in advance (Chapter 2); KDE finds the peaks wherever the data happens to pile up.

6.2.2 From points to densities: the estimator

In plain words — KDE in three sentences

You have a scatter of points and you want a smooth "how likely is this region?" surface. Put a small identical hill on top of every single data point and add all the hills together. Where points crowd, the hills stack into a mountain; where there are no points, the surface slopes down towards zero — and the only decision you ever make is how wide to build the hills.

Key Concept — Kernel density estimate

Given n samples $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \in \mathbb{R}^d$ and a *kernel* K—a symmetric, non-negative function integrating to one—the KDE with *bandwidth* $h > 0$ is

$$\hat{f}_h(\boldsymbol{x}) \;=\; \frac{1}{n}\sum_{i=1}^{n} K_h(\boldsymbol{x} - \boldsymbol{x}_i) \;=\; \frac{1}{nh}\sum_{i=1}^{n} K\!\left(\frac{x - x_i}{h}\right), \tag{6.1}$$

in one dimension; in d dimensions the product kernel divides by h^d (or uses a Mahalanobis-scaled kernel). Each data point "lays a kernel" around itself; summing the bumps yields a smooth density.

Two small details in Equation (6.1) are easy to skip and easy to lose marks on.

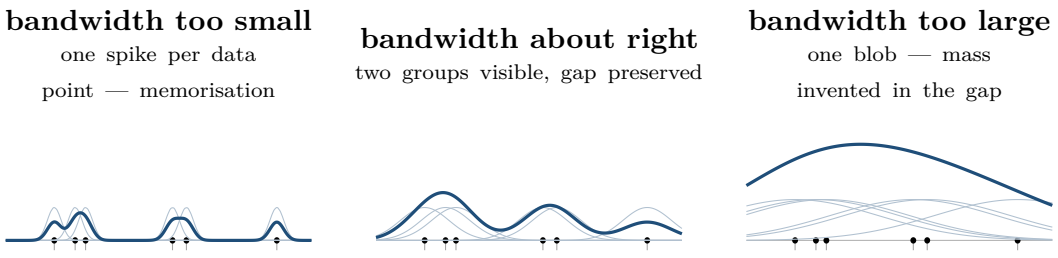


Figure 6.2. Kernel density estimation is "drop a bump on every data point and add them up". The only decision is how wide the bumps are, and that one decision is the entire bias–variance trade-off.

How to read it: Look at the gap between the left group and the lone right-hand point. In the third panel the gap has been filled in with probability that no data supports.

Why each kernel must be a density. We require $K(u) \geq 0$ and $\int K(u)\,du = 1$ — each hill is non-negative and has total area 1. Averaging n things that each have area 1 gives something with area 1. So $\hat{f}_h$ is automatically a proper probability density, whatever h you pick.

Why there is a $1/h$ in $K_h(u) = \frac{1}{h}K(u/h)$. Dividing u by h makes the hill *wider*. But a wider hill of the same height has more area — it would manufacture probability out of nothing. Multiplying by $1/h$ makes it *shorter* by exactly the amount needed to keep its area at 1. Wider and shorter, always together; that is the only way to widen a hill without cheating.

The two extremes of the bandwidth.

- $h \to 0$**:** each bump becomes an infinitely thin, infinitely tall spike sitting exactly on a training point, and zero everywhere else. The model has memorised the data and says nothing about new points.
- $h \to \infty$**:** each bump becomes so wide it is flat, and every point contributes almost equally everywhere. Clusters, gaps and peaks all disappear.

Everything useful happens between these two. Choosing where is "the bandwidth problem", and the worked example below puts numbers on it.

Common kernels, all available in scikit-learn and all plotted on slide 6: Gaussian, tophat (uniform ball), Epanechnikov (parabola), exponential, linear (triangle), cosine bump. The *shape* of the kernel matters far less than the *bandwidth*, which is why the lecture spends the next slide on the Gaussian specific case:

$$\hat{f}_h(x) \;=\; \frac{1}{n}\sum_{i=1}^{n} \frac{1}{h\sqrt{2\pi}}\,\exp\!\left(-\frac{(x-x_i)^2}{2h^2}\right). \tag{6.2}$$

This is nothing more than an n-component isotropic Gaussian mixture with one component per data point, all sharing bandwidth h.

6.2.3 Worked example: five points in the plane

In plain words — how to read a formula like $\hat{f}_h(\boldsymbol{q})$

The example below is written in a compressed notation that hides how simple it is. Take one line apart before reading on:

$$\underbrace{\hat{f}}_{\text{estimated density}}\ \underbrace{0.3}_{\text{hill width } h}\ (\ \underbrace{\boldsymbol{q}}_{\text{the spot we ask about}}\ ) = \underbrace{\tfrac{1}{5}}_{\text{average over the } n=5 \text{ points}}\ (\ \underbrace{3 \times 0.110}_{\text{three points each delivered } 0.110}\ ) \approx 0.066.$$

Said aloud: *the estimated density at* $\boldsymbol{q}$*, using hills of width* 0.3*, is one fifth of three lots of* 0.110. The hat on f means this is an *estimate* built from five points, not the true density. The subscript is the bandwidth. The thing in brackets is where we are standing.

Everyday analogy — five streetlamps

Think of the five data points as five **streetlamps**, and of $\boldsymbol{q}$ as the spot where you are standing. The question "what is the density here?" becomes "how bright is it here?"

- Every lamp throws some light on your spot. Near lamps contribute a lot, distant ones almost nothing.
- The bandwidth h is **how widely each lamp spreads its glow**. Small h is a narrow spotlight; large h is a broad floodlight.
- Add up what all five deliver, then divide by five.

That is the whole method. So when the text says "$\boldsymbol{x}_4$ now has a say", it means precisely this: at $h = 0.3$ that lamp was too far away for its light to reach you at all, and at $h = 1$ it finally does. *Bandwidth decides how far away a data point is still allowed to have an opinion.*

Where the analogy breaks. Real lamps just add light; a density has to integrate to one over the whole plane. That is what the $\frac{1}{n}$ and the $\frac{1}{2\pi h^2}$ are for — they are bookkeeping that keeps the total fixed, which is why widening the hills makes each one *shorter*.

Common Confusion

"The narrow spotlight is more intense, so why does each near point give *less* at $h = 0.3$ (0.110) than at $h = 1$ (0.124)?"

Because $\boldsymbol{q}$ is not standing at the lamp. It is standing 0.71 units away.

The $h = 0.3$ kernel is indeed far more intense at its own centre — its peak is $\frac{1}{2\pi(0.3)^2} \approx 1.77$ against $\frac{1}{2\pi} \approx 0.159$ for $h = 1$, eleven times taller. But 0.71 units is more than *two* widths away when $h = 0.3$, so you are out on the thin tail and collect only about 6% of that tall

peak: $1.77 \times e^{-2.78} \approx 0.110$. When $h = 1$, the same 0.71 units is comfortably inside one width, so you collect about 78% of a much shorter peak: $0.159 \times e^{-0.25} \approx 0.124$.

A tall narrow hill you are standing *beside* gives you less than a low wide hill you are standing *on*.

Worked Example — KDE by hand on five 2-D points

Take $n = 5$ data points in $\mathbb{R}^2$: $\boldsymbol{x}_1 = (0, 0)$, $\boldsymbol{x}_2 = (0, 1)$, $\boldsymbol{x}_3 = (1, 0)$, $\boldsymbol{x}_4 = (2, 2)$, $\boldsymbol{x}_5 = (5, 5)$. Use the product Gaussian kernel $\hat{f}_h(\boldsymbol{x}) = \frac{1}{n}\sum_i \frac{1}{2\pi h^2} \exp\big(-\|\boldsymbol{x} - \boldsymbol{x}_i\|^2/(2h^2)\big)$, and evaluate at the query point $\boldsymbol{q} = (0.5, 0.5)$.

Figure 6.3 draws every step of what follows; read it alongside the arithmetic.

The squared distances from $\boldsymbol{q}$ are: 0.5 (three times, for $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$), 4.5 (for $\boldsymbol{x}_4$), and 40.5 (for $\boldsymbol{x}_5$). Now vary the bandwidth.

Small bandwidth $h = 0.3$. Each kernel contributes $\frac{1}{2\pi(0.09)} e^{-d^2/0.18}$. The three near points each contribute about 0.110; the two far points contribute effectively 0 (the exponent for $\boldsymbol{x}_4$ is -25). Result: $\hat{f}_{0.3}(\boldsymbol{q}) \approx 0.066$. Three sharp, well-separated bumps sit on the cluster.

Natural bandwidth $h = 1$. Each near point contributes 0.124, and $\boldsymbol{x}_4$ now adds 0.017; $\hat{f}_1(\boldsymbol{q}) \approx 0.078$. The density is smooth, and the fourth point—previously invisible—now pulls a little density out of the cluster toward $\boldsymbol{x}_4$.

Large bandwidth $h = 5$. All five points contribute: 0.0063 each for the near three, 0.0058 for $\boldsymbol{x}_4$, 0.0028 for $\boldsymbol{x}_5$, giving $\hat{f}_5(\boldsymbol{q}) \approx 0.0055$. The estimate is nearly uniform over the whole square; the cluster structure has been washed out entirely.

The pattern generalizes: $h \to 0$ gives one spike per data point (pure memorization); $h \to \infty$ gives one wide bump ignoring the data entirely. Choosing h is the bias–variance trade-off made visible—small h overfits the sample, large h underfits the distribution.

Worked Example — A density in the gap: two clusters, large bandwidth

Figure 6.4 shows both bandwidths side by side. Take the 1-D training set $\{2.0, 2.1, 2.2, 8.0, 8.1, 8.2\}$: two tight clusters of three points, separated by an empty gap of width 5.8. Fit a Gaussian KDE with bandwidth $h = 10$—much wider than the gap—and evaluate at the gap's midpoint $x = 5.0$, where there is no data at all.

Every kernel has standard deviation 10, so from $x = 5.0$ every training point is well inside one standard deviation and all six contribute near their peak value of $\frac{1}{10\sqrt{2\pi}} = 0.0399$. The nearest, $x_i = 2.2$, contributes $0.0399\, e^{-(2.8)^2/200} \approx 0.0384$; the farthest, $x_i = 8.2$, contributes $0.0399\, e^{-(3.2)^2/200} \approx 0.0379$. The six contributions span barely one part in a hundred. Averaging them,

$$\hat{f}_{10}(5.0) \;\approx\; 0.0381.$$

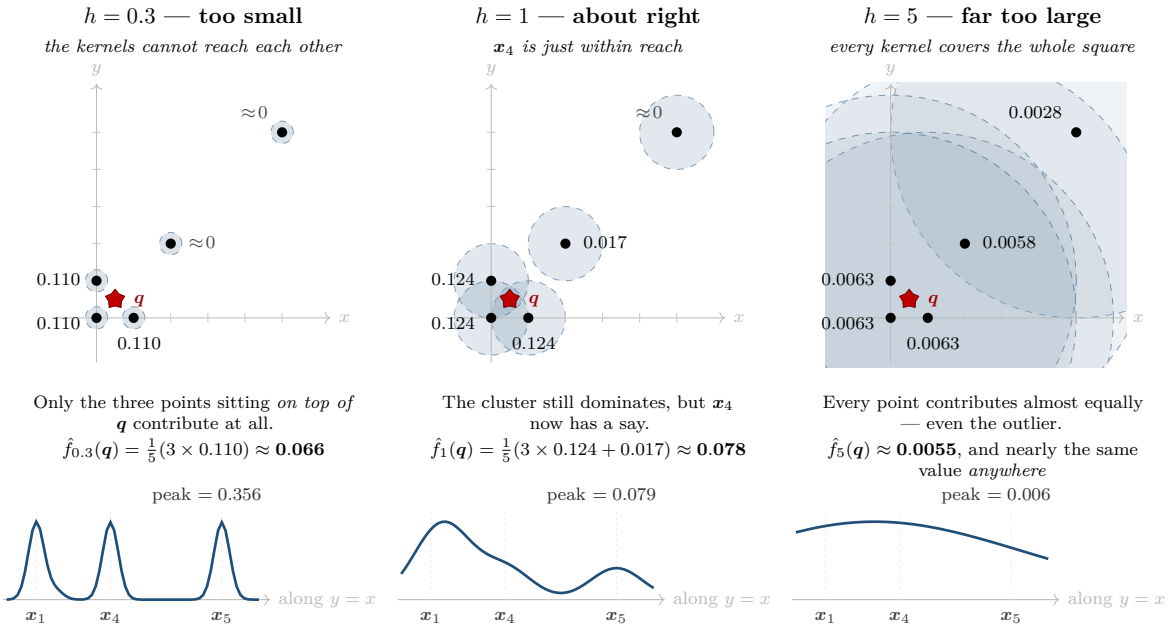


Figure 6.3. The five-point worked example of Section 6.2.3, drawn. *Top row:* black dots are the data, the red star is the query $\boldsymbol{q} = (0.5, 0.5)$, and each shaded disc is one kernel's reach (radius h). The bold number beside a data point is that point's contribution to $\hat{f}_h(\boldsymbol{q})$. *Bottom row:* the whole estimated density along the diagonal line $y = x$, which passes through the cluster, through $\boldsymbol{x}_4$ and out to the outlier $\boldsymbol{x}_5$ (each curve is scaled to its own peak, printed at the right).

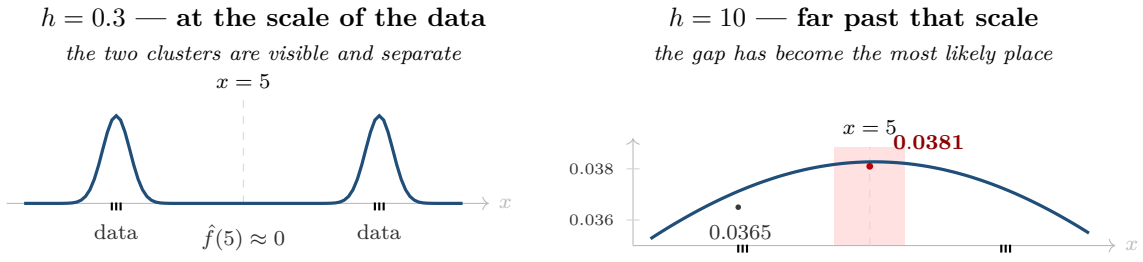
How to read it: Follow the middling point $\boldsymbol{x}_4 = (2, 2)$ across the top row. At $h = 0.3$ it contributes nothing; at $h = 1$ a little; at $h = 5$ almost as much as the points sitting on top of $\boldsymbol{q}$. Bandwidth is exactly the knob that decides how far away a data point is still allowed to have an opinion. Then look at the bottom-right curve: the valley between the cluster and the outlier has been filled in, and no data supports it.

Now compare that with the density at a point where data actually *is*. At $x = 2.0$ the three nearby points still contribute near-peak, but the three points in the far cluster are now about six units away and are discounted by $e^{-36/200} \approx 0.84$, giving

$$\hat{f}_{10}(2.0) \;\approx\; 0.0365.$$

Read those two numbers again. The estimate is *higher* in the middle of the empty gap than it is on top of a real observation. This is not blurring — it is inversion. With h this large the estimator has effectively become a single wide Gaussian centred on the mean of the data, and the mean of $\{2.0, 2.1, 2.2, 8.0, 8.1, 8.2\}$ is 5.1: a location where nothing was ever observed, and which the model now reports as the most likely place of all. The bimodality — the one structural fact that matters about this dataset — has not merely been softened; it has been replaced by its opposite. On an exam, "large bandwidth"

questions are always this story: smoothing trades variance for bias, and past the scale of the true structure the bias wins outright.



Six observations (tick marks), two clusters, an empty gap between them. The estimate says exactly that.

The peak now sits at $x \approx 5.1$ — the *mean* of the data, in the middle of the shaded empty region. The estimate is higher where there is no data than where there is.

Figure 6.4. Over-smoothing does not merely blur — past a certain bandwidth it inverts the answer. Both panels show the same six observations; only h differs. The left curve is scaled to its own peak; the right one carries a labelled axis, because its whole story is that it barely varies at all.

How to read it: Put a finger on the shaded band in the right-hand panel. No tick mark falls inside it, yet that is where the model places its maximum. Compare the two printed numbers: 0.0381 *in the void against* 0.0365 *on top of real data.*

Worked Example — Midpoint of two points, exactly

Figure 6.5 draws the two kernels and their sum. Let the training set be $D = \{0, 2\}$ and the kernel Gaussian with $h = 1$. Compute $\hat{f}(1)$, the density at the midpoint. Substituting into Equation (6.2) with $n = 2$:

$$\hat{f}(1) = \frac{1}{2\sqrt{2\pi}}\left[e^{-(1-0)^2/2} + e^{-(1-2)^2/2}\right] = \frac{1}{2\sqrt{2\pi}} \cdot 2e^{-1/2} = \frac{e^{-1/2}}{\sqrt{2\pi}} \approx \frac{0.6065}{2.5066} \approx 0.242.$$

Both points contribute symmetrically—as they must, since the kernel is symmetric—and the density at the unseen midpoint is a respectable 0.24, roughly two-thirds of the 0.40 that a *single* standard Gaussian achieves at its own center. One point's kernel does summarize its neighborhood; two points' kernels fill the gap between them. That filling-in is the entire mechanism of KDE generalization, and the previous example shows it overshooting when h grows past the scale of real structure.

6.2.4 KDE generates—badly

Because $\hat{f}_h$ is a bona fide density, we can sample from it: pick one of the n training points uniformly at random, then add Gaussian noise of bandwidth h. Slide 8 (from the scikit-learn user guide) applies this to the handwritten digits dataset (8×8 grayscale, MNIST-like): the top

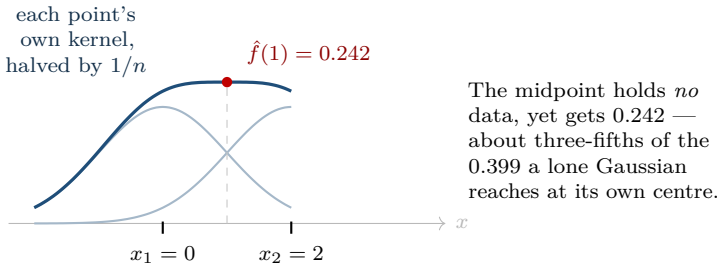


Figure 6.5. Why kernel density estimation generalises at all. Two observations, at 0 and 2; the faint curves are their individual kernels and the bold curve is the average. The unobserved midpoint inherits a substantial density purely from the overlap of two neighbours.

How to read it: This filling-in is the mechanism that makes KDE useful. Figure 6.4 is the same mechanism running well past the point of usefulness.

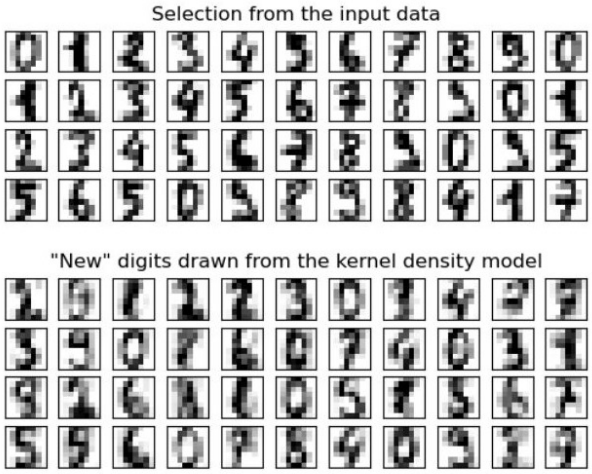


Figure 6.6. Kernel density estimation used as a generative model. *Top:* real handwritten digits from the training set. *Bottom:* "new" digits sampled from the fitted kernel density model. They are recognisably digits — and that is the problem, because each one is simply a memorised training digit with a little Gaussian noise added. Nothing about the *structure* of digit space was learned.

How to read it: Compare a bottom-row sample with the top rows and try to find the training digit it came from. You usually can. That is memorisation, not generation.

Source: Lecture 04, slide 8 (scikit-learn user guide).

grid shows real digits, the bottom grid shows new digits drawn from the kernel density model. The samples are recognizably digits but noticeably *blurrier* than the training data—the kernel smears each memorized example into a small Gaussian neighborhood, and nothing about the structure of digit space has been learned.

How should h be chosen in practice? The tempting shortcut—maximize the training-set log-likelihood $\sum_i \log \hat{f}_h(\boldsymbol{x}_i)$—collapses immediately: as $h \to 0$, the self-contribution of each training point to its own density estimate grows toward $\hat{f}_h(\boldsymbol{x}_i) \geq \frac{1}{n \cdot h\sqrt{2\pi}}$, which diverges, so "best" training likelihood is achieved at the memorization pole. The standard repair is *leave-one-out* cross-validation: score each point by the density computed from the *other* $n-1$ points,

$$\mathrm{LOO}(h) \;=\; \sum_{i=1}^{n} \log \hat{f}_{h,-i}(\boldsymbol{x}_i), \tag{6.3}$$

and pick the h that makes it largest (it is a held-out log-likelihood, a score to maximise, not a loss). This works because held-out points are scored badly at *both* extremes: if h is too small they fall in the empty gaps between spikes; if h is too large they sit on a flat smear. The best h is in between.

Notice the rule this introduces, which holds for every generative model in the book: *judge a model on data it did not train on.*

A refinement from ecology, where the species maps came from: *adaptive* KDE gives each point its own bandwidth h_i — narrow where points are crowded, to keep detail, and wide where they are sparse, to avoid false spikes in empty country.

Failure Modes

Where KDE fails. (1) Memorization. Each component sits exactly on a training point; new samples are noisy copies, not novel structure. **(2) Curse of dimensionality.** The number of points needed to estimate a d-dimensional density to fixed accuracy grows exponentially in d; on 64-D digit vectors a few thousand points are nowhere near enough—and a modest face image at $256 \times 256 \times 3$ lives in 196,608 dimensions, where training points are effectively infinitely isolated and asking KDE for a meaningful face density is asking for more images than there are atoms in the universe. **(3) The metric is wrong for images.** KDE's geometry is raw Euclidean distance in input space. Translating a digit by one pixel causes a large pixel-space distance jump while changing nothing semantically; the estimator cannot tell the difference. It has no way to learn that strokes, not pixel coordinates, are the invariant. **(4) Non-parametric cost.** "Fitting" KDE means storing the training set; evaluating one density costs $\mathcal{O}(n \cdot d)$ distances, so inference price scales linearly with dataset size—impractical at web scale, where every other model in this chapter gets *cheaper* per query as it absorbs data into fixed weights. **(5) Bandwidth sensitivity.** No single h is right everywhere when the data density varies across space (adaptive bandwidths, discussed above, are the classical patch). **(6) No learned representation.** The model does not compress, factor, or explain anything—it

is a lookup table of bumps. What we want next is a model whose internal parameters *encode* the density structure, so sampling and generalization go beyond the training set. This is the doorway to deep generative models.

6.3 Normalizing Flows: Exact Likelihoods by Inversion

6.3.1 Replacing the encoder–decoder pair with one bijection

In plain words — what a normalizing flow is, before any mathematics

A flow is one network that you are allowed to run *backwards*.

Feed it a real image and it comes out the other side as a point in a plain Gaussian cloud. Feed it a point from that Gaussian cloud and it comes out as an image. Same network, same weights, two directions. Nothing is approximated, nothing is sampled in the middle, and no second network is needed.

Because the map is exact and reversible we can do something no other model in this book can: compute, for any image you hand us, the *exact* probability our model assigns to it. That is a real number, not a bound. The catch — and it is a big one — is that "runs backwards" is a brutal constraint on what layers you are allowed to use, and the rest of this section is a tour of increasingly clever ways to satisfy it without giving up too much power.

Everyday analogy — a reversible recipe

An ordinary neural network is like cooking: flour, eggs and sugar go in, a cake comes out, and no amount of staring at the cake gives you back the eggs. Information was destroyed at every step — which is exactly what makes a ReLU or a max-pool irreversible.

A flow is a sequence of reversible moves instead: shuffling a deck, folding a sheet of paper, turning a Rubik's cube. Every move can be undone exactly, because nothing was thrown away — only rearranged. So a flow is constrained to *rearrange* its input rather than *summarise* it, and the number of output values must equal the number of input values.

Where the analogy breaks — and it matters. A shuffle preserves volume. A flow must also be able to *stretch and squeeze*, because the whole point is to turn a lumpy data distribution into a smooth Gaussian, and rearranging alone cannot do that. Section 6.3.5 shows a real model that forgot to allow squeezing, and what it had to bolt on to fix it.

Where we are coming from. The VAE of Chapter 4 needs *two* networks: an encoder that guesses a code for an image, and a decoder that turns a code back into an image. Because the encoder only guesses, the VAE can only compute a *lower bound* on how likely an image is, never the exact number.

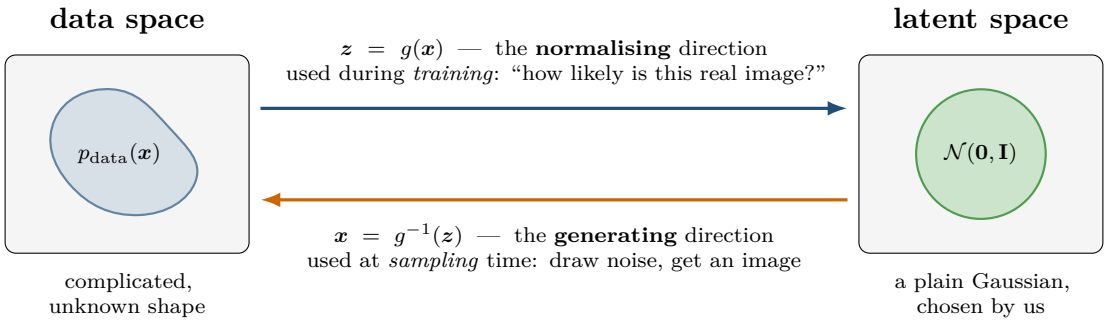


Figure 6.7. What a normalizing flow is. A single invertible network connects the messy data distribution to a tidy Gaussian. Run it one way to score data, the other way to generate data.

Slide 9 asks the natural question: *what if the encoder and the decoder were the same network, run in opposite directions?* Then nothing is a guess. Going $\boldsymbol{x} \to \boldsymbol{z}$ is exact, going $\boldsymbol{z} \to \boldsymbol{x}$ is exact, and the likelihood can be computed exactly too.

The catch is that the network must be *invertible*: every output must come from exactly one input, so nothing can be thrown away. Most familiar layers throw information away (a ReLU turns all negative numbers into 0, and you can't tell afterwards which negative number it was). The rest of this section is about building layers that are powerful *and* invertible.

Common Confusion

"My autoencoder reconstructs the training set perfectly—is it a generative model?" No. Perfect reconstruction is an *empirical optimization success*, not a structural property. Nothing in the reconstruction loss constrains *where* the latent codes land: they may clump into disjoint islands surrounded by vast uncharted regions of latent space where the decoder never received a single gradient. Sample $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, I)$ into such a space and you will usually land in a dead zone, and the decoder will cheerfully decode the void into noise. Generation requires not just a decoder but a *guarantee* that the prior's support is densely covered by codes the decoder was trained on. The VAE buys that guarantee softly, with a KL term pulling the aggregated posterior toward the prior; a normalizing flow buys it *hard*, since its decoder is a bijection: every latent point maps to some image, and the model's density accounts for all of them, so there are no untrained dead zones by construction (samples can still be poor wherever the fit itself is poor). Reconstruction quality and generation quality are different objectives; conflating them is among the most common exam errors on this material.

6.3.2 Change of variables: the mathematics of flows

Worked Example — Warm-up in one dimension

Figure 6.8 shows the whole transformation end to end. Let $Z \sim \mathcal{N}(0,1)$ and define $X = e^Z$ (so $z = g(x) = \log x$, $x > 0$). Differentiate the inverse: $dg/dx = 1/x$, so $|dg/dx| = 1/x$. The change-of-variables formula gives $p_X(x) = p_Z(\log x) \cdot |1/x|$. Check at $x = 1$: $p_X(1) = \frac{1}{\sqrt{2\pi}} \cdot 1 \approx 0.399$. At $x = e$: $p_X(e) = \frac{1}{\sqrt{2\pi}} e^{-1/2} \cdot e^{-1} \approx 0.089$. This X has the *log-normal* distribution—a skewed, heavy-right-tailed density produced from a plain Gaussian by one invertible squashing. Note the role of the $1/x$ factor: the logarithm compresses large x values into an ever-narrower latent range, so the density there must be scaled *down* to account for the stored-up mass; without the Jacobian factor the transformed density would not even integrate to one. Normalizing flows do exactly this, in high dimensions, with dozens of learned steps.

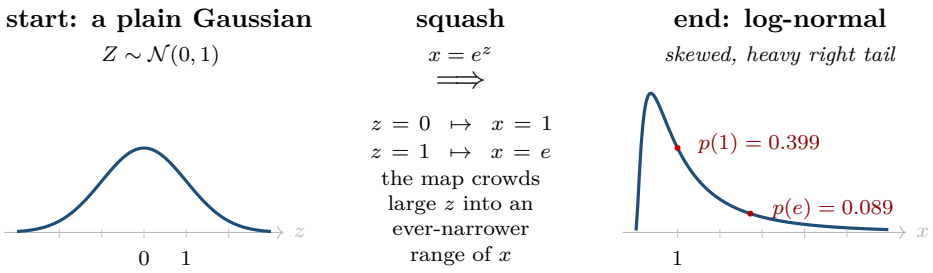


Figure 6.8. Change of variables in its smallest possible instance. One invertible squashing turns a symmetric Gaussian into a skewed, heavy-tailed density — and the Jacobian factor $1/x$ is what makes the result a genuine probability density rather than merely a reshaped curve.

How to read it: The peak height survives at $x = 1$, where $z = 0$ and the stretch factor is exactly 1. The value at $x = e$ has been discounted by $1/e$. That discount is *the Jacobian term, visible as the difference between the two marked points.*

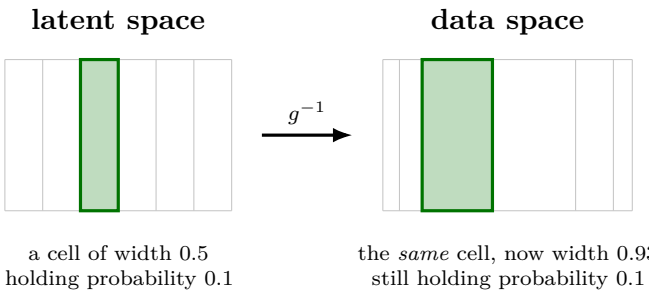
Here is the one sentence to remember: **a flow is a machine for stretching space, and the Jacobian determinant records how much stretching happened at each spot.**

Picture the data distribution as a lump of dough with a lumpy, awkward shape. Picture the Gaussian as a neat round ball made of *the same amount* of dough. The flow g reshapes the lump into the ball by smooth stretching and squashing — no tearing and no gluing, so the move can always be undone.

Now ask: *how dense was the dough at some point $\boldsymbol{x}$?* You need two facts.

1. **Where did that bit of dough end up on the ball?** Its density there is $p_Z(g(\boldsymbol{x}))$.
2. **How much was it stretched on the way?** That factor is $|\det J_g(\boldsymbol{x})|$.

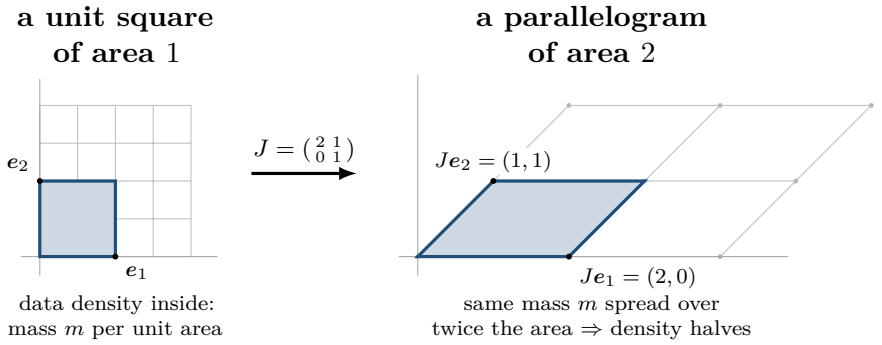
If a small patch had to be stretched a lot to cover its share of the ball, it must have been packed tightly to begin with. So *original density = density on the ball × stretch factor* — which is exactly Equation (6.4).



The map stretched this cell by a factor of 1.86. The probability inside it cannot change — probability is conserved — so the *density* (probability per unit width) must drop by the same factor of 1.86. The Jacobian determinant is exactly that stretch factor, and that is the whole content of the change-of-variables formula: *stretch ⇒ thinner density, squeeze ⇒ denser.*

Figure 6.9. Change of variables, with no calculus. A flow moves probability mass around like a fixed amount of dough: where the map stretches, the dough gets thinner; where it squeezes, the dough piles up. The Jacobian determinant is the local stretch factor and the $\log|\det J|$ term in the loss is the bookkeeping that accounts for it.

How to read it: Compare the widths of the highlighted cell on the left and the right. Its area *of probability is the same; only its width changed.*



The map sends every unit cell to a parallelogram with $|\det J| = 2$ times its area — base times height, $1 \times |\det J|$, nothing else is needed. Probability conservation then forces each point's density down by the same factor of 2. This is Equation (6.4) for a linear map, where the Jacobian is constant over space; a nonlinear flow just lets the stretch factor vary from point to point.

Figure 6.10. The Jacobian determinant in two dimensions. A square becomes a parallelogram whose area is $|\det J|$ times the original; the density is rescaled by the reciprocal. The change-of-variables formula is this picture made pointwise.

How to read it: The parallelogram's base and height came from J's columns — $J\boldsymbol{e}_1$ and $J\boldsymbol{e}_2$ are simply the columns of J, read off the arrows.

Deriving the formula. The general version (slide 11, credit: Abuzar Khan, CMU) follows from one fact: *probability is neither created nor destroyed by the map.* Take any small region $\mathcal{R}_X$ of data space, and let $\mathcal{R}_Z = g(\mathcal{R}_X)$ be where it lands. Every point of one ends up in the other, so they hold the same probability:

$$\int_{\mathcal{R}_X} p_X(\boldsymbol{x})\, d\boldsymbol{x} \;=\; \int_{\mathcal{R}_Z} p_Z(\boldsymbol{z})\, d\boldsymbol{z}.$$

Now change variables in the right-hand integral, $\boldsymbol{z} = g(\boldsymbol{x})$. A tiny volume $d\boldsymbol{x}$ in data space becomes a volume $d\boldsymbol{z}$ in latent space that is larger or smaller by the stretch factor: $d\boldsymbol{z} = |\det J_g(\boldsymbol{x})|\, d\boldsymbol{x}$. So

$$\int_{\mathcal{R}_X} p_X(\boldsymbol{x})\, d\boldsymbol{x} \;=\; \int_{\mathcal{R}_X} p_Z(g(\boldsymbol{x}))\, |\det J_g(\boldsymbol{x})|\, d\boldsymbol{x}.$$

This holds for *every* region $\mathcal{R}_X$, however small — which is only possible if the two things being integrated are equal at every point:

$$p_X(\boldsymbol{x}) \;=\; p_Z(g(\boldsymbol{x}))\, |\det J_g(\boldsymbol{x})|, \tag{6.4}$$

where $J_g(\boldsymbol{x})_{ij} = \partial g_i(\boldsymbol{x})/\partial x_j$ is the matrix of all first partial derivatives of the *normalizing direction* $g : \boldsymbol{x} \to \boldsymbol{z}$. Taking logs turns the product into the objective function of the entire field,

$$\log p_X(\boldsymbol{x}) \;=\; \log p_Z(g(\boldsymbol{x})) \;+\; \log |\det J_g(\boldsymbol{x})|. \tag{6.5}$$

Before building any layer, look at Equation (6.5) as a tug of war between its two terms.

Term 1, $\log p_Z(g(\boldsymbol{x}))$, pulls inward. The Gaussian is tallest at the origin, so this term is happiest when every training image lands near $\boldsymbol{z} = \mathbf{0}$.

Term 2, $\log|\det J_g|$, pushes outward. It rewards stretching space, and punishes squashing it.

Neither can win outright. If the flow squashed *every* image onto the origin, term 1 would be perfect — but squashing everything to one point means infinite compression, and term 2 would go to $-\infty$. If the flow stretched everything enormously, term 2 would be happy, but the images would land far out where the Gaussian is nearly zero, and term 1 would collapse. Training settles where the two forces balance, and that balance point *is* the learned flow.

A useful side effect: a flow *cannot* collapse all its inputs to the same output, the failure that haunts Chapter 10. Collapsing is exactly the infinite squashing that term 2 forbids.

Worked Example — One-dimensional affine flow

Let $z = g(x) = ax + b$ with $a \neq 0$, and $p_Z = \mathcal{N}(0, 1)$. The Jacobian is the constant $dg/dx = a$, so Equation (6.4) gives $p_X(x) = \frac{|a|}{\sqrt{2\pi}} \exp(-(ax+b)^2/2)$, and taking logs:

$$\log p_X(x) \;=\; \log|a| \;-\; \tfrac{1}{2}\log(2\pi) \;-\; \frac{(ax+b)^2}{2}.$$

Read term by term: b slides the density along the axis, $|a|$ in the exponent sharpens or flattens the bell, and $\log|a|$ out front is the volume-correction bookkeeping so the result still integrates to 1. Every trick in Real NVP and Glow is this example's three lines—scale, shift, count the stretching—made multivariate and composed.

Equation (6.5) is a *tractable* objective whenever $\det J_g$ is cheap—it is the exact log-likelihood of the data, no bound required. Training a flow is plain maximum likelihood by stochastic gradient descent.

6.3.3 Composing many flows

A single coupling or affine map is too weak to turn images into Gaussians. Slide 12 (credit: Abuzar Khan, CMU) composes T simple invertible steps. Writing $\boldsymbol{z}_0 = \boldsymbol{z} \sim p_Z$ for the base, each step applies f_t in the *sampling* direction $\boldsymbol{z}_{t-1} \to \boldsymbol{z}_t$, or its inverse $g_t = f_t^{-1}$ in the *likelihood* direction $\boldsymbol{x} \to \boldsymbol{z}$:

$$\boldsymbol{x} = f_T \circ f_{T-1} \circ \cdots \circ f_1(\boldsymbol{z}), \qquad \boldsymbol{z} = g_1 \circ g_2 \circ \cdots \circ g_T(\boldsymbol{x}).$$

Probability mass picks up one log-determinant per step, and determinants of compositions multiply, so the log-determinants add (with $\boldsymbol{z}_T = \boldsymbol{x}$, $\boldsymbol{z}_0 = \boldsymbol{z}$):

$$\log p_X(\boldsymbol{x}) \;=\; \log p_Z(\boldsymbol{z}_0) \;+\; \sum_{t=1}^{T} \log \left| \det J_{g_t}(\boldsymbol{z}_t) \right|. \tag{6.6}$$

Where p_Z is, say, $\mathcal{N}(\mathbf{0}, I)$, every term on the right is explicit.

Mind the order of composition—it is the single most common slip in this material. Sampling starts at $\boldsymbol{z}_0 = \boldsymbol{z}$ and applies f_1 *first*, then $f_2, \ldots, f_T$; likelihood evaluation starts at $\boldsymbol{x} = \boldsymbol{z}_T$ and applies $g_T = f_T^{-1}$ *first*, unwinding to g_1. Under the hood this works because the Jacobian of a composition is the product of the Jacobians (chain rule), and determinants of products are products of determinants, so log-determinants simply accumulate additively along the chain—one scalar per layer, whatever the architecture.

6.3.4 The design constraint: triangular Jacobians

What should a flow layer look like? Slide 13 (credit: Stefano Ermon) lists the desiderata. The prior p_Z must allow efficient sampling and tractable density (isotropic Gaussian is the standard

choice). The transformations must be invertible with tractable evaluation in *both* directions: likelihood training needs efficient $\boldsymbol{x} \to \boldsymbol{z}$; sampling needs efficient $\boldsymbol{z} \to \boldsymbol{x}$. And every layer must offer a cheap Jacobian determinant—recall from Chapter 2's linear algebra refresher that det costs $\mathcal{O}(d^3)$ by generic LU factorization, hopelessly slow for the $256^2 \times 3$-dimensional space of an image.

In plain words — why "triangular" is the magic word

A determinant of a general $d \times d$ matrix costs about d^3 operations. For a 256×256 colour image $d \approx 200{,}000$, so d^3 is about 8×10^{15} — per image, per training step. Completely impossible.

But one shape of matrix has a free determinant: a triangular one, where everything above (or below) the diagonal is zero. For those the determinant is just the product of the diagonal entries — d multiplications instead of d^3 operations. So the question "what layer should a flow use?" becomes the far more concrete question: *which functions have a triangular Jacobian?*

And there is a clean answer. The Jacobian entry $\partial y_i / \partial x_j$ is zero whenever output i does not look at input j. So the Jacobian is triangular exactly when **output i depends only on inputs $1, \dots, i$** — when the layer is *autoregressive*. Every flow in the rest of this chapter is a different way of arranging that dependency pattern.

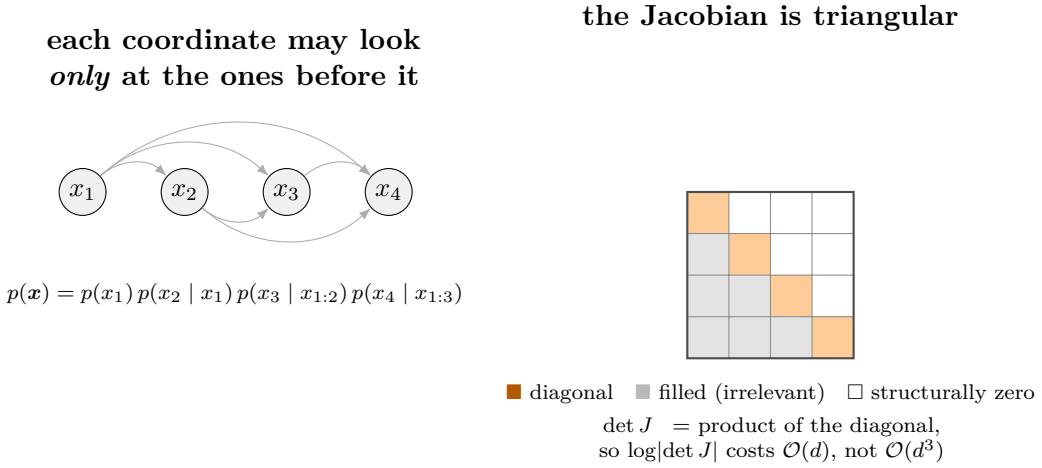


Figure 6.11. Autoregressive conditioning and the triangular Jacobian it produces. Every flow in this chapter is a different way of arranging for this picture to be true, because a triangular Jacobian is the only kind whose determinant is cheap at image scale.

How to read it: White cells above the diagonal are zero because output y_1 cannot depend on input x_4 — that is what "autoregressive" means. A coupling layer is the special case where the top-left block is the identity.

Key takeaway — the sentence that generates the whole flow literature

> A flow layer must be (i) invertible and (ii) have a triangular Jacobian. Triangular means "output i sees only inputs $1{:}i$". Every named flow model — NICE, Real NVP, Glow, MAF, IAF — is a different answer to the question *how do I get the most expressive power out of that restriction?*

The key idea: choose transformations whose Jacobian has *special structure*. If J is triangular, the determinant is the product of the diagonal entries—$\mathcal{O}(d)$. The entire flows literature can be read as a hunt for expressive, learnable layers with this property.

What this rules out. Most familiar layers fail one of the two requirements:

- **Strided convolution** shrinks the image, so information is lost — not invertible.
- **Max pooling** keeps the largest value and throws the rest away — not invertible.
- **ReLU** turns every negative number into 0; afterwards you cannot tell whether the input was -1 or -100 — not invertible.
- **A plain fully connected layer** can be invertible, but its determinant costs $\mathcal{O}(d^3)$ — far too slow.

So "just use a ResNet" is not an option. Flow layers have to be *designed* so that invertibility and a cheap determinant are guaranteed by their structure, not hoped for. The next subsections give the two classic designs.

6.3.5 NICE: the first coupling flow

What is it?

NICE — *Non-linear Independent Components Estimation* (Dinh et al., 2015) — is the model that invented the trick every later flow in this chapter is built on. It is a deep, invertible network whose Jacobian determinant is not merely cheap to compute but is *exactly one*, so the change-of-variables correction term disappears from the loss altogether.

Why was it introduced?

In 2014 there were two kinds of generative model with an exact likelihood, and each had a fatal flaw. *Autoregressive* models — predict pixel 1, then pixel 2 given pixel 1, and so on — gave exact likelihoods but needed one network pass per pixel to generate a single image, so tens of thousands of passes. *Latent-variable* models such as the VAE generated in one pass but could only report a lower bound on the likelihood, never the likelihood itself. NICE asked whether a deep network could be made invertible *by construction*, so that both directions cost one pass and the likelihood comes out exact.

The obstacle was the determinant. Equation (6.5) needs $\log|\det J_g(\boldsymbol{x})|$, and a general determinant costs $\mathcal{O}(d^3)$. NICE's answer is the most economical one imaginable: build a layer whose determinant is 1, so the term vanishes and nothing has to be computed at all.

The intuition

In plain words — the additive coupling trick

Cut the vector into two halves. Leave the first half completely alone. Add to the second half *any* function of the first half — a deep network, as complicated as you like.

Why is this reversible? Because the first half came through untouched. When you want to go backwards you read the first half straight off the output, feed it into the same network to recompute exactly the same shift, and subtract it. The network itself is never inverted — it is only ever *evaluated*, in the same direction, both times.

That single observation is the foundation of Real NVP, Glow, and half the modern flow literature.

Model at a glance: NICE Dinh, Krueger & Bengio, 2015 (arXiv 2014)

In one line	Split the vector in half, leave one half unchanged, and shift the other half by a neural network applied to the first half.
Problem it solves	Making a deep network invertible with a *free* Jacobian determinant, so exact maximum-likelihood training becomes possible at scale.
The layer	$\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}, \quad \boldsymbol{y}_{d+1:D} = \boldsymbol{x}_{d+1:D} + m(\boldsymbol{x}_{1:d})$, where m is any neural network.
Jacobian	Lower triangular with an all-ones diagonal, so $\det J = 1$ and $\log\lvert\det J\rvert = 0$. The correction term is not approximated — it is absent.
Inverse	$\boldsymbol{x}_{1:d} = \boldsymbol{y}_{1:d}, \quad \boldsymbol{x}_{d+1:D} = \boldsymbol{y}_{d+1:D} - m(\boldsymbol{y}_{1:d})$. One subtraction.
Full architecture	Four additive coupling layers with the split alternated between them, followed by one diagonal *rescaling* layer (discussed below).
Prior	Factorised logistic or Gaussian. "Independent components" in the name refers to this: the map is trained to turn dependent pixels into independent latent coordinates.
Training	Plain maximum likelihood by SGD, with $\log p_X(\boldsymbol{x}) = \log p_Z(g(\boldsymbol{x}))$ — no determinant term at all except from the final rescaling layer.
Sampling	Draw $\boldsymbol{z}$ from the prior and run the layers backwards. One pass.
Key limitation	*Volume preserving.* With $\det J = 1$ everywhere the map can bend space but never compress or expand it, so it cannot concentrate probability mass. The bolted-on rescaling layer is a partial fix; Real NVP's affine coupling is the real one.
Leads directly to	Real NVP (Section 6.3.6).

How data moves through it, step by step

Step by step — one NICE layer, forwards and backwards

Step 1. Split. Take the input $\boldsymbol{x} \in \mathbb{R}^D$ and cut it into $\boldsymbol{x}_{1:d}$ (the "identity half") and $\boldsymbol{x}_{d+1:D}$ (the "transformed half"). For images the split is typically by checkerboard position or by channel.

Step 2. Condition. Run *only* the identity half through a neural network m. Nothing about m needs to be special: a deep ReLU MLP or a ResNet is fine. It outputs a vector the same size as the transformed half.

Step 3. Shift. Add: $\boldsymbol{y}_{d+1:D} = \boldsymbol{x}_{d+1:D} + m(\boldsymbol{x}_{1:d})$, and copy the identity half straight through: $\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}$.

Step 4. Alternate and repeat. On the next layer, swap which half is the identity half — otherwise half the coordinates would never be touched by anything.

Step 5. To invert, read the identity half off the output, recompute m on it (forwards, as always), and subtract.

Worked Example — NICE by hand in $\mathbb{R}^4$

Let $\boldsymbol{x} = (1.0,\ 2.0,\ 3.0,\ 4.0)$ and split at $d = 2$, so the identity half is $(1.0, 2.0)$ and the transformed half is $(3.0, 4.0)$. Suppose the network m, evaluated at $(1.0, 2.0)$, returns $(0.5,\ -1.5)$. Then

$$\boldsymbol{y} = (\underbrace{1.0,\ 2.0}_{\text{copied}},\ \underbrace{3.0 + 0.5,\ 4.0 - 1.5}_{\text{shifted}}) = (1.0,\ 2.0,\ 3.5,\ 2.5).$$

The determinant. Write out the Jacobian:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^\top} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ * & * & 1 & 0 \\ * & * & 0 & 1 \end{bmatrix},$$

where the $*$ entries are $\partial m / \partial \boldsymbol{x}_{1:2}$ — whatever they happen to be. The matrix is lower triangular with every diagonal entry equal to 1, so $\det J = 1 \cdot 1 \cdot 1 \cdot 1 = 1$ *regardless of what m does.* The $*$ entries never enter the determinant.

Inverting. Given $\boldsymbol{y} = (1.0, 2.0, 3.5, 2.5)$, read off $\boldsymbol{x}_{1:2} = (1.0, 2.0)$, evaluate $m(1.0, 2.0) = (0.5, -1.5)$ again — the same call as on the way forward — and subtract: $\boldsymbol{x}_{3:4} = (3.5 - 0.5,\ 2.5 + 1.5) = (3.0,\ 4.0)$. Recovered exactly, at the cost of one forward pass through m.

The catch, and NICE's patch for it

Common Confusion

"$\det J = 1$ sounds like the best possible outcome. Why did anyone need Real NVP?" Because $\det J = 1$ is not only cheap, it is also a *constraint*. Look again at $\log p_X(\boldsymbol{x}) = \log p_Z(g(\boldsymbol{x})) + \log|\det J_g(\boldsymbol{x})|$. If the second term is identically zero, the model has no way to say "this region of data space is dense and that one is sparse" by squeezing space. All it can do is *move* points around so that the dense regions happen to land where the Gaussian prior is already tall. For simple data that is enough; for images it is a serious handicap.

NICE's own fix is a single *diagonal rescaling layer* at the very top, $z_i = s_i \cdot h_i$ with learned s_i, contributing $\log|\det| = \sum_i \log|s_i|$. That gives the model one global opportunity to allocate volume differently per latent coordinate — and in the paper the learned s_i come out strongly non-uniform, which is precisely the evidence that the constraint was binding. Real NVP's insight was to give *every* layer that freedom, per coordinate, conditioned on the data.

Where it sits historically

What changed — from autoregressive models to NICE

Before (NADE, RNADE and the autoregressive family). Exact likelihood and good density estimates, but generating one sample costs d sequential network passes, because pixel i cannot be produced until pixel $i-1$ exists.

NICE's change. Give up the fully general autoregressive dependency and use the coarsest possible version of it — a single split into two blocks. The Jacobian is still triangular, so the determinant is still cheap, but now *all* the coordinates in a block are computed at once, so both directions cost one pass.

What it cost. Expressive power per layer. One coupling layer transforms half the coordinates using information from the other half; an autoregressive layer transforms *every* coordinate using all of its predecessors. NICE compensates with depth, and that trade — depth instead of sequential passes — is the defining feature of the coupling-flow lineage.

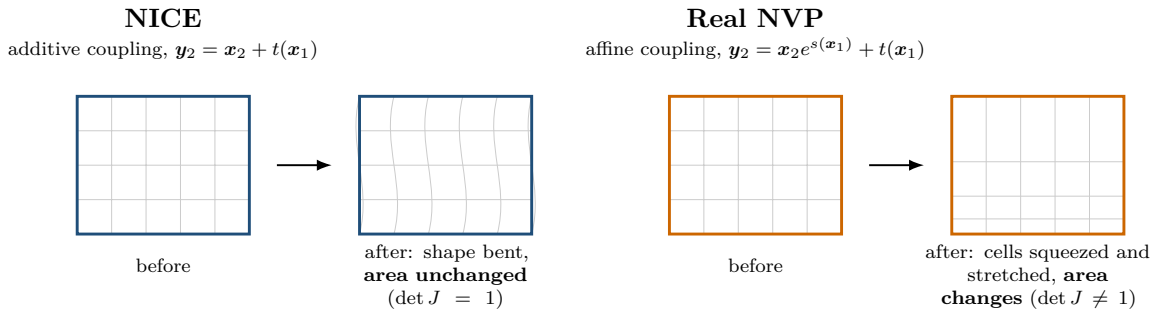


Figure 6.12. The one difference between NICE and Real NVP, and why the name says "non-volume preserving". Additive coupling can only *bend* space; affine coupling can also *compress and expand* it, which is what lets the model pile probability mass up where the data is dense.

How to read it: Count grid cells. On the left the cells change shape but each keeps its area. On the right the cells at the bottom have been squashed flat — that is mass being concentrated.

6.3.6 Real NVP: affine coupling layers

What changed — NICE → Real NVP: one multiplication

NICE's coupling layer *adds* something to the second half. Real NVP's coupling layer also *multiplies* it by something. That is the entire difference, and it is worth exactly the name of the paper: "non-volume preserving" means $\det J$ is no longer stuck at 1.

$$\text{NICE:}\quad \boldsymbol{y}_2 = \boldsymbol{x}_2 + t(\boldsymbol{x}_1) \qquad \Longrightarrow \qquad \text{Real NVP:}\quad \boldsymbol{y}_2 = \boldsymbol{x}_2 \odot e^{s(\boldsymbol{x}_1)} + t(\boldsymbol{x}_1).$$

The per-coordinate scale $e^{s(\boldsymbol{x}_1)}$ is what lets the model pile probability mass up where data is dense and thin it out where data is sparse — at every layer, and *conditioned on the data*, rather than once at the top with a global constant as in NICE.

Model at a glance: Real NVP — Dinh, Sohl-Dickstein & Bengio, 2017 (arXiv 2016)

In one line	NICE's coupling layer plus a learned per-coordinate scale, stacked with alternating masks into a multi-scale architecture.
Problem it solves	NICE could not change local volume, so it could not shape a density properly. Real NVP can.
The layer	$\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}, \quad \boldsymbol{y}_{d+1:D} = \boldsymbol{x}_{d+1:D} \odot \exp\big(s(\boldsymbol{x}_{1:d})\big) + t(\boldsymbol{x}_{1:d})$.
Jacobian	Lower triangular with diagonal $\big(\mathbf{1}, e^{s(\boldsymbol{x}_{1:d})}\big)$, so $\log\lvert\det J\rvert = \sum_j s_j(\boldsymbol{x}_{1:d})$ — read straight off the scale network's output. No matrix is ever formed.
Inverse	$\boldsymbol{x}_{d+1:D} = \big(\boldsymbol{y}_{d+1:D} - t(\boldsymbol{y}_{1:d})\big) \odot \exp\big(-s(\boldsymbol{y}_{1:d})\big)$. Elementwise, exact, one pass.
Masks	*Checkerboard* masks (alternate pixels) capture fine spatial structure; *channel* masks (alternate channels, after a "squeeze" that trades resolution for channels) capture structure across feature maps. The two are interleaved so that every coordinate eventually gets transformed.

Multi-scale	Half the variables are factored out as latents after each block (Figure 6.14), so only a shrinking remainder pays for the deep layers.
Training	Exact maximum likelihood — literally Equation (6.6) — with Adam. No bound, no adversary, no sampling inside the loss.
Sampling	One backwards pass, the same cost as a forward pass.
Strengths	Exact likelihood, exact inference, one-pass sampling, a navigable latent space, and training that is stable compared with a GAN.
Weaknesses	Samples visibly blurrier than contemporary GANs; the latent must have the same dimension as the data; the hand-designed masks waste capacity — which is what Glow fixes.
Leads to	Glow (Section 6.3.8), conditional flows (Section 6.3.12), and SRFlow (Lugmayr et al., 2020).

The flow half of the lecture is built around one paper, *Density Estimation using Real NVP* by Dinh, Sohl-Dickstein and Bengio (Dinh et al., 2017) (slide 14). The name is worth decoding, because it says exactly what changed from NICE (Dinh et al., 2015): *real-valued non-volume preserving*. NICE could only *shift* half the variables, so it could never squeeze or stretch space ($|\det J| = 1$ everywhere). Real NVP also *scales* them by $\exp(s(\cdot))$, so it can squeeze space where data is dense and stretch it where data is sparse — which is precisely what turning a lumpy image distribution into a round Gaussian requires.

Before the equations, the intuition. Picture two people packing two suitcases under a strict division of labor. One person packs valise $\boldsymbol{x}_{1:d}$ however they like and closes it. The second person *looks inside the closed valise* and, based on its contents, chooses how to fold and compress the items going into valise $\boldsymbol{x}_{d+1:D}$. Unpacking is trivial: open the first valise (untouched), look at it, deduce exactly which folding rule was applied to the second, and undo it item by item. The point of the story is the information flow: the rule for transforming the second half is allowed to be arbitrarily complicated—a chaotic function of the first half—because at unpacking time the first half is sitting there in plain view, unchanged. That is why the scale and translation networks in a coupling layer may be *any* neural networks whatsoever, including big non-invertible ResNets: they are only ever evaluated forward, never inverted.

Key Concept — Affine coupling layer

Split the input $\boldsymbol{x} \in \mathbb{R}^D$ into $\boldsymbol{x}_{1:d}$ and $\boldsymbol{x}_{d+1:D}$ at some cut d. The output is

$$\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}, \qquad \boldsymbol{y}_{d+1:D} = \boldsymbol{x}_{d+1:D} \odot \exp\big(s(\boldsymbol{x}_{1:d})\big) + t(\boldsymbol{x}_{1:d}), \tag{6.7}$$

where s and t are *arbitrary* neural networks (they need not be invertible—they only ever run forward) and $\odot$ is element-wise product. The first half passes through unchanged; the second half is scaled and shifted by functions of the first half. The exp in the scale guarantees a strictly positive multiplier, so division by it during inversion is always defined.

Two of the three properties we demanded in Section 6.3.4 now fall out for free.

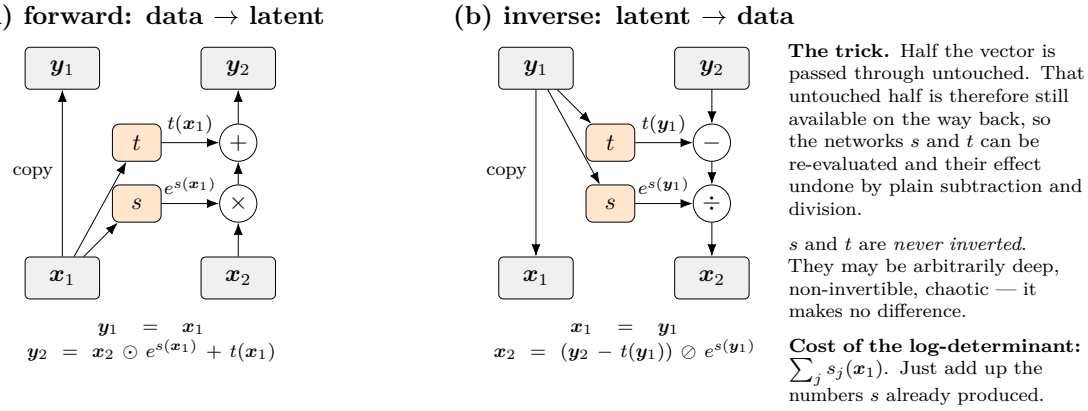


Figure 6.13. The affine coupling layer of Real NVP, forward and backward. Redrawn after Lecture 04, slide 15. Reading the two panels side by side is the fastest way to see why the layer is invertible.

How to read it: Panel (b) is panel (a) read upwards with $\times$ replaced by $\div$ and $+$ by $-$. The orange boxes are identical in both — same networks, same inputs, run forwards both times.

The Jacobian of this map is lower-triangular by construction:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^\top} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \partial \boldsymbol{y}_{d+1:D} / \partial \boldsymbol{x}_{1:d}^\top & \operatorname{diag}\big(\exp[s(\boldsymbol{x}_{1:d})]\big) \end{bmatrix}, \tag{6.8}$$

so the log-determinant is $\sum_j s_j(\boldsymbol{x}_{1:d})$—no factorization needed. And the inverse is analytic: recover $\boldsymbol{x}_{1:d} = \boldsymbol{y}_{1:d}$, then $\boldsymbol{x}_{d+1:D} = \big(\boldsymbol{y}_{d+1:D} - t(\boldsymbol{x}_{1:d})\big) \odot \exp\big(-s(\boldsymbol{x}_{1:d})\big)$. Slide 15 draws forward and inverse propagation side by side: identity on the first half, and $\times, +$ (respectively $-, \div$) on the second.

Worked Example — One coupling layer in $\mathbb{R}^3$

Let $\boldsymbol{x} = (x_1, x_2, x_3)$, split at $d = 1$, and suppose $s(x_1) = (0.2, -0.5)$, $t(x_1) = (1, 0)$ say, after evaluating the networks at the observed x_1. Then

$$y_1 = x_1, \qquad y_2 = e^{0.2}\, x_2 + 1 \approx 1.221\, x_2 + 1, \qquad y_3 = e^{-0.5}\, x_3 \approx 0.607\, x_3.$$

The Jacobian is triangular with diagonal $(1, 1.221, 0.607)$, so $\det J = 1 \cdot 1.221 \cdot 0.607 \approx 0.741$. To invert, read x_1 off from y_1, then $x_2 = (y_2 - 1)/1.221$, $x_3 = y_3/0.607$. Three scalar divisions—no matrix inversion, no iteration.

Worked Example — Forward and inverse with data-dependent s, t

To see that nothing changes when s and t are functions, take a 2-D layer with $s(x_1) = 0.5\, x_1$ and $t(x_1) = 2\, x_1$, and push $\boldsymbol{x} = (2, 3)$ through it. Forward: $y_1 = 2$, and $y_2 = 3 \cdot e^{1.0} + 4 \approx 12.154$. Inverse: read $x_1 = y_1 = 2$; re-evaluate the networks at it, $s(2) = 1.0$, $t(2) = 4.0$;

then $x_2 = (12.154 - 4) \cdot e^{-1.0} \approx 8.154 \times 0.368 \approx 3$. Perfect reconstruction, with s and t evaluated only in the forward direction in both passes. The Jacobian of this same layer is

$$J = \begin{bmatrix} 1 & 0 \\ 0.5\, x_2 e^{0.5x_1} + 2 & e^{0.5x_1} \end{bmatrix},$$

and at $\boldsymbol{x} = (2, 3)$ it evaluates to $\begin{smallmatrix} 1 & 0 \\ 6.08 & 2.718 \end{smallmatrix}$: diagonal $(1, e)$, determinant $e \approx 2.718$, log-determinant $1.0 = s(2)$. The lower-left entry is a messy derivative of the networks s and t—and the determinant never sees it; only the diagonal counts.

Why go to all this trouble to get an *easy* inverse? You might ask: why not just use an ordinary, powerful layer, and when we need to run it backwards, *solve* for the input numerically — guess, check, adjust, repeat until it fits?

Because a flow runs backwards all the time, not once in a while.

- **Training** runs every layer in the direction $\boldsymbol{x} \to \boldsymbol{z}$, for every image, at every step.
- **Sampling** runs every layer in the direction $\boldsymbol{z} \to \boldsymbol{x}$, for every image you generate.

If "backwards" meant a trial-and-error search, each layer might need, say, 20 guesses to converge. With 50 layers that is 1,000 network passes for *one* image, instead of 50. Worse, backpropagation would then have to differentiate through the whole guessing loop.

A coupling layer avoids all of that. Going backwards costs exactly *one* extra run of s and t — and those two networks are only ever run *forwards*, to read off the scale and shift. Nothing is ever solved.

Key takeaway

> An easy inverse is not a nicety. A flow uses its inverse on every training step and every generated sample, so the inverse must cost about the same as a forward pass. Coupling layers are the cheapest way anyone has found to guarantee that.

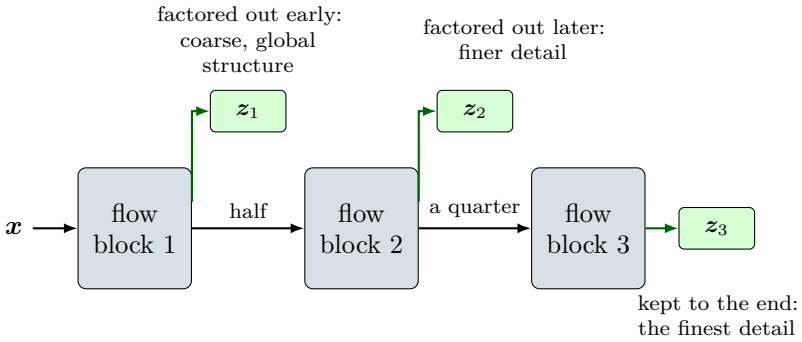
Why the split has to alternate. One coupling layer copies half of its input straight through, unchanged. Now imagine stacking ten coupling layers that all split the same way — say, always "left half copied, right half transformed". The left half is copied by layer 1, copied again by layer 2, and so on: it reaches the output *exactly as it went in.* Ten layers of expensive network, and half the pixels were never touched.

The fix is simple: swap the roles on every layer.

	left half	**right half**
layer 1	copied	transformed
layer 2	transformed	copied
layer 3	copied	transformed

After two layers, every coordinate has been transformed at least once, and information from each half has been used to reshape the other. For images Real NVP does the swapping in two patterns — a *checkerboard* (alternate pixels) and a *channel split* (alternate colour/feature channels) — but the idea is only ever this: *don't let any coordinate ride through the whole stack untouched.*

6.3.7 Multi-scale architectures: factoring out latents



Why bother. A flow cannot change dimension, so a $256 \times 256 \times 3$ image needs 196,608 latent numbers no matter what. Running *all* of them through *every* layer is ruinously expensive. Multi-scale lets most variables leave early, so only a shrinking remainder pays for the deep layers — and it produces a useful side effect: a coarse-to-fine latent code.

Figure 6.14. The multi-scale architecture of Real NVP, inherited by Glow. Variables are released as latents in batches rather than all at the end.

How to read it: Follow the green arrows: each one is a group of numbers that has finished its journey and is handed straight to the Gaussian prior.

The problem. A flow cannot throw information away, so the latent has exactly as many numbers as the image. A 256×256 colour image has 196,608 numbers, and pushing all of them through dozens of layers is very expensive.

Real NVP's answer: let variables leave early. After the first block of layers, half of the variables are declared finished. They are sent straight to the Gaussian prior as latents and take no further part. The other half continue into the next block. After that block, half of *those* leave, and so on (Figure 6.14).

Two benefits.

- **Cost.** Only a shrinking remainder of the variables pays for the deep layers.
- **A coarse-to-fine code.** It turns out that the variables that leave early end up describing coarse, global structure, and those kept to the end describe fine detail. You get an organised latent space as a side effect.

Between blocks the image is *squeezed*: each 2×2 block of pixels is folded into four channels, halving the width and height. That keeps the coupling layers working on progressively coarser grids.

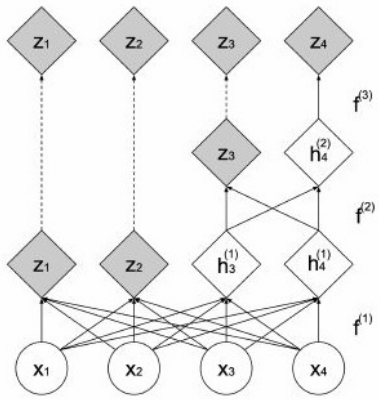


Figure 6.15. The lecture's four-variable example of the multi-scale factorization, in the slide's own dependency diagram (the block-level view of Figure 6.14 shows the same idea at a coarser grain). Circles are the data coordinates $x_1, \ldots, x_4$; grey diamonds are latents handed straight to the prior; white diamonds $h_i^{(k)}$ are variables still in flight. Reading bottom to top: the first flow $f^{(1)}$ mixes all four inputs; $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ are factored out immediately, while $h_3^{(1)}$ and $h_4^{(1)}$ pass through $f^{(2)}$; $f^{(2)}$ releases $\boldsymbol{z}_3$, and $f^{(3)}$ finishes with $\boldsymbol{z}_4$. Variables that leave early cost the fewest layers and end up carrying the coarsest, most global structure; $\boldsymbol{z}_4$, dragged through every layer, carries the finest.

How to read it: Trace x_4's fate versus x_1's: both enter $f^{(1)}$, but only the x_4 line of descent continues through $f^{(2)}$ and $f^{(3)}$. That asymmetry of depth of processing *is the entire design.*

Source: Lecture 04, slide 16 (Dinh et al., 2017).

Slide 17 shows Real NVP being honest about itself: real-vs. model on CelebA faces and LSUN bedrooms, plus latent interpolations. The samples are visibly imperfect—blurry, artifacted. This is the expected state of flows before Glow, and the lecture will return to it: exact likelihoods and good-looking samples are different goals. (Look at them with Real NVP's masking history in mind: the fixed checkerboard and channel splits that alternate the identity half are hand-designed spatial priors, not learned mixings—a main suspect for the artifacts, and exactly the component Glow is about to replace. There is also a fairness point in Real NVP's favor worth registering now, because it will invert when we reach GANs: trained by maximum likelihood, a flow is penalized for every training point it assigns low density, so it is *forced to cover every mode* of the data. Diversity is exactly what likelihood buys; sharpness is what it fails to buy.)

Two implementation notes complete the multi-scale picture. The *squeezing* operation that accompanies each scale is a pure reshape: a $c \times h \times w$ tensor becomes a $4c \times h/2 \times w/2$ tensor by rearranging 2×2 spatial blocks into channels. It is trivially invertible (reshape back) and its Jacobian is a permutation matrix with determinant ± 1, so it costs nothing

Figure 6.16. Real NVP being honest about itself. *Top row:* real CelebA faces and LSUN bedrooms (left) beside model samples (right). *Bottom row:* latent-space interpolations. The samples are plausible at a glance and fall apart on inspection — soft edges, smeared textures, faces that are not quite faces. This is the state of flow samples immediately before Glow.

How to read it: Look at the model samples with Figure 6.18 in mind: the two models differ by little more than a learned channel mixing, and the visual gap is enormous. Note also that every *sample here is a different person in a different pose — likelihood training forces coverage of all the modes, which is exactly what the GANs of Section 6.4 are not obliged to do.*

Source: Lecture 04, slide 17 (Dinh et al., 2017).

in Equation (6.6)—yet it is what lets coupling layers at deep scales operate on semantically coarse variables. And one caution about the factorization step itself:

Common Confusion

Factoring out latents does not discard information. When half the variables are output as $\boldsymbol{z}$ at an intermediate scale, they are not thrown away—they are *kept verbatim* as coordinates of the final latent vector $\boldsymbol{z}$, and they receive their likelihood under p_Z at that scale. The total latent code is the concatenation of everything factored out at every scale plus the deepest residue, and it has exactly the same dimension as $\boldsymbol{x}$, as bijectivity demands. What the architecture removes is *computation* (factored coordinates see no further layers), not information: at sampling time those coordinates are drawn from the prior and pass only through the inverses of the layers that actually acted on them. Discarding them would break injectivity and turn the flow into a lossy autoencoder with extra steps.

6.3.8 Glow: invertible 1×1 convolutions

What changed — Real NVP → Glow: learn the shuffling

Between coupling layers, Real NVP has to *rearrange* which coordinates sit in which half — otherwise the same half would be copied through unchanged forever. It does this with a fixed, hand-designed pattern (checkerboard, then channel-swap, alternating). Glow replaces that fixed pattern with a *learned, invertible* mixing of the channels, implemented as a 1×1 convolution, and swaps batch normalisation for *actnorm*. Two small parts changed; everything else is Real NVP.

Glow's complaint about Real NVP. Glow, by Kingma and Dhariwal (Kingma and Dhariwal, 2018) (slide 18), points at one weakness in Real NVP. Between coupling layers the halves must be *reshuffled*, or some coordinates would pass through the whole network untouched. Real NVP reshuffles with fixed rules picked by the authors — reverse the channel order, or alternate checkerboard patterns — and never changes them. But the best way to reshuffle surely depends on the data. Glow's idea: *learn* the reshuffling.

How: the invertible 1×1 convolution (slide 19). On an image of size $h \times w$ with c channels, a 1×1 convolution does one simple thing: at every pixel, it multiplies that pixel's channel vector $\boldsymbol{v} \in \mathbb{R}^c$ by the same $c \times c$ matrix W, giving $W\boldsymbol{v}$. A fixed reshuffle is the special case where W is a permutation matrix; a learned W can reshuffle *and* mix and rescale. Its log-determinant is easy:

$$\log \left| \det \frac{d\,\texttt{conv2D}(\boldsymbol{h}; W)}{d\boldsymbol{h}} \right| \;=\; h \cdot w \cdot \log|\det W|, \tag{6.9}$$

because the same W is applied at every one of the $h \cdot w$ spatial positions independently: the full Jacobian is block-diagonal with $h \cdot w$ copies of W on the diagonal, and determinants of block-diagonal matrices multiply. Do not forget the $h \cdot w$ factor—it is what makes channel-mixing layers carry real weight in the total log-likelihood at image resolutions.

A library call to compute $\det W$ would cost $\mathcal{O}(c^3)$ per training step per layer—tolerable in a demo, ruinous inside a deep flow evaluated millions of times. Glow's move, drawn from the same insight as the coupling layer, is to never face a determinant of an unstructured matrix at all: it *parameterizes* W directly in a factored form,

$$W = P\,L\,(U + \operatorname{diag}(\boldsymbol{s})), \tag{6.10}$$

where:

- P is a fixed permutation (a reshuffle of the channels, chosen once and never trained);
- L is lower-triangular with 1s on its diagonal;
- U is upper-triangular with 0s on its diagonal;

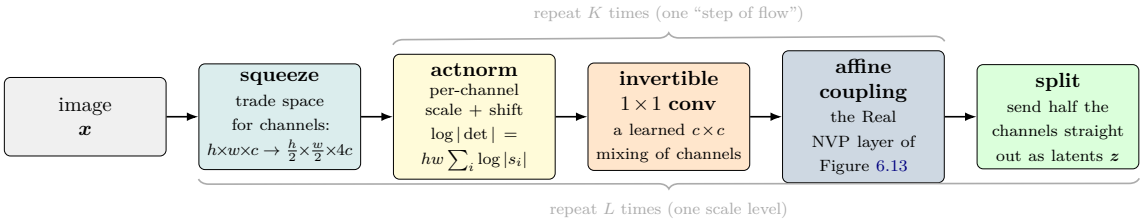
- $\boldsymbol{s}$ is a vector of learned scales placed on the diagonal.

The key point: the optimiser never learns W directly. It learns L, U and $\boldsymbol{s}$, and W is built from them each time it is needed. Any invertible matrix can be written this way, so nothing is lost.

Why bother? Because the determinant becomes trivial. The determinant of a product is the product of the determinants, and each piece is easy: $\det P = \pm 1$ (reshuffling does not change volume), $\det L = 1$ (triangular with a unit diagonal), and $\det(U + \mathrm{diag}(\boldsymbol{s}))$ is triangular, so it is the product of its diagonal:

$$\log|\det W| \;=\; \sum_{i=1}^{c} \log|s_i|. \tag{6.11}$$

The $\mathcal{O}(c^3)$ determinant collapses to reading off c numbers the optimizer already owns. (Why is P fixed rather than learned? Permutations are discrete; there is no useful gradient on "which permutation." The learnable parts are the continuous factors.)



What each new piece is for. *Actnorm* replaces batch normalisation, which a flow cannot use because it must be invertible per sample. *The* 1×1 *convolution* replaces Real NVP's hand-designed, fixed channel permutation with a *learned* one — the paper's actual contribution. *Squeeze* and *split* are inherited unchanged from Real NVP's multi-scale design.

Figure 6.17. One Glow block. Compared with Real NVP, exactly two things are new (yellow and orange); everything else is the previous model.

Model at a glance: Glow — Kingma & Dhariwal, 2018

In one line	Real NVP with two upgrades: a learned invertible channel mixing (a 1×1 convolution) in place of a fixed permutation, and actnorm in place of batch normalisation.
Problem it solves	Real NVP's fixed masks decide by hand which coordinates ever influence which. Glow lets the model decide, and gets much sharper samples for it.
One block	`actnorm` $\to$ invertible 1×1 conv $\to$ affine coupling, repeated K times per scale, with squeeze and split between scales (Figure 6.17).
Actnorm	A per-channel scale and shift, initialised so that the first minibatch has zero mean and unit variance per channel, then trained as ordinary parameters. A flow cannot use batch normalisation, because batch-norm's output for one sample depends on the other samples in the batch, which destroys per-sample invertibility.

Invertible 1×1 conv	At each spatial position, multiply the channel vector by a learned $c \times c$ matrix W. Its log-determinant is $h \cdot w \cdot \log\lvert\det W\rvert$, and W is *parameterised* as $PL(U + \mathrm{diag}(\boldsymbol{s}))$ so that $\log\lvert\det W\rvert = \sum_i \log\lvert s_i\rvert$ costs $\mathcal{O}(c)$.
Training	Exact maximum likelihood, as in Real NVP.
Sampling	One backwards pass, usually with a temperature $T < 1$: draw $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, T^2 I)$ to stay in the high-density core.
What it demonstrated	That flows can produce genuinely sharp, high-resolution faces (CelebA-HQ at 256×256) with smooth semantic interpolation between *real* photographs — something a GAN cannot do without an extra optimisation just to invert its generator.
Weaknesses	Very large models for the sample quality obtained; W can drift towards singularity during training and send the log-determinant to $-\infty$ (see the box below); still dimension-preserving.

Failure Modes

The DANGER note the lecture puts in bold. The LU factorization is an *existence* theorem for invertible matrices, not a stability guarantee for training dynamics. As Equation (6.11) shows, each s_i enters the objective through $\log\lvert s_i\rvert$, and nothing in the loss architecture prevents the optimizer from walking some s_i toward 0. When that happens, (i) the log-determinant dives to $-\infty$ and the loss to NaN, (ii) W becomes ill-conditioned and then singular, so the layer is no longer invertible and the flow's encode/decode correspondence breaks, and (iii) the run is usually unrecoverable without intervention. The remedy stated on the slide is unglamorous engineering, not math: log the smallest $\lvert s_i\rvert$ as a health metric, and when collapse is detected, reset to the last checkpoint and continue with reduced step sizes. This is the concrete face of the chapter's refrain that flows are "not very easy to train": the objective itself contains a cliff, and production training means building a fence at the cliff's edge.

Figure 6.18 (slide 20) shows two things worth understanding, not just admiring.

Temperature 0.7. During training, latents come from a standard Gaussian. At sampling time Glow draws from a *narrower* Gaussian instead — the same bell curve, squeezed to 70% of its width. Why? The edges of the Gaussian are rarely visited in training, so the flow is least reliable there, and that is where odd-looking faces come from. Staying near the centre gives cleaner faces, at the cost of less variety: unusual hairstyles and poses live out in those edges too. It is a dial between "crisp" and "varied", and Chapter 7 turns the same dial in diffusion models.

Interpolating between two real people. Take two real photographs, $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$. Because the flow is exactly invertible, you can run each one backwards and get its latent code directly: $\boldsymbol{z}_A = g(\boldsymbol{x}_A)$, $\boldsymbol{z}_B = g(\boldsymbol{x}_B)$. Walk in a straight line from $\boldsymbol{z}_A$ to $\boldsymbol{z}_B$, and run each point on the line forwards again. The result is a smooth morph from one face to the other, and every frame along the way is still a plausible face.

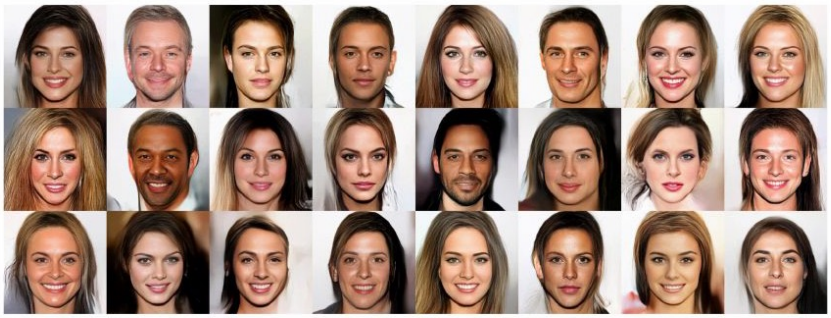


Figure 4: Random samples from the model, with temperature 0.7.

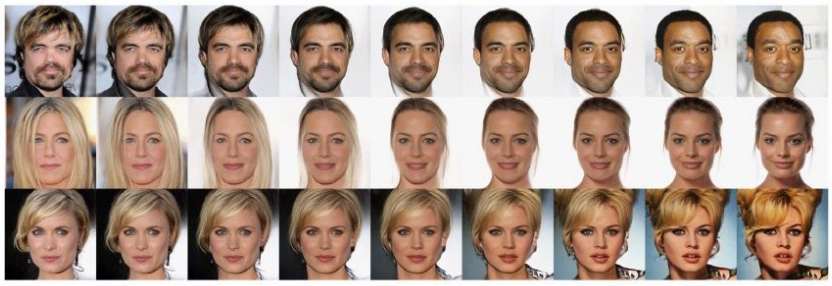


Figure 6.18. Glow on CelebA-HQ. *Top:* random samples drawn at temperature 0.7. *Bottom:* linear interpolation in latent space between two *real* photographs. Both are things only an exactly invertible model can do cheaply: the interpolation endpoints are the true latent codes of real images, obtained by simply running the network backwards, with no optimisation needed to find them.

How to read it: Walk along a bottom row. Every intermediate frame is a plausible face — the path never leaves the region the model considers realistic. A GAN cannot even begin this experiment without first solving an optimisation problem to invert its own generator.

Source: Lecture 04, slide 20 (Kingma and Dhariwal, 2018).

The key word is *directly*. A GAN has no backwards direction at all, so finding the latent code of a real photo means running an optimisation. A VAE has an encoder, but it only gives an approximate, noisy code. A flow gives the exact one, for free. Compare these samples with Real NVP's in Figure 6.16: the jump in quality comes from little more than learning the channel shuffling.

6.3.9 MAF: letting each coordinate see all of its predecessors

What is it?

Masked Autoregressive Flow (Papamakarios et al., 2017) keeps the affine transformation of Real NVP — scale and shift — but throws away the half-and-half split. Instead of conditioning on "the other half of the vector", each coordinate is conditioned on *every coordinate before it*.

Why was it introduced?

Look hard at a coupling layer and you notice it wastes most of its opportunity. Of D coordinates, d are copied through untouched, and the $D - d$ that are transformed are allowed to look only at those same d. Half the information is ignored, and half the output does nothing. Coupling flows compensate by stacking many layers with alternating masks, but that is paying in depth for something the dependency structure gave away for free.

The triangular-Jacobian argument of Section 6.3.4 never actually required a two-block split. It required only that output i depend on inputs 1 through i. The full autoregressive version of that condition is strictly more general — and strictly more expressive per layer.

The intuition

In plain words — from "the other half" to "everything so far"

A coupling layer asks: *given the left half of this vector, how should I rescale and shift the right half?* An autoregressive layer asks the same question D times, once per coordinate, each time with strictly more information: *given coordinates* 1 *to* $i - 1$*, how should I rescale and shift coordinate* i*?*

Coordinate 1 gets nothing to condition on, coordinate 2 gets one number, coordinate D gets $D - 1$ numbers. Nothing ever looks forwards, so the Jacobian is still triangular and the determinant is still the product of the diagonal. All that changed is that the conditioner is allowed to see more.

New words used from here on

Conditioner. The network that *produces* the scale and shift for a coordinate, as opposed to the arithmetic that applies them. In a coupling layer the conditioner is s, t applied to the identity half. In an autoregressive flow the conditioner is a masked network applied to all preceding coordinates. Every flow in this chapter has the same two-part shape: a conditioner (arbitrarily complex, never inverted) and a simple invertible transformation (affine, here) that the conditioner parameterises.

MADE. *Masked Autoencoder for Distribution Estimation* (Germain et al., 2015) — the trick that makes an autoregressive conditioner affordable. Naively, computing D different conditioners means D networks. MADE instead takes *one* network and multiplies its weight matrices by binary masks chosen so that output i is structurally incapable of depending on input j for $j \geq i$. One network, one pass, D correct conditioners. MAF is built out of stacked MADE blocks, which is where the "masked" in its name comes from.

Notation for this section and the next

MAF and IAF are easiest to keep apart when every symbol has exactly one meaning. These are the only symbols the two sections use.

Symbol	Name	In plain words
$\boldsymbol{x} = (x_1, \ldots, x_D)$	data	the thing we model: an image, a clip of audio
$\boldsymbol{u} = (u_1, \ldots, u_D)$	noise	a draw from $\mathcal{N}(\mathbf{0}, I)$. It is the same thing as $\boldsymbol{z}$ in Section 6.3.2; flow papers call it $\boldsymbol{u}$
$x_{<i}$, $u_{<i}$	"the past"	shorthand for $x_1, \ldots, x_{i-1}$ (or $u_1, \ldots, u_{i-1}$); empty when $i = 1$
μ_i	shift	the network's best guess for coordinate i
α_i, $\sigma_i = e^{\alpha_i}$	log-scale, scale	how unsure that guess is. The network outputs α_i; exponentiating keeps $\sigma_i > 0$ for free
conditioner		the one masked network (MADE) that outputs every (μ_i, α_i)

With these symbols, one layer of either model is the same scale-and-shift, $x_i = \mu_i + \sigma_i\, u_i$. The *only* thing that separates MAF from IAF is what the conditioner reads to produce μ_i and σ_i:

MAF: μ_i, σ_i are computed from $x_{<i}$ **IAF:** μ_i, σ_i are computed from $u_{<i}$.

Keep that one line in mind; every property of the two models follows from it.

A second way in: a Gaussian autoregressive model is already a flow

There is a route to MAF that needs no new machinery at all. Take an ordinary autoregressive model for continuous data, the chain rule with a Gaussian at every step:

$$p(\boldsymbol{x}) \;=\; \prod_{i=1}^{D} p(x_i \mid x_{<i}), \qquad p(x_i \mid x_{<i}) \;=\; \mathcal{N}\!\left(x_i;\; \mu_i(x_{<i}),\; \sigma_i(x_{<i})^2\right). \tag{6.12}$$

Now ask the plainest possible question: *how do you draw a sample from it?*

Step by step — sampling a Gaussian autoregressive model

1. Draw all the noise up front: $u_1, \ldots, u_D \sim \mathcal{N}(0, 1)$, independently.
2. Coordinate 1 has nothing to condition on, so μ_1, σ_1 are two learned constants. Set $x_1 = \mu_1 + \sigma_1 u_1$. (That is the usual way to turn a standard normal into $\mathcal{N}(\mu_1, \sigma_1^2)$.)
3. Feed x_1 to the network to get μ_2, σ_2. Set $x_2 = \mu_2 + \sigma_2 u_2$.
4. Feed x_1, x_2 to get μ_3, σ_3. Set $x_3 = \mu_3 + \sigma_3 u_3$. Continue until x_D.

In plain words — the observation that turns a sampler into a flow

Look at what that loop actually is. Once the noise $\boldsymbol{u}$ has been drawn, nothing random happens any more: every step is arithmetic. So $\boldsymbol{x}$ is a *deterministic function* of $\boldsymbol{u}$.

And the function can be undone. Hand it a finished $\boldsymbol{x}$ and you can read off every μ_i, σ_i (they only need $x_{<i}$, which you have), then recover each $u_i = (x_i - \mu_i)/\sigma_i$.

A deterministic, invertible map from Gaussian noise to data is exactly the definition of a normalizing flow. So **the sampler of any Gaussian autoregressive model is a flow**, and MAF is simply the name for that flow when the conditioner is a MADE network. This is where the phrase "autoregressive flow" comes from, and it is why MAF inherits the habits of an autoregressive model: fast to score, slow to sample.

Figure 6.19 shows what one such layer does to data you can see. The data is a curved "banana": x_2 is roughly $\frac{1}{2}x_1^2 - 1$, plus a little noise. A single MAF layer whose conditioner has learned that curve standardizes x_2 against its prediction, and the banana comes out as a round Gaussian cloud.

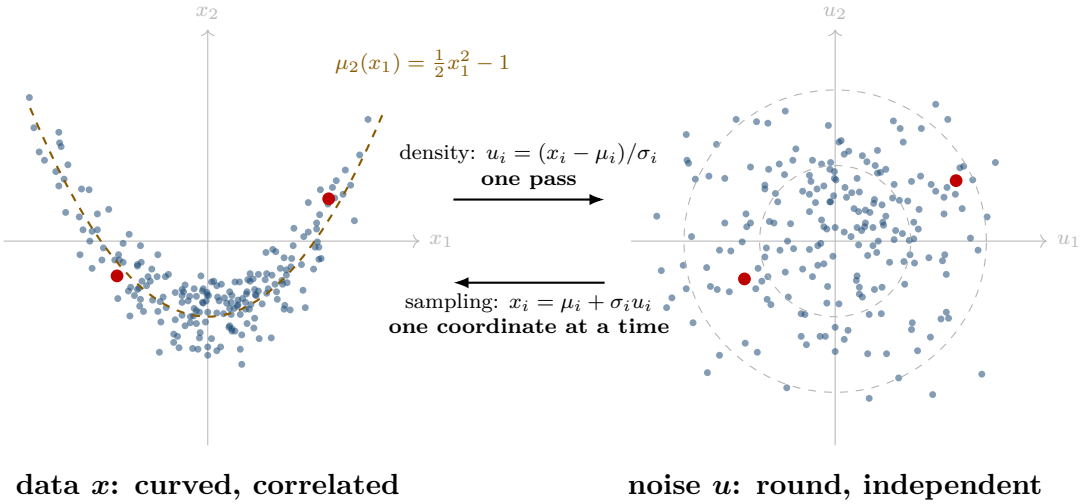


Figure 6.19. One MAF layer on two-dimensional data. The conditioner predicts x_2 from x_1 (dashed curve) with spread $\sigma_2 = 0.35$; subtracting the prediction and dividing by the spread straightens the banana into a standard Gaussian. The two red points are the same two samples on both sides. The 208 points were generated from exactly this model (tails beyond ± 2.4 trimmed for the plot), so the right-hand cloud is genuinely $\mathcal{N}(\mathbf{0}, I)$.

How to read it: Coordinate 1 *is only shifted and scaled by constants (here, not at all), so horizontal positions match across the two panels. All the work is vertical: each point moves by however far it sat above or below the curve, measured in units of* σ_2.

The mathematics

Write the flow in the *normalizing* direction (data → latent), which is the direction training uses. For each coordinate i,

$$u_i \;=\; \big(x_i - \mu_i(x_{1:i-1})\big)\, \exp\!\big(-\alpha_i(x_{1:i-1})\big), \tag{6.13}$$

where μ_i and α_i are the outputs of the masked conditioner network at position i: μ_i is a predicted mean for x_i given everything before it, and $\exp(\alpha_i)$ is a predicted standard deviation. Read the equation as a *standardisation*: subtract the prediction, then divide by the predicted

spread. If the conditioner is good, what is left over — u_i — is unpredictable residual noise, and the prior $p_U = \mathcal{N}(\mathbf{0}, I)$ says exactly that.

The generating direction inverts Equation (6.13) coordinate by coordinate:

$$x_i \;=\; u_i \,\exp\!\big(\alpha_i(x_{1:i-1})\big) \;+\; \mu_i(x_{1:i-1}). \tag{6.14}$$

Notice what the right-hand side needs: $x_{1:i-1}$. Those are outputs, not inputs. So x_1 must be computed before x_2, which must be computed before x_3, and so on — D sequential passes.

The Jacobian of Equation (6.13) is lower triangular with $\partial u_i/\partial x_i = \exp(-\alpha_i)$ on the diagonal, so

$$\log\!\left|\det J_g(\boldsymbol{x})\right| \;=\; -\sum_{i=1}^{D} \alpha_i(x_{1:i-1}), \tag{6.15}$$

and the training objective is the exact log-likelihood

$$\log p_X(\boldsymbol{x}) \;=\; \log p_U\big(g(\boldsymbol{x})\big) \;-\; \sum_{i=1}^{D} \alpha_i(x_{1:i-1}). \tag{6.16}$$

In plain words — the asymmetry, and why it is unavoidable

Equation (6.13) needs $x_{1:i-1}$ to compute u_i. When you are *training*, you already have the whole data vector $\boldsymbol{x}$ — it came from the dataset. So every μ_i and α_i can be computed at the same time, in one masked pass, and all D values of u_i fall out together.

When you are *sampling*, you have the whole of $\boldsymbol{u}$ but none of $\boldsymbol{x}$. You have to manufacture $\boldsymbol{x}$ one coordinate at a time, because coordinate i literally cannot be written down until coordinate $i-1$ exists. That is D passes — for an image, hundreds of thousands.

This is not an implementation weakness. It is the arithmetic of Equation (6.14), and no amount of engineering removes it.

Model at a glance: MAF (Masked Autoregressive Flow) Papamakarios, Pavlakou & Murray, 2017

In one line	An affine flow whose scale and shift for coordinate i are predicted from *all* previous coordinates, using a single masked network (MADE) so that all D predictions come out of one pass.
Problem it solves	Coupling layers leave half the coordinates untouched and let the conditioner see only half the input. MAF removes both restrictions, so one layer does much more work.
Key intuition	Standardise each coordinate against a prediction made from its predecessors. What is left over should be pure noise — and "should be pure noise" is exactly what the Gaussian prior asserts.
The layer	$u_i = (x_i - \mu_i(x_{<i}))\exp(-\alpha_i(x_{<i}))$, with μ, α from a MADE block (Germain et al., 2015).

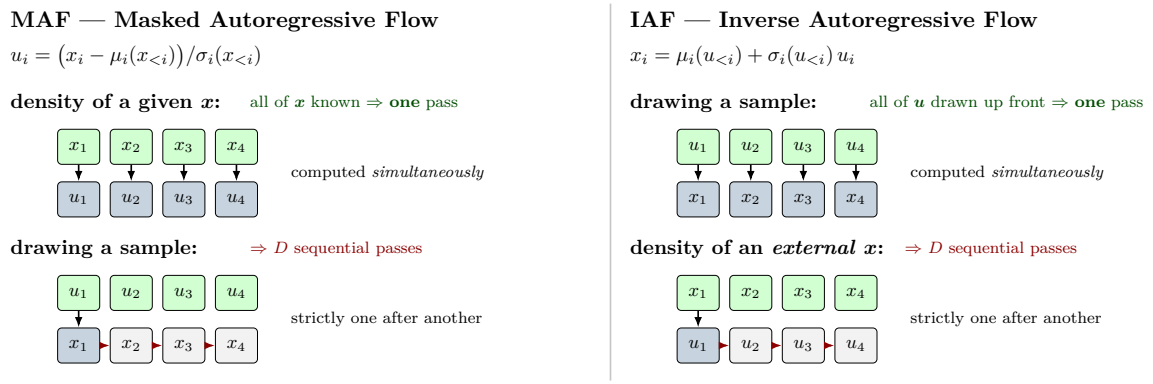


Figure 6.20. MAF and IAF are the same equation solved for a different variable, and their speed profiles are exact mirror images. MAF is fast in the direction you need for *training a density model*; IAF is fast in the direction you need for *generating samples*.

How to read it: Green boxes are values you already have; blue boxes are values computed in the current step; grey boxes are still unknown. Count the levels of red arrows: each one is a separate network pass.

Jacobian	Lower triangular; $\log\lvert\det J\rvert = -\sum_i \alpha_i$. Still $\mathcal{O}(D)$.
Cost, density direction	**One** pass. All of $\boldsymbol{x}$ is known, so every conditioner evaluates in parallel.
Cost, sampling direction	D sequential passes. Each coordinate must exist before the next can be computed.
Training	Exact maximum likelihood on Equation (6.16). Fast, because training only ever runs the fast direction.
Best at	*Density estimation* — scoring data, detecting outliers, comparing models by held-out log-likelihood. MAF was state-of-the-art on general-purpose density-estimation benchmarks when it appeared.
Bad at	Generating images. Sampling a $256 \times 256 \times 3$ image would need 196,608 sequential network passes.
Relation to the others	A coupling layer is the special case where the dependency pattern is a single two-block split rather than a full ordering — so Real NVP is, in this sense, a crippled but parallel MAF.
Successors	Neural Autoregressive Flows (Huang et al., 2018) and Neural Spline Flows (Durkan et al., 2019) replace the affine transformation with something more expressive while keeping the triangular structure.

Worked Example — MAF by hand in $\mathbb{R}^3$

Take $\boldsymbol{x} = (1.0,\ 2.0,\ 0.5)$ and suppose the masked conditioner produces

$$\mu_1 = 0,\ \ \alpha_1 = 0; \qquad \mu_2 = 0.5,\ \ \alpha_2 = 0.2; \qquad \mu_3 = -1.0,\ \ \alpha_3 = -0.3.$$

(Coordinate 1 has nothing to condition on, so its μ and α are learned constants.) Applying Equation (6.13):

$$u_1 = (1.0 - 0)e^0 = 1.0, \qquad u_2 = (2.0 - 0.5)e^{-0.2} \approx 1.5 \times 0.8187 = 1.228,$$

$$u_3 = (0.5 + 1.0)e^{0.3} \approx 1.5 \times 1.3499 = 2.025.$$

All three were computed *at the same time*, because every x_i was already known.
The log-determinant is $-(\alpha_1 + \alpha_2 + \alpha_3) = -(0 + 0.2 - 0.3) = 0.1$, so

$$\log p_X(\boldsymbol{x}) = \underbrace{-\tfrac{1}{2}(1.0^2 + 1.228^2 + 2.025^2) - \tfrac{3}{2}\log 2\pi}_{\log p_U(\boldsymbol{u})} + 0.1 \approx -3.304 - 2.757 + 0.1 = -5.96.$$

Now sample. Draw $\boldsymbol{u} = (0.3, -1.1, 0.8)$ and invert with Equation (6.14). Coordinate 1 needs nothing: $x_1 = 0.3\,e^0 + 0 = 0.3$. Only now can the conditioner be run to get μ_2, α_2 — they depend on x_1, which did not exist a moment ago. Suppose it returns $\mu_2 = 0.15, \alpha_2 = 0.1$; then $x_2 = -1.1\,e^{0.1} + 0.15 \approx -1.066$. Only now can μ_3, α_3 be computed. Three coordinates, three passes — and the passes cannot be merged, reordered or batched.

6.3.10 IAF: the same equation, solved the other way round

What is it?

Inverse Autoregressive Flow (Kingma et al., 2016) is MAF with the roles of the two variables exchanged. Where MAF conditions on previous values of $\boldsymbol{x}$, IAF conditions on previous values of $\boldsymbol{u}$.

Why was it introduced?

IAF was not invented to estimate densities. It was invented to fix a VAE.

Recall from Chapter 4 that a VAE's encoder outputs a diagonal Gaussian $q_\phi(\boldsymbol{z} \mid \boldsymbol{x})$ — a posterior whose coordinates are independent and whose shape is an axis-aligned ellipsoid. Real posteriors are neither. The gap between the true posterior and that simple family is a permanent tax on the ELBO, and no amount of training removes it.

The fix is to push the encoder's Gaussian sample through a flow, so the posterior becomes as flexible as the flow is (Rezende and Mohamed, 2015). But this use imposes an unusual requirement. Inside the ELBO you need $\log q(\boldsymbol{z} \mid \boldsymbol{x})$ evaluated *at the sample you just drew* — never at an arbitrary externally supplied point. So you need sampling to be fast, and you need the density only along the path you already walked.

MAF is exactly the wrong way round for that: fast to score, slow to sample. IAF turns it around.

The mathematics

IAF's transformation, written in the *generating* direction, is

$$x_i \;=\; \mu_i(u_{1:i-1}) \;+\; \sigma_i(u_{1:i-1})\, u_i. \tag{6.17}$$

Compare it, term for term, with MAF's generating direction, Equation (6.14): the arithmetic is identical, but the conditioner is fed $u_{1:i-1}$ instead of $x_{1:i-1}$.

That one substitution flips every cost in the model. When sampling, the whole of $\boldsymbol{u}$ is drawn up front from the prior — so all the conditioners can be evaluated at once, and all D coordinates of $\boldsymbol{x}$ come out of a single pass. When you want the density of some $\boldsymbol{x}$ the model did not generate, you must first recover $\boldsymbol{u}$, and u_i depends on $u_{1:i-1}$, so you are back to D sequential steps.

Key takeaway — the duality, stated once

> MAF and IAF are not two ideas. They are one equation read in two directions. Every "MAF is fast at X" has a mirror "IAF is fast at the opposite of X". Formally, the density direction of an IAF *is* the sampling direction of a MAF, and vice versa — so anything you prove about one transfers to the other by relabelling.

Seeing the one-wire difference

Figure 6.21 draws one step of each model, for coordinate $i = 3$, with the data row on top and the noise row underneath. The arithmetic box is identical. Only the wire feeding the conditioner has moved, from the top row to the bottom row. Flip the picture upside down and the two models swap places, which is the precise sense in which IAF is MAF "transposed".

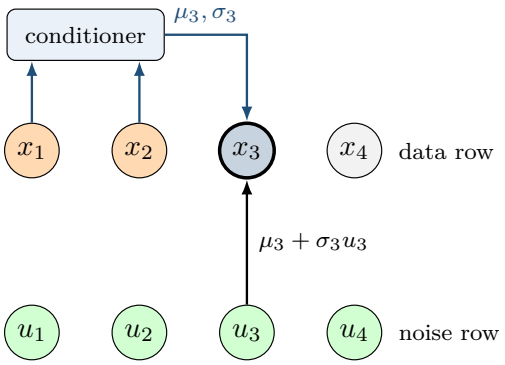
In plain words — one rule decides every speed

> A direction of a flow runs in **one pass** exactly when everything the conditioner reads is *already in hand* at the start of that direction. Otherwise it runs one coordinate at a time.
>
> MAF's conditioner reads $\boldsymbol{x}$. Scoring data starts from $\boldsymbol{x}$, so scoring is fast; sampling starts from $\boldsymbol{u}$, so sampling is slow. IAF's conditioner reads $\boldsymbol{u}$, so everything flips.

	conditioner reads	draw a sample	score a given $\boldsymbol{x}$
MAF	$x_{<i}$	D sequential steps	**one pass**
IAF	$u_{<i}$	**one pass**	D sequential steps
IAF, scoring *its own* sample	$u_{<i}$ (kept)		**one pass**
Coupling (Real NVP)	$x_{1:d} = u_{1:d}$	**one pass**	**one pass**

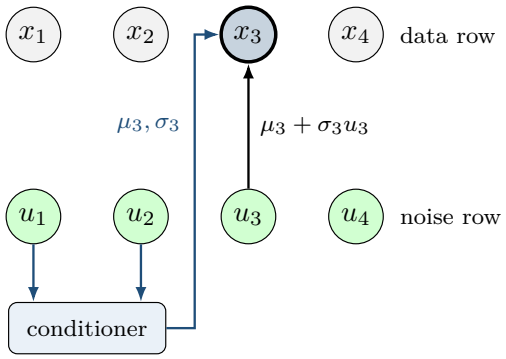
The last two rows deserve a sentence each. An IAF can score a point *it generated itself* in one pass, because it drew the noise $\boldsymbol{u}$ that produced that point and can simply keep it:

MAF: conditioner reads $x_{<i}$



sampling: x_1, x_2 must be *generated first* ⇒ one step per coordinate, slow
density of data: all of $\boldsymbol{x}$ is given ⇒ one pass, fast

IAF: conditioner reads $u_{<i}$



sampling: all of $\boldsymbol{u}$ is drawn up front ⇒ one pass, fast
density of outside data: u_1, u_2 must be *recovered first* ⇒ slow

Figure 6.21. The whole difference between MAF and IAF is one wire (blue). Both compute $x_3 = \mu_3 + \sigma_3 u_3$. MAF feeds the conditioner the *earlier outputs* x_1, x_2; IAF feeds it the *earlier inputs* u_1, u_2. Whichever row the conditioner reads is the row you must already hold before that step can run.

How to read it: Green nodes are available as soon as sampling starts (the noise is drawn first). Orange nodes are values sampling has to produce before step 3 can run. The blue node is the coordinate being computed.

every $\alpha_i(u_{<i})$ was already computed on the way out. That loophole is the whole reason IAF is useful, as the worked example below and Parallel WaveNet show.

The coupling layer is the case where MAF and IAF coincide. Its conditioner reads the copied half, and a copied half is the same in $\boldsymbol{x}$ and $\boldsymbol{u}$, so it is in hand at the start of *either* direction. That makes a coupling layer simultaneously a block-wise MAF and a block-wise IAF, and it is why Real NVP is fast both ways. The price is the one Section 6.3.9 began with: per layer, half of the coordinates are not transformed at all.

Worked Example — IAF by hand in $\mathbb{R}^3$

Draw $\boldsymbol{u} = (0.3, -1.1, 0.8)$. The conditioner reads only $\boldsymbol{u}$, so *all* its outputs can be computed immediately, in one pass. Suppose it returns

$$\begin{aligned} \mu_1 &= 0, & \alpha_1 &= 0 & &\text{(learned constants)} \\ \mu_2 &= 0.15, & \alpha_2 &= 0.1 & &\text{(from } u_1\text{)} \\ \mu_3 &= -0.4, & \alpha_3 &= -0.2 & &\text{(from } u_1, u_2\text{)}. \end{aligned}$$

Then every coordinate of the sample comes out at once:

$$x_1 = e^0(0.3) = 0.3, \quad x_2 = 0.15 + e^{0.1}(-1.1) \approx -1.066, \quad x_3 = -0.4 + e^{-0.2}(0.8) \approx 0.255.$$

Compare the MAF sampling example above: there, μ_3, α_3 could not be computed until x_2 existed. Here they were available before any x_i was.

Score the sample we just made. We still hold $\boldsymbol{u}$ and every α_i, so no extra passes are needed. Since $\partial x_i / \partial u_i = e^{\alpha_i}$, the change of variables gives

$$\begin{aligned} \log p_X(\boldsymbol{x}) &= \log p_U(\boldsymbol{u}) - \textstyle\sum_i \alpha_i \\ &= \underbrace{-\tfrac{1}{2}(0.3^2 + 1.1^2 + 0.8^2) - \tfrac{3}{2}\log 2\pi}_{-0.970 - 2.757} - (0 + 0.1 - 0.2) \;\approx\; -3.63. \end{aligned}$$

Now score someone else's point, say $\boldsymbol{x} = (1.0, 2.0, 0.5)$. We do not have its $\boldsymbol{u}$, so we must recover it. $u_1 = (1.0 - \mu_1)/e^{\alpha_1} = 1.0$ is immediate. But μ_2, α_2 depend on u_1, so a second conditioner pass is needed before u_2 can be computed, and a third before u_3. Three coordinates, three sequential passes: exactly MAF's sampling cost, now paid on the scoring side.

Model at a glance: IAF (Inverse Autoregressive Flow) Kingma, Salimans, Józefowicz, Chen, Sutskever & Welling, 2016

In one line	MAF with the conditioner reading the *latent* coordinates instead of the data coordinates, which makes sampling one pass and external density evaluation D passes.
Problem it solves	A VAE's diagonal-Gaussian posterior is too simple and permanently loosens the ELBO. IAF makes the posterior flexible while keeping sampling cheap — which is the only speed a VAE posterior needs.
Key intuition	Inside the ELBO you only ever need the density of *your own* samples. IAF is built around that asymmetry instead of fighting it.
The layer	$x_i = \mu_i(u_{<i}) + \sigma_i(u_{<i})\, u_i$, again with a MADE-style masked conditioner.
Jacobian	Lower triangular; $\log\lvert\det\rvert = \sum_i \log \sigma_i(u_{<i})$.
Cost, sampling direction	**One** pass — all of $\boldsymbol{u}$ is available up front.

Cost, density of external data	D sequential passes.
Used for	Flexible variational posteriors in VAEs; and as a fast *generator* distilled from a slow autoregressive teacher.
The famous application	Parallel WaveNet (van den Oord et al., 2018a): an IAF student distilled from a trained WaveNet teacher, so speech is sampled in one pass. It works precisely *because* the student's own samples are the only thing either network has to score. See Section 6.3.10.
Weaknesses	Cannot be trained by straightforward maximum likelihood on a dataset, because that requires scoring external data — the slow direction. It needs either a variational objective or a teacher.
Relation to MAF	The same equation, read the other way. See Figure 6.20.

Common Confusion

"If MAF and IAF are the same equation, why not just train a MAF and then run it backwards when you want fast sampling?" Because "backwards" is the slow direction, by construction. Running a trained MAF in reverse gives you a correct sampler that takes D passes — it does not give you an IAF. The two models differ in *which variable the conditioner reads*, and that is fixed at the moment you wire up the network, not something you choose at run time. What you *can* do is what Parallel WaveNet does: train the fast-to-score model, then use it as a teacher to train a separate fast-to-sample model.

Getting both speeds: probability density distillation

MAF is fast to train and slow to sample; IAF is fast to sample and slow to train on data. Parallel WaveNet (van den Oord et al., 2018a) gets both by using each model only in its fast direction.

The problem it solved was WaveNet, an autoregressive model of raw audio. Audio runs at tens of thousands of samples per second, and WaveNet generates them one at a time, so the network has to be run tens of thousands of times, in sequence, for every second of speech. Training WaveNet is cheap, because scoring a real clip is one parallel pass: it behaves like a MAF. Sampling is the slow direction.

The fix is to train a *student* S to imitate a trained *teacher* T by minimizing

$$\mathrm{KL}(S \,\|\, T) \;=\; \mathbb{E}_{\boldsymbol{x}\sim S}\big[\,\log S(\boldsymbol{x}) \;-\; \log T(\boldsymbol{x})\,\big]. \tag{6.18}$$

Estimating this with samples needs three things per training step, and the point of the method is that each one lands on somebody's fast direction:

Needed for Eq. (6.18)	Who does it	Why it is one pass
draw $\boldsymbol{x} \sim S$	IAF student	IAF sampling reads only $\boldsymbol{u}$
$\log S(\boldsymbol{x})$ at that same $\boldsymbol{x}$	IAF student	it is the student's *own* sample, so $\boldsymbol{u}$ and every α_i are already in hand
$\log T(\boldsymbol{x})$	teacher (WaveNet)	$\boldsymbol{x}$ is fully known, so every conditional is computed in parallel, as in MAF

The direction of the KL matters. The reverse, $\mathrm{KL}(T \,\|\, S)$, would need samples from the teacher and the student's density on the teacher's samples: both of those are slow directions.

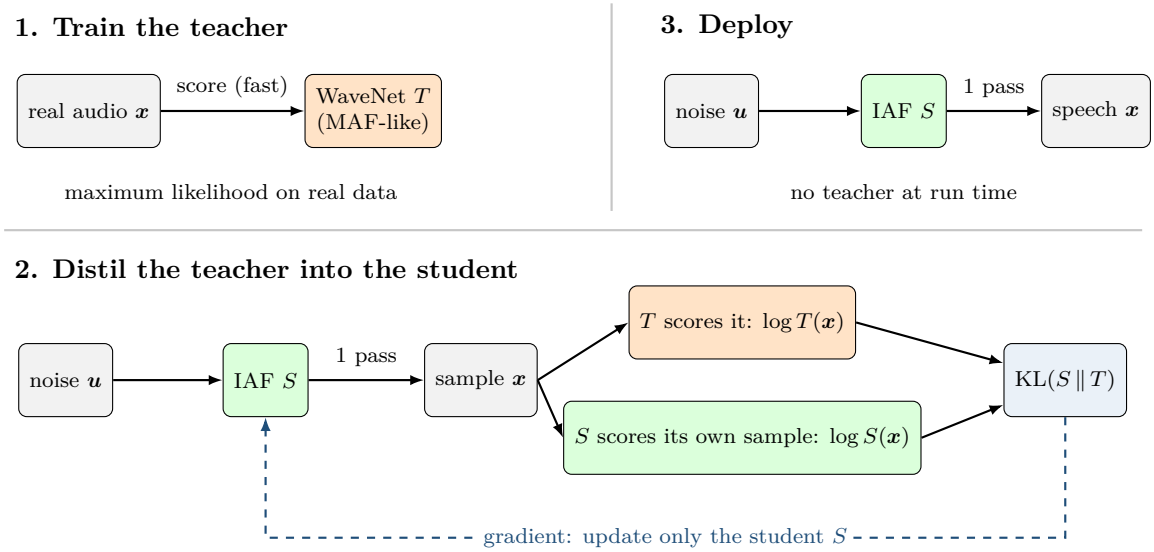


Figure 6.22. Probability density distillation, the recipe behind Parallel WaveNet. Each network is only ever run in its fast direction: the teacher only scores, the student only samples and scores its own samples.

How to read it: Orange is the slow-to-sample teacher, green is the fast-to-sample student. The teacher is frozen after stage 1: gradients pass through *it to reach $\boldsymbol{x}$, but only the student's weights change (dashed arrow).*

The paper reports a speed-up of more than 1000× over the original WaveNet, generating speech more than twenty times faster than real time, and the system was deployed in production (van den Oord et al., 2018a). The KL term was not the whole objective: the paper adds a power loss, a perceptual loss and a contrastive loss on top of Equation (6.18), so treat the equation as the core idea rather than the full recipe.

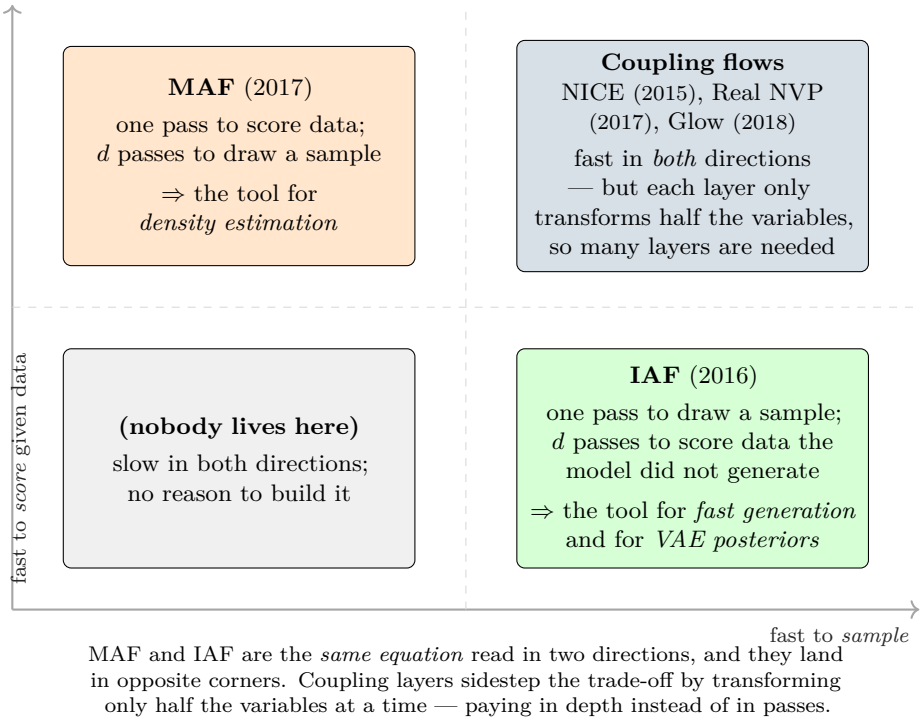


Figure 6.23. The design space of normalizing flows, on the two axes that decide which one you can actually use. Note that none of this is about accuracy; it is entirely about which direction of the map you can afford to run.

How to read it: Find the corner that matches your use case first, then pick the model. Training a density model? Top-left or top-right. Generating in real time? Bottom-right or top-right.

6.3.11 The five flows side by side

In plain words — which one would I actually use?

You want to score data — outlier detection, model comparison, held-out log-likelihood: **MAF**. It is the most expressive per layer and its fast direction is the one you need.

You want to generate images and you want exact likelihoods too: **Glow**, or Real NVP if you want something simpler. Both directions cost one pass; you pay in depth and parameter count.

You want a flexible posterior inside a VAE, or a one-pass generator distilled from a slow teacher: **IAF**.

You want to understand the field: **NICE**, because everything else is a patch on it.

And the honest answer for images in practice today: none of them. Diffusion models (Chapter 7) give far better samples, at the cost of the exact likelihood and one-pass sampling that this whole section worked so hard to preserve. Flows survive where the *number* matters more than the picture — density estimation, simulation-based inference,

Table 6.1. The five normalizing flows developed in this chapter. Read the table down the "What it fixed" column to get the story of the field in five lines; read across a row to get one model. "D passes" means one network evaluation per data dimension, which for images is hundreds of thousands.

Model	Year	Transformation	Score data	Draw sample	What it fixed	What it left broken
NICE	2015	additive coupling, $\det J = 1$	1 pass	1 pass	first deep net that is invertible *by construction* with a free determinant	volume preserving, so it cannot concentrate mass
Real NVP	2017	affine coupling	1 pass	1 pass	added a per-coordinate scale, so $\det J \neq 1$	fixed hand-designed masks; only half the coordinates move per layer
Glow	2018	affine coupling + learned 1×1 conv	1 pass	1 pass	learned the channel mixing; actnorm replaces batchnorm	huge models; W can drift to singular and kill the run
MAF	2017	affine, autoregressive in $\boldsymbol{x}$	1 pass	D passes	conditioner sees *all* predecessors, not half the vector	sampling is sequential, so useless as an image generator
IAF	2016	affine, autoregressive in $\boldsymbol{u}$	D passes	1 pass	made the fast direction the *sampling* one	cannot be trained by plain maximum likelihood on a dataset

variational posteriors, and normalising layers inside larger models (Papamakarios et al., 2021).

Common Confusion

"Exact likelihood must mean the model knows what is real." It does not, and the counterexample is famous. Nalisnick et al. (2019) showed that flows and other exact-likelihood models trained on CIFAR-10 assign *higher* likelihood to SVHN images — data from a completely different distribution that the model has never seen. Likelihood in high dimensions is dominated by low-level statistics such as local smoothness, and SVHN images happen to be smoother than CIFAR-10 ones. A high likelihood is not a statement that an input looks like the training data, and Theis et al. (2016) make the broader point:

log-likelihood, sample quality and representation quality are three loosely related axes, and a model can be excellent on one while being poor on the others.

6.3.12 Conditional flows and super-resolution

Everything so far models $p(\boldsymbol{x})$ unconditionally: "give me *an* image." Many useful problems are instead of the form "give me an image *that fits this*": a sharp photo that fits a blurry thumbnail, a colour photo that fits a grey one, a segmentation that fits a scan. The final flow topic (slides 21–23) is *Learning Likelihoods with Conditional Normalizing Flows* by Winkler, Worrall, Hoogeboom, and Welling (Winkler et al., 2019), and its idea fits in one sentence: *keep the flow exactly as it is, but let every one of its knobs also look at the condition.*

A notation flip, and a way to remember it. Slide 22 carries a warning the book repeats here: the paper follows the supervised-learning convention, so the *condition* is $\boldsymbol{x}$ and the *data being modelled* is $\boldsymbol{y}$ — the reverse of every earlier section of this chapter. For this subsection only we follow the paper. The mnemonic is the one from regression: $\boldsymbol{x}$ is what you are *given* (the question), $\boldsymbol{y}$ is what you are *asked for* (the answer). In super-resolution, $\boldsymbol{x}$ is the low-resolution thumbnail and $\boldsymbol{y}$ is the sharp image.

Why model a distribution at all? A two-pixel example

The cheapest way to "undo" a downsampling is a regression network trained with squared error, $\min \|f(\boldsymbol{x}) - \boldsymbol{y}\|^2$. It is worth seeing precisely why that fails, because the failure is the whole reason conditional *generative* models exist.

Worked Example — The smallest super-resolution problem there is

Take a "sharp image" of just two pixels, $\boldsymbol{y} = (y_a, y_b)$, with 0 = black and 1 = white. The "thumbnail" is one pixel, their average: $x = \frac{1}{2}(y_a + y_b)$. Suppose the world contains only one kind of sharp patch: a hard edge, which runs either way with equal probability,

$$\boldsymbol{y} = (0,1) \ \text{(black|white)} \quad \text{or} \quad \boldsymbol{y} = (1,0) \ \text{(white|black)}, \qquad \text{each with probability } \tfrac{1}{2}.$$

Both give the same thumbnail, $x = 0.5$: a single mid-grey pixel. Seeing $x = 0.5$, the honest answer is "an edge, and I cannot tell which way."

What squared error learns. The prediction that minimises the expected squared error is the conditional mean, $\mathbb{E}[\boldsymbol{y} \mid x = 0.5] = \frac{1}{2}(0,1) + \frac{1}{2}(1,0) = (0.5, 0.5)$ — two grey pixels. Score the candidates:

- Predict the mean $(0.5, 0.5)$: the squared error is $0.5^2 + 0.5^2 = 0.5$ whichever edge is true. Expected error 0.5.
- Predict a real edge, say $(0,1)$: error 0 half the time and $1^2 + 1^2 = 2$ the other half. Expected error 1.0.

Squared error prefers the grey answer by a factor of two, even though *no sharp patch in the world looks like it.* The edge has been averaged away. That is super-resolution blur, reduced to two numbers.

What a conditional density learns. A model of $p(\boldsymbol{y} \mid x = 0.5)$ puts two bumps of probability, one at $(0, 1)$ and one at $(1, 0)$, and almost none at $(0.5, 0.5)$. Asked for a sample, it returns one of the two edges — each time a crisp one, and over many draws both directions equally often.

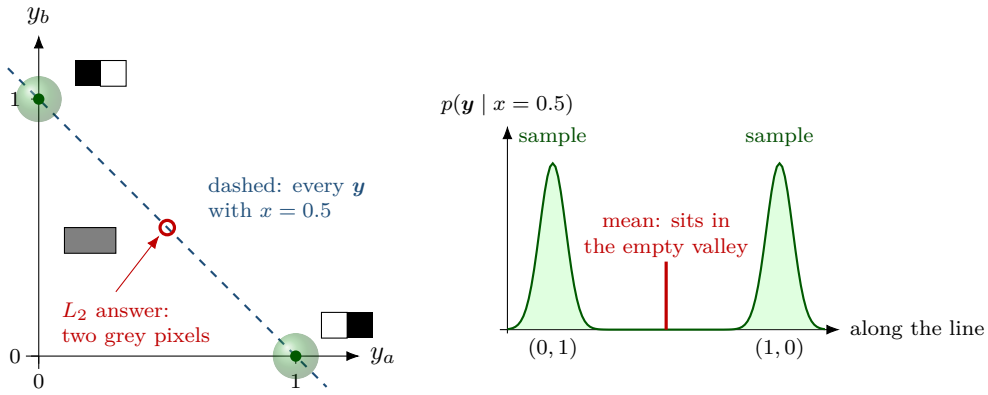


Figure 6.24. The two-pixel super-resolution problem. *Left:* every sharp image $\boldsymbol{y} = (y_a, y_b)$ whose average is the grey thumbnail $x = 0.5$ lies on the dashed line. Real data sits only at its two ends (green: a black|white or a white|black edge). Squared-error regression lands on the red point in the middle, two grey pixels, which is not a real image. *Right:* the same line, now showing the conditional density. It has two peaks and a valley in between, and the mean falls in the valley. A conditional flow learns the whole curve and samples from a peak.

How to read it: keep the red point and the green points apart in your head. Every blurry super-resolution output you have ever seen is a red point in a space with millions of dimensions.

Real images are this example repeated millions of times over: every patch of fur, grass, or text in a thumbnail is consistent with many sharp versions, and their average is always smoother than any one of them. The fix is to stop predicting a single $\boldsymbol{y}$ and model the whole conditional density $p(\boldsymbol{y} \mid \boldsymbol{x})$. A flow can do that exactly, and the surprise is how little has to change.

The recipe: give every knob a view of the condition

Recall what an unconditional flow is: a simple Gaussian blob $\boldsymbol{z}$, and an invertible machine that reshapes the blob into the data distribution. A conditional flow is *the same machine with a dial on the side.* Turn the dial to a different $\boldsymbol{x}$ and the machine reshapes the blob into

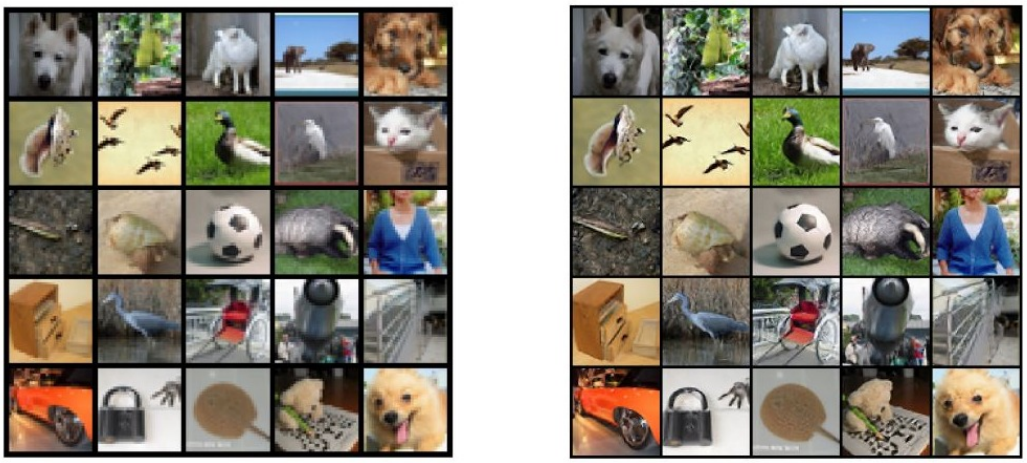


Figure 6.25. Conditional flows applied to super-resolution. *Left:* the low-resolution input. *Right:* a sample from the conditional flow. The point is not merely that the right-hand image is sharper — it is that the model produces a *sample* from the distribution of plausible sharp images, rather than their average. An L_2-trained regressor, asked the same question, has no choice but to output the mean of everything consistent with the input, and the mean of many sharp images is a blurry one: the red point of Figure 6.24, in millions of dimensions.

How to read it: Look at fine texture — fur, grass, fabric. A regressor would render those as smooth grey; a generative model has to commit to one plausible answer.

Source: Lecture 04, slide 23 (Winkler et al., 2019).

a different distribution: the distribution of sharp images that fit *that* thumbnail. Concretely, every piece that was a function of $\boldsymbol{z}$ alone becomes a function of $\boldsymbol{z}$ *and* $\boldsymbol{x}$:

$$\text{Conditional prior:} \quad p(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}\big(\boldsymbol{z};\, \boldsymbol{\mu}(\boldsymbol{x}),\, \sigma^2(\boldsymbol{x})\big), \tag{6.19}$$

$$\text{Conditional split prior:} \quad p(\boldsymbol{z}_1 \mid \boldsymbol{z}_0, \boldsymbol{x}) = \mathcal{N}\big(\boldsymbol{z}_1;\, \boldsymbol{\mu}(\boldsymbol{z}_0, \boldsymbol{x}),\, \sigma^2(\boldsymbol{z}_0, \boldsymbol{x})\big), \tag{6.20}$$

$$\text{Conditional coupling:} \quad \boldsymbol{y}_0 = \exp\big(s(\boldsymbol{z}_1, \boldsymbol{x})\big) \odot \boldsymbol{z}_0 + t(\boldsymbol{z}_1, \boldsymbol{x}), \qquad \boldsymbol{y}_1 = \boldsymbol{z}_1, \tag{6.21}$$

where $\boldsymbol{\mu}(\cdot)$, $\sigma^2(\cdot)$, $s(\cdot)$, $t(\cdot)$ are ordinary neural networks that now take $\boldsymbol{x}$ as an extra input. Read each line as an answer to a question:

- **Conditional prior (6.19): where does the blob start?** Before any reshaping, the thumbnail already says a lot — this region is dark, that one is bright. So the starting Gaussian is no longer centred at $\mathbf{0}$ with unit width. It is centred at $\boldsymbol{\mu}(\boldsymbol{x})$ with width $\sigma(\boldsymbol{x})$: narrow where the thumbnail leaves little doubt, wide where it leaves a lot.
- **Conditional coupling (6.21): how is the blob reshaped?** Exactly the Real NVP affine coupling of Equation (6.7), written in the generative direction $\boldsymbol{z} \to \boldsymbol{y}$: half the variables ($\boldsymbol{z}_1$) pass through untouched, the other half ($\boldsymbol{z}_0$) is scaled by e^s and shifted by t. The only change is that the scale and shift networks now read the thumbnail too.

- **Conditional split prior (6.20): what about the variables that leave early?** In the multi-scale architecture (Section 6.3.7) some latents are factored out part-way through. Their Gaussian is also predicted, now from the variables that stay ($\boldsymbol{z}_0$) *and* from the thumbnail. That is, the coarse structure already decided, together with $\boldsymbol{x}$, sets the likely range of the details. (Here $\boldsymbol{z}_0, \boldsymbol{z}_1$ label the two halves at one split, not the $\boldsymbol{z}_1, \boldsymbol{z}_2, \boldsymbol{z}_3$ of Figure 6.14.)

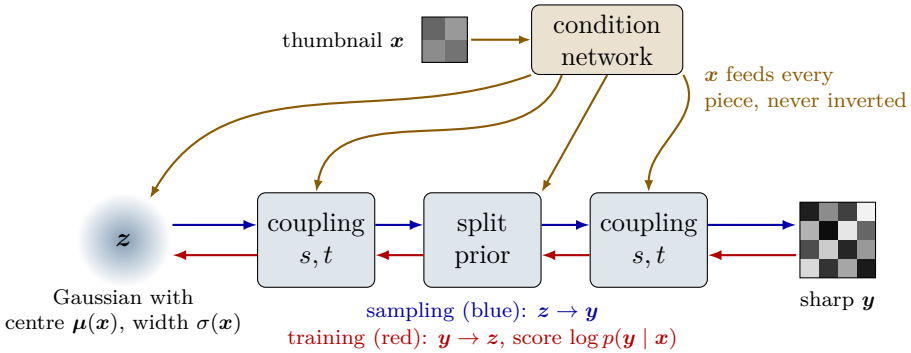


Figure 6.26. A conditional normalizing flow. The flow itself (blue blocks) is the ordinary invertible stack. The thumbnail $\boldsymbol{x}$ is read by a condition network whose features go into the prior, every coupling layer's s, t networks, and every split prior (orange arrows). Training runs right to left (red): push the real sharp image back to $\boldsymbol{z}$ and score it. Sampling runs left to right (blue): draw $\boldsymbol{z}$ from the $\boldsymbol{x}$-dependent Gaussian and push it forward.

How to read it: the orange arrows only ever point into *the flow. Nothing flows back into $\boldsymbol{x}$, and nothing ever has to invert the condition network. That is why it can be any network at all.*

Worked Example — A one-layer conditional flow in 1-D, with numbers

Strip everything down to one pixel y and one layer:

$$z \sim \mathcal{N}(0,1), \qquad y = e^{s(x)}\, z + t(x).$$

For one particular thumbnail x, suppose the networks output $t(x) = 0.5$ ("this region is mid-bright") and $e^{s(x)} = 0.2$, so $s(x) = \ln 0.2 \approx -1.609$ ("and I am fairly sure"). Then $p(y \mid x)$ is simply a bell curve centred at 0.5 with width 0.2. Change x and the networks move and widen the bell. That is all a single conditional layer can do.

Training (right to left). The real sharp pixel is $y = 0.7$. Invert: $z = (0.7 - 0.5)/0.2 = 1.0$. Score it with the change-of-variables formula (6.5):

$$\log p(y \mid x) = \underbrace{\log \mathcal{N}(1.0;\, 0, 1)}_{-0.919 - 0.5 \,=\, -1.419} \; \underbrace{-\, s(x)}_{+1.609} \;=\; 0.190.$$

Check it directly: a bell of width 0.2 at 0.5 has density $\varphi(1)/0.2 = 0.242/0.2 = 1.21$ at $y = 0.7$, and $\ln 1.21 = 0.19$. Gradient descent on -0.190 nudges $t(x)$ toward 0.7 and adjusts $s(x)$ toward the right width.

Sampling (left to right). Draw $z = -0.5$: $y = 0.5 + 0.2(-0.5) = 0.4$. Draw $z = 1.3$: $y = 0.76$. Different draws give different, equally legitimate answers for the same x.
Why stack layers. One layer gives one bump, but the two-pixel edge of Figure 6.24 needs *two* bumps. Stacking coupling layers lets the flow bend the single Gaussian into that shape (compare the MAF banana of Figure 6.19), with $\boldsymbol{x}$ deciding where the bumps go.

Training and sampling, the whole loop. Training runs the flow in the likelihood direction $\boldsymbol{y} \to \boldsymbol{z}$ and minimises $-\log p(\boldsymbol{y} \mid \boldsymbol{x})$. The change-of-variables formula applies unchanged, with $\boldsymbol{x}$ carried along as a fixed input:

$$\log p(\boldsymbol{y} \mid \boldsymbol{x}) \;=\; \log \mathcal{N}(\boldsymbol{z};\, \boldsymbol{\mu}(\boldsymbol{x}), \sigma^2(\boldsymbol{x})) \;-\; \sum_{\text{layers}} \sum_{j} s_j(\cdot\,, \boldsymbol{x}), \qquad \boldsymbol{z} = f^{-1}(\boldsymbol{y}; \boldsymbol{x}). \tag{6.22}$$

Sampling runs the generative direction: draw $\boldsymbol{z} \sim p(\boldsymbol{z} \mid \boldsymbol{x})$, push it through f, and out comes a $\boldsymbol{y}$ that fits $\boldsymbol{x}$.

Step by step — Conditional flow for super-resolution, end to end

Step 1. Make the data from raw images. Take any sharp image $\boldsymbol{y}$. Downsample it to get $\boldsymbol{x}$. That is one training pair, and no human was involved.

Step 2. Read the condition. Run $\boldsymbol{x}$ through the condition network once. Its features will be handed to every piece of the flow.

Step 3. Go backwards. Push the *real* $\boldsymbol{y}$ through the inverse flow, layer by layer, to get $\boldsymbol{z}$. At each coupling layer, add up its s values. That sum is the log-volume change.

Step 4. Score. Compute Equation (6.22): how likely is $\boldsymbol{z}$ under the $\boldsymbol{x}$-dependent Gaussian, corrected for the volume change. Minimise the negative. Every network, including the condition network, gets its gradient from this one number.

Step 5. Generate. For a new thumbnail $\boldsymbol{x}$: draw $\boldsymbol{z}$ from $\mathcal{N}(\boldsymbol{\mu}(\boldsymbol{x}), \sigma^2(\boldsymbol{x}))$, run the flow forward, read off $\boldsymbol{y}$. Draw again for a different plausible $\boldsymbol{y}$.

```
def log_prob(y, x):                          # training: sharp y, thumbnail x
    h = cond_net(x)                          # read the condition once
    z, sum_s = f_inverse(y, h)               # y -> z, adding up each layer's s
    mu, sigma = prior_net(h)                 # x-dependent Gaussian
    return normal_log_prob(z, mu, sigma) - sum_s

def sample(x):                               # generation: new thumbnail x
    h = cond_net(x)
    mu, sigma = prior_net(h)
    z = mu + sigma * randn_like(mu)          # a fresh draw each call
    return f_forward(z, h)                   # z -> y

# data: pairs built from unlabeled images
for y in sharp_images:
    x = downsample(y)
    loss = -log_prob(y, x).mean()
    loss.backward(); optimizer.step()
```

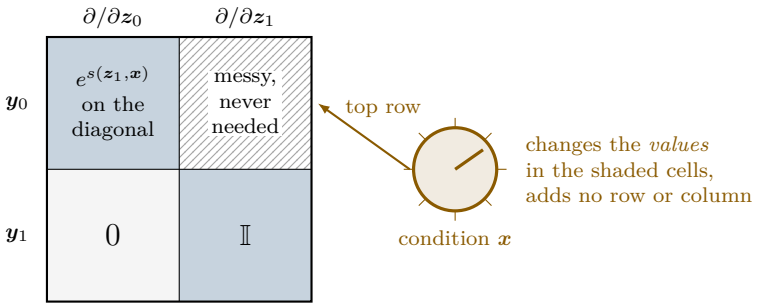
Listing 6.1. A conditional flow's two directions in PyTorch-like pseudocode. The condition `x` is an input to every network and is never inverted; `f_inverse` returns the summed s values as it goes.

Why adding a condition costs nothing

A natural worry: flows live or die by a cheap Jacobian determinant. Feeding a big image $\boldsymbol{x}$ into every coupling layer must surely wreck that. It does not, and the reason is one sentence: *the Jacobian is taken with respect to the variable being transformed, $\boldsymbol{z}$, and $\boldsymbol{x}$ is not that variable.* From the flow's point of view, $\boldsymbol{x}$ is a fixed setting, like the weights. It changes the *numbers* in the Jacobian but not its *shape*. Differentiating Equation (6.21) with respect to $\boldsymbol{z} = (\boldsymbol{z}_0, \boldsymbol{z}_1)$:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}^\top} = \begin{bmatrix} \operatorname{diag}\big(\exp s(\boldsymbol{z}_1, \boldsymbol{x})\big) & \partial \boldsymbol{y}_0 / \partial \boldsymbol{z}_1^\top \\ 0 & \mathbb{I} \end{bmatrix}, \qquad \text{so} \qquad \log \left| \det \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}^\top} \right| = \sum_j s_j(\boldsymbol{z}_1, \boldsymbol{x}).$$

The matrix is still block-triangular. Its determinant is still the product of the diagonal. The messy top-right block, where all the complicated dependence on $\boldsymbol{z}_1$ and $\boldsymbol{x}$ lives, is still never needed. The condition *does* appear on the diagonal, inside $e^{s(\boldsymbol{z}_1, \boldsymbol{x})}$, but only as a number we evaluate, not as a new row or column. So the condition network can be as large as you like, even a frozen pretrained one, and the log-determinant stays exactly as cheap. The only thing a flow ever has to be invertible in is the data. (The likelihood direction uses the inverse map, whose log-determinant is simply $-\sum_j s_j(\boldsymbol{z}_1, \boldsymbol{x})$, which is the minus sign in Equation (6.22).)



$\det = e^{s(z_1,x)} \cdot 1$: read off the diagonal, whatever x is.

Figure 6.27. The Jacobian of a conditional coupling layer, with respect to z. The condition acts like a dial on the side: turning it changes the numbers in the cells, but the zero block stays zero, so the determinant is still the product of the diagonal.

Worked Example — Turning the dial: one 2-D conditional coupling

Take scalars z_0, z_1 and a scalar condition x, with

$$s(z_1, x) = 0.5\,(z_1 + x), \qquad t(z_1, x) = z_1 - x, \qquad y_0 = e^s\, z_0 + t, \quad y_1 = z_1.$$

Push $z = (0.4,\ 1.0)$ through it for two different thumbnails.
Condition $x = 1$. $s = 0.5(1+1) = 1$, $t = 1 - 1 = 0$. So $y_0 = e^1(0.4) + 0 = 1.087$ and $y_1 = 1.0$. Jacobian:

$$J = \begin{bmatrix} e^1 & \partial y_0/\partial z_1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2.718 & 1.544 \\ 0 & 1 \end{bmatrix}, \qquad \det J = 2.718, \quad \log\det J = 1 = s.$$

(The top-right entry is $0.5\,e^s z_0 + 1 = 1.544$. It is computed here only to show it exists. The determinant ignores it.)
Condition $x = 3$. $s = 0.5(1+3) = 2$, $t = 1 - 3 = -2$. So $y_0 = e^2(0.4) - 2 = 0.956$. Now $\det J = e^2 = 7.389$, so $\log\det J = 2 = s$. The same z lands somewhere else and space is stretched more, but the recipe for the determinant has not changed at all.
Invert (with $x = 1$). Read $z_1 = y_1 = 1.0$. Re-evaluate $s(1,1) = 1$ and $t(1,1) = 0$ forward. Then $z_0 = (1.087 - 0)\,e^{-1} = 0.4$. Exact recovery. x was used only as an input, never solved for.

In plain words — conditional flows

A normal flow learns to turn random noise into a realistic image. A conditional flow learns to turn random noise into a realistic image *that matches a given hint*, such as a blurry thumbnail. The hint is shown to every part of the machine, so the machine can decide where brightness goes, how uncertain each region is, and which details to add. Because the hint is only ever read, never reconstructed, it costs nothing extra in the

maths. And because the model gives a full distribution, not one best guess, it can hand you many different sharp images for the same thumbnail, each one crisp, instead of one blurry compromise.

Common Confusion

Conditional does not mean "trained on labeled data" here. In the super-resolution application the pair $(\boldsymbol{x}, \boldsymbol{y})$ is manufactured from the raw image itself: downsample or blur a sharp image $\boldsymbol{y}$ to obtain $\boldsymbol{x}$, and the dataset builds itself from unlabeled corpora. The condition is a *self-induced corruption*, in the same spirit as the augmentations of Chapter 8 and the masking of Chapters 5 and 11. Contrast Pix2Pix in Section 6.4.13, where the pairs must be collected with ground truth—that difference in provenance, not any architectural feature, is what decides whether a conditional generative model counts as self-supervised.

Slide 23 (Figure 6.25) shows the concrete payoff: super-resolution. Given a low-resolution image $\boldsymbol{x}$, a conditional flow models the full *distribution* of high-resolution images $\boldsymbol{y}$ consistent with it—unlike an L_2-trained regressor which averages all plausible sharp images and blurs. The side-by-side ImageNet-style grids (blurry left, sharp right) are the visual argument: likelihood-based conditional generation synthesizes detail instead of predicting the conditional mean. The 2020 SRFlow paper (Lugmayr et al., 2020) is the well-known descendant of this idea; conditional flows remain one of the strongest exact-likelihood approaches to image-to-image problems.

Key takeaway — conditional flows

(1) Squared-error regression returns the *mean* of all plausible answers, and the mean of many sharp images is a blurry one. (2) A conditional flow models the whole $p(\boldsymbol{y} \mid \boldsymbol{x})$ by letting the prior, every coupling layer and every split prior read $\boldsymbol{x}$. (3) The Jacobian is taken only with respect to $\boldsymbol{z}$, so $\boldsymbol{x}$ changes its values but not its triangular shape, and the log-determinant stays $\sum s$. (4) Pairs built by downsampling raw images make the whole setup self-supervised.

6.3.13 Where flows fail

Slide 24 closes the flow half with an honest summary.

Failure Modes

Normalizing flow trade-offs. (1) Dimension preservation. $\boldsymbol{x}$ and $\boldsymbol{z}$ have the same dimensionality, so memory and compute scale linearly with data size; the multi-scale factorization of Section 6.3.7 mitigates but does not remove this. **(2) Topology.** A bijection between $\mathbb{R}^d$ and $\mathbb{R}^d$ cannot change topology: it cannot tear a connected set apart, glue disjoint components together, or change intrinsic dimension. If p_{data} lives on a curved

manifold inside pixel space (it does), a Gaussian latent in $\mathbb{R}^d$ is topologically the *wrong* base—the model must fake it with nearly-singular Jacobians. Exact likelihood makes this a real cost, not a philosophical one. **(3) Likelihood–sample mismatch.** Real NVP's blurry samples (slide 17) had respectable likelihoods; many later models with excellent samples underperform flows in bits/dim. High likelihood is not sufficient for good samples, and good samples need not mean high likelihood. We will revisit whether likelihood is even the right objective once GANs give a competing answer. **(4) Architectural restriction.** Every layer must be invertible with a tractable Jacobian—this excludes most of the modern deep-learning toolbox and makes flows notoriously "not very easy to train," as the slide puts it.

The flows subsection ends with the design space mapped: exact likelihood, exact sampling, exact inference—available simultaneously, at a steep architectural price. Chapter 7 will show diffusion escaping through a different door (keep invertibility nowhere, use an SDE/ODE instead), and Chapter 8's pretext tasks escape through yet another (give up density estimation entirely).

6.4 Generative Adversarial Networks: From Likelihood to Game

6.4.1 Setup: a generator and a critic

The GAN half of the lecture follows the Stanford CS231n build sequence (slides 25–29, credited to Li et al.). The story starts with the same goal as flows—sample from $p_{\text{data}}(\boldsymbol{x})$—but abandons density evaluation entirely. Slide 25: fix a simple prior $p(\boldsymbol{z})$ (unit Gaussian); define a *generator network* $\boldsymbol{x} = G(\boldsymbol{z})$ mapping latents to data space; the distribution of $G(\boldsymbol{z})$ under $\boldsymbol{z} \sim p(\boldsymbol{z})$ is the generator distribution p_G. The training goal is $p_G = p_{\text{data}}$.

Connection: a generator *is* a reparameterised sampler. $\boldsymbol{x} = G(\boldsymbol{z})$ with $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is exactly the pattern of the VAE's reparameterisation trick (Section 4.6.2): fixed, parameter-free noise goes *in*, and a differentiable function of the parameters turns it into a sample. That is why the discriminator's gradient can flow back through the fake $G(\boldsymbol{z})$ into the generator's weights. It is also why GANs struggle with discrete outputs such as text: a sampled token has no gradient, so one needs the same workarounds (Gumbel-Softmax, REINFORCE). Flows (above) share the pattern too: sample $\boldsymbol{z}$, then apply a differentiable, invertible $\boldsymbol{x} = f(\boldsymbol{z})$.

Notice how completely the constraints of Section 6.3 evaporate in this one move. G is not required to be invertible, and in fact is typically not even a map between equal-dimensional spaces: a 100-dimensional $\boldsymbol{z}$ may expand into a $256^2 \times 3$-dimensional image. It may use pooling, stride, ReLU, dropout—the full unconstrained toolbox. Nor do we ever need $\det J$. What we lose, correspondingly, is everything Section 6.3 bought: there is no density to evaluate

(the push-forward p_G of an implicit map through a non-invertible network has no closed form), and there is no encoder—given a real image, finding its latent code is its own optimization problem. The entire architecture budget that flows spend on invertibility, GANs spend on *finding a training signal that works without a likelihood.*

Common Confusion

A flow and a GAN both "map noise to images." The mechanisms are opposites. The flow's map must be a bijection; it is *trained by* the change-of-variables likelihood; it encodes real data exactly. The GAN's map is unconstrained; it is *trained by* a second network's gradient; it cannot encode anything by itself. If a method chapter ever hands you a generative model, the first diagnostic question is: does $p(\boldsymbol{x})$ come out in closed form? If yes, expect restrictions; if no, expect adversaries.

How do we measure closeness to p_{data} without ever evaluating a density? Slide 26's answer: don't—train a second network to do the measuring. The *discriminator* D takes an image (real or generated) and outputs Fake/Real. Train D to classify correctly; train G to produce images that *fool* D. G never sees a density value; all signal is the gradient flowing back through D's classification boundary.

Historical Note

The forger and the detective. The standard pedagogical metaphor—in circulation since the original 2014 presentations of the idea, and alive in CS231n—casts G as an art forger and D as a detective. Initially both are incompetent: the forger paints blobs, the detective guesses at chance. The detective studies museum archives (the training set) and reports what gives fakes away ("real paintings have consistent lighting; yours do not"). The forger fixes the lighting, forcing the detective to find subtler flaws—canvas texture, brushstroke micro-rhythms. The equilibrium is reached when the forgery is indistinguishable and the detective is reduced to coin flips. The metaphor survives contact with the math uncommonly well: $D^* \equiv 1/2$ at equilibrium *is* the coin flip, and mode collapse (Section 6.4.4) is precisely the forger who, having mastered one painting the detective cannot call out, forges only that single painting forever.

Key Concept — GAN objective

The joint training objective (slide 27; Goodfellow et al., NeurIPS 2014) is the minimax value

$$\min_G \max_D \; \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\big[\log D(\boldsymbol{x})\big] \; + \; \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\big[\log\big(1 - D(G(\boldsymbol{z}))\big)\big], \tag{6.23}$$

where $D(\boldsymbol{x}) = p(\boldsymbol{x} \text{ is real})$ in $[0,1]$, $G : \mathbb{R}^{d_z} \to \mathbb{R}^{d_x}$ is the generator, and the outer optimization alternates gradients on D and G.

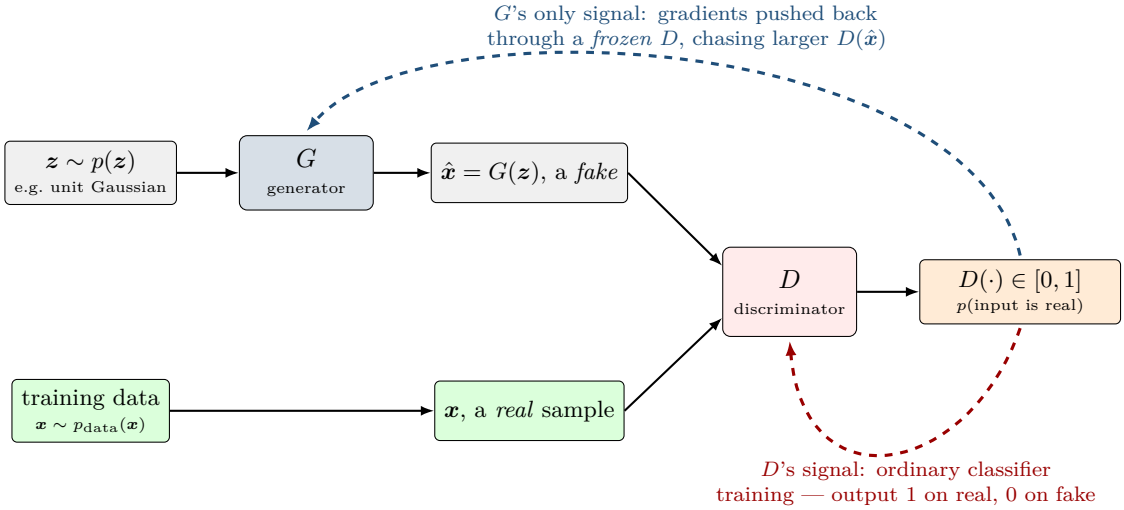


Figure 6.28. The GAN training loop. Solid arrows are the forward computation: samples flow from the prior and the dataset to a verdict. The two dashed curves are the entire training signal — D learns like an ordinary real/fake classifier, and G learns only from gradients that leak back through a momentarily frozen D. There is no likelihood anywhere in the picture, and no encoder.

How to read it: Trace the blue dashed curve. It starts not at the data but at D's output — which is why the generator never compares its samples to real ones directly, and why nothing in this diagram measures coverage.

Source: Redrawn after Lecture 04, slides 26–27 (Li et al., CS231n) (Goodfellow et al., 2014).

6.4.2 Reading the minimax game one player at a time

The lecture unfolds Equation (6.23) over five slides, and we compress that reveal into one clean derivation.

Fix G, optimize D (slides 27–28). The first expectation rewards D for outputting $\log D(\boldsymbol{x}) \to 0$, i.e. $D(\boldsymbol{x}) \to 1$ on real data. The second rewards $\log(1 - D(G(\boldsymbol{z}))) \to 0$, i.e. $D(G(\boldsymbol{z})) \to 0$ on fakes. So the optimal discriminator drives $D \to 1$ on the data manifold and $D \to 0$ off it—a real/impostor classifier trained against the current generator.

Fix D, optimize G (slide 29). Now the first term, having no G, is irrelevant to $\min_G$. The second term tells the generator to *minimize* $\log(1 - D(G(\boldsymbol{z})))$—that is, to push $D(G(\boldsymbol{z}))$ toward 1: make fakes look real to the current D. The two networks play a game with a moving target: D chases the boundary, G moves the boundary's input distribution.

What is the best possible discriminator? This short derivation is a standard exam question, and it shows what the GAN game is really optimising.

Fix the generator G, and ask which D maximises Equation (6.23). Write both averages as integrals over all possible images, where p_G is the distribution of the generator's output:

$$V(D,G) \;=\; \int \Big[p_{\text{data}}(\boldsymbol{x})\,\log D(\boldsymbol{x}) \;+\; p_G(\boldsymbol{x})\,\log\big(1-D(\boldsymbol{x})\big)\Big]\,d\boldsymbol{x}.$$

D can be chosen independently at each $\boldsymbol{x}$, so maximize the integrand pointwise. Set $a = p_{\text{data}}(\boldsymbol{x})$, $b = p_G(\boldsymbol{x})$, $y = D(\boldsymbol{x})$, and differentiate $f(y) = a\log y + b\log(1-y)$: $f'(y) = a/y - b/(1-y) = 0$ yields $a(1-y) = by$, i.e.

$$D^*(\boldsymbol{x}) \;=\; \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})}. \tag{6.24}$$

The optimal discriminator is a calibrated posterior: it outputs exactly the fraction of probability mass at $\boldsymbol{x}$ that is real. Its limiting behavior sanity-checks the formula. Where only fakes exist, $p_{\text{data}}(\boldsymbol{x}) = 0$ and $D^* = 0$; where only real data exists, $D^* = 1$; and at full convergence $p_G = p_{\text{data}}$, $D^* \equiv 1/2$ everywhere—the detective's coin flip.

Now substitute D^* back into the value function and ask what G is really minimizing. Splitting a factor of 2 out of each denominator gives

$$V(D^*,G) \;=\; D_{\text{KL}}\Big(p_{\text{data}} \,\Big\|\, \frac{p_{\text{data}}+p_G}{2}\Big) \;+\; D_{\text{KL}}\Big(p_G \,\Big\|\, \frac{p_{\text{data}}+p_G}{2}\Big) \;-\; 2\log 2,$$

which is precisely

$$V(D^*,G) \;=\; 2\,\text{JS}\big(p_{\text{data}} \,\|\, p_G\big) - 2\log 2, \tag{6.25}$$

by the definition of the *Jensen–Shannon divergence* (JS), a measure of how different two distributions are.

What this means:

- JS is never negative, and is zero only when the two distributions are identical.
- So the best the generator can do is make $p_G = p_{\text{data}}$ exactly. The value of the game is then $V = -2\log 2 \approx -1.386$.
- At that point the best discriminator outputs $D^* = 1/2$ for every input — it genuinely cannot tell real from fake, and guesses.

Neither player can improve by changing only its own strategy, which is what "equilibrium" means. (The slides skip this derivation; Part III uses it when asking what quantity generative models really optimise.)

6.4.3 Saturating vs. non-saturating loss: a look at the gradients

Equation (6.23) is the form on the slides and in the 2014 paper, but it is *not* the form anyone trains. The reason is purely gradient-level.

Why the textbook generator loss barely trains. Early in training the generator is bad, so the discriminator easily spots its fakes: $D(G(\boldsymbol{z})) \approx 0$. Look at what the original loss $\log(1 - D)$ does there. Call $a = D(G(\boldsymbol{z}))$:

	loss	**slope at** $a = 0.01$	**slope at** $a = 0.5$
original ("saturating")	$\log(1 - a)$	≈ 1.0	2.0
fix ("non-saturating")	$-\log a$	100	2.0

When the fakes are terrible ($a = 0.01$), the original loss gives a gentle slope of about 1 — and in practice even less, because the discriminator's own sigmoid is also flat there. The generator is being told almost nothing at exactly the moment it most needs guidance. The fixed loss gives a slope of 100: *the worse the generator is, the louder the signal.* Once the fakes are decent ($a = 0.5$) the two agree. So the fix is to train the generator on

$$\max_G \; \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\big[\log D(G(\boldsymbol{z}))\big], \tag{6.26}$$

In words: instead of "make the discriminator *less* sure my fakes are fake", ask for "make the discriminator *more* sure my fakes are real". Both goals are reached at the same point, so nothing about the destination changes. What changes is the slope along the way: steep when the generator is bad, which is when steepness helps. Every practical GAN uses Equation (6.26) for the generator and leaves the discriminator's loss as it was.

Common Confusion

The two "GAN losses" are not two variants of the algorithm. The saturating objective $\min_G \mathbb{E}[\log(1 - D(G(\boldsymbol{z})))]$ is what the paper's minimax game literally says. The non-saturating objective $\max_G \mathbb{E}[\log D(G(\boldsymbol{z}))]$ is the same game re-parameterized so gradient descent behaves. You will see both written in papers side by side; they correspond to one equilibrium, two optimization dynamics. The theory section proves things about the first; the engineering section trains the second.

6.4.4 Training dynamics and failure modes

GAN training is a two-player dynamical system, and the failure modes are game-theoretic rather than optimization-noise.

Failure Modes

Mode collapse. Nothing in Equation (6.23) forces p_G to cover all of p_{data}. Look again at the generator's fixed-D objective, $\min_G \mathbb{E}_{\boldsymbol{z}}[\log(1 - D(G(\boldsymbol{z})))]$: it penalizes fakes that get *caught*, and only those. It contains no term measuring coverage or diversity—no reward for hitting *all* the modes, only for hitting *a* mode D currently believes. So if the generator finds a single face $\boldsymbol{x}^*$ that reliably passes the current discriminator, the cost-minimizing program is degenerate: set $G(\boldsymbol{z}) = \boldsymbol{x}^*$ for every $\boldsymbol{z}$. Train a GAN on a dataset of cats and

dogs and the symptom is unmistakable: day one produces both species, day ten produces breathtaking cats, no dogs. Fixing D afterward exposes the collapse (all outputs rate identically); but if the updates alternate quickly, G can instead *rotate* among modes—this week cats, next week dogs—always one step ahead of D's moving boundary, never covering the support at any instant. The original paper's Figure 1 shows the healthy version of this interaction schematically: (a) near convergence, p_G (green) is close to p_{data} (black) and D (blue, dashed) is only partly accurate; (b) D is trained towards D^*; (c) G moves towards the regions D calls real; (d) after enough rounds $p_G = p_{\text{data}}$ and D is flat at 1/2. Pathological training is what happens when (b)–(c) cycle without ever reaching (d). Standard repairs include minibatch discrimination (let D score *batches*, so a batch of near-duplicates is itself damning evidence) (Salimans et al., 2016) and replacing the JS objective with the Wasserstein distance (WGAN), which keeps gradients informative even when real and fake supports are disjoint (Arjovsky et al., 2017). Section 6.4.5 develops both. Do not confuse collapse with its everyday cousin: an *overfit* generator would emit many distinct memorized training images—diverse but unoriginal; a *collapsed* generator emits the same image again and again—original-looking but non-diverse. The symptoms are opposite in the diversity axis, and the chapter's exercises probe the distinction. (Also keep the generative-family accounting straight: VAEs have their own collapse—*posterior collapse*, where the decoder learns to ignore $\boldsymbol{z}$ entirely—which is a different mechanism, different symptoms, and different remedies than mode collapse; Chapter 4 covers it.)

Discriminator overconfidence. A too-strong D becomes confidently "fake" over a whole region around the generator's samples, so its output is nearly flat there and the gradient with respect to the generated image — the only thing G learns from — is tiny or points nowhere useful. The non-saturating loss fixes the vanishing factor contributed by the *loss*, but it cannot fix a *discriminator* that is flat around the fakes, so training can still stall. **No evaluation curve.** Slide 40 closes with GAN pros (green) and cons (red): simple formulation, very good image quality, *versus* no loss curve to look at, unstable training, hard to scale to big models and big data. The "no loss curve" point is structural: there's no single scalar that tracks progress, because the loss measures the *game*, not data fit. Practical GAN training is diagnosed by looking at samples, not by reading training curves—a habit modern practice is only now growing out of.

Learning note from study sessions — Precision on mode collapse

A useful sharpening from study sessions on this deck: mode collapse is a statement about the *objective*, not about generator capacity. Against a *fixed* D, the generator's objective is minimized by $G(\boldsymbol{z}) = \boldsymbol{x}^*$ whenever $\boldsymbol{x}^*$ fools D. The global optimum of the full game is still $p_G = p_{\text{data}}$, but alternating gradient steps only ever see the current D, and more generator capacity only makes the collapse *easier* to reach. This is why every anti-collapse trick modifies either what D sees (minibatch discrimination) or what the global game measures (Wasserstein distance), rather than the generator's size.

6.4.5 Why GAN training is hard, and what fixes each problem

The failure modes above are symptoms. This section takes them one at a time and asks the same four questions of each: *what do you see*, *what is actually going wrong*, *what fixes it*, and *what evidence says the fix works*. Table 6.2 is the map; the rest of the section walks through it.

Table 6.2. GAN training problems and the standard fixes. Every fix in the right column is developed below, with its source.

Problem	What you see	Root cause	Standard fixes
Vanishing gradients	G stops improving early; D's accuracy near 100%	real and fake barely overlap, so JS is stuck at $\log 2$ and gives no direction	non-saturating loss; Wasserstein loss (WGAN, WGAN-GP)
Oscillation, no convergence	samples keep changing without getting better; losses cycle	the game's gradient field *rotates* around the equilibrium instead of pointing at it	R1 gradient penalty; spectral normalization; two learning rates (TTUR)
Mode collapse	many $\boldsymbol{z}$ give near-identical images	nothing in the loss rewards diversity	minibatch discrimination; minibatch standard deviation; larger batches
Discriminator overfitting	works on big datasets, diverges on small ones	D memorizes the training images	adaptive augmentation (ADA), DiffAugment
Collapse at scale	a long, healthy run suddenly falls apart	weight spectra grow until training is unstable	early stopping; orthogonal regularization; spectral normalization
No progress signal	the loss curve says nothing about quality	the loss measures the *game*, not the fit	FID, Inception Score, precision/recall

Problem 1: a perfect discriminator gives no gradient

What you see. Early in training the discriminator wins easily, its accuracy climbs towards 100%, and the generator stops improving.

What is going wrong. Section 6.4.2 showed that with a perfect discriminator, the generator is minimizing the Jensen–Shannon divergence. JS has a nasty property: when two distributions do not overlap at all, it equals $\log 2$ *no matter how far apart they are*. A generator whose faces are one pixel off and a generator emitting static get exactly the same score, so there is no slope to follow. And not overlapping is the normal case: real images occupy a thin, low-dimensional sheet inside pixel space, the generator's output is another thin sheet, and two thin sheets in a huge space almost never overlap (Arjovsky et al., 2017).

Worked Example — the "parallel lines" example of Arjovsky et al. (2017)

Let the real data be the vertical line segment $\{(0, z) : z \in [0, 1]\}$ and let the generator produce the parallel segment $\{(\theta, z)\}$, with one parameter θ. The ideal answer is $\theta = 0$. Now compare two ways of measuring the gap:

$$\mathrm{JS}(p_{\text{data}}, p_\theta) = \begin{cases} \log 2 & \theta \neq 0 \\ 0 & \theta = 0 \end{cases} \qquad W(p_{\text{data}}, p_\theta) = |\theta|.$$

At $\theta = 5$ and at $\theta = 0.01$, JS says the same thing: "wrong". Its derivative is zero everywhere except at the answer. The Wasserstein distance W says "you are 5 away" and "you are 0.01 away", and its slope always points home. Figure 6.29 plots both.

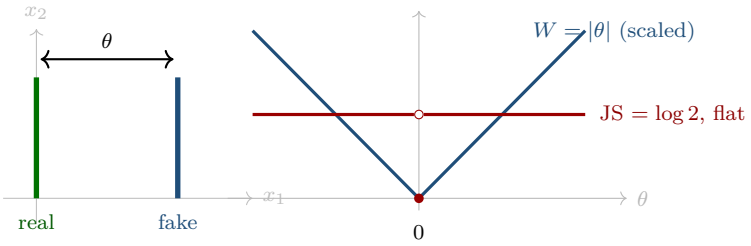


Figure 6.29. Why JS gives no training signal when the supports do not overlap. Along the whole red line the slope is zero, so a generator anywhere on it gets no hint which way to move. The Wasserstein distance (blue) grows with the distance to the answer and has a useful slope everywhere.

How to read it: Pick any $\theta \neq 0$ and ask which way is downhill. On the blue curve the answer is always "towards zero". On the red one there is no downhill at all.

Fix 1: the non-saturating loss. Section 6.4.3 already gave the cheapest fix: train G on $-\log D(G(\boldsymbol{z}))$, which is steep exactly when D is winning. It helps a great deal in practice, but it does not change the underlying divergence.

Fix 2: change the distance (WGAN). The Wasserstein GAN (Arjovsky et al., 2017) replaces the classifier with a *critic* C that outputs an unbounded score, not a probability, and trains

$$\min_G \max_{C\,:\,\|C\|_{\text{Lip}} \leq 1} \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}}[C(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z}}[C(G(\boldsymbol{z}))]. \tag{6.27}$$

The constraint $\|C\|_{\text{Lip}} \leq 1$ ("1-Lipschitz") says the critic's score may not change faster than the input does; under it, the maximum equals the Wasserstein distance. The catch is enforcing it. The original paper clips every critic weight into a small box such as $[-0.01, 0.01]$ and itself calls this "a clearly terrible way to enforce a Lipschitz constraint".

Fix 2b: enforce the constraint with a penalty (WGAN-GP). Gulrajani et al. (2017) drop clipping and instead penalize the critic's gradient norm for straying from 1:

$$\mathcal{L}_C = \mathbb{E}[C(G(\boldsymbol{z}))] - \mathbb{E}[C(\boldsymbol{x})] + \lambda\, \mathbb{E}_{\hat{\boldsymbol{x}}}\big(\|\nabla_{\hat{\boldsymbol{x}}} C(\hat{\boldsymbol{x}})\|_2 - 1\big)^2, \tag{6.28}$$

where each $\hat{\boldsymbol{x}}$ is a random point on the straight line between a real and a fake sample, and $\lambda = 10$ in all of the paper's experiments. Checking the gradient everywhere is impossible; checking it along those lines, where the optimal critic should have slope exactly 1, turns out to be enough.

Problem 2: the game circles instead of converging

What you see. Samples keep changing from checkpoint to checkpoint without getting better. Both losses oscillate.

What is going wrong. Ordinary training walks *downhill* on one loss. A GAN is two players pulling on the same quantity in opposite directions, and the combined update can point *around* the equilibrium rather than towards it. The smallest example that shows this is the *Dirac-GAN* of Mescheder et al. (2018).

Worked Example — the Dirac-GAN: one number against one number

The real data is a single point at $x = 0$. The generator is a single point at $x = \theta$ (one parameter). The discriminator is a line through the origin, $D_\psi(x) = \psi x$ (one parameter). The equilibrium is obviously $\theta = 0$ (fake sits on real) and $\psi = 0$ (nothing to tell apart).

With the standard GAN loss, near that equilibrium the two gradient steps are approximately

$$\theta \;\leftarrow\; \theta - \tfrac{h}{2}\,\psi, \qquad\qquad \psi \;\leftarrow\; \psi + \tfrac{h}{2}\,\theta.$$

Read them as a story. If the fake is to the right ($\theta > 0$), the discriminator tilts to score the right side as fake (ψ grows). A tilted discriminator pushes the fake left (θ shrinks), past zero. Now the fake is on the left, so the discriminator tilts the other way, and so on. Each player reacts to where the other *was*. That is the recipe for a rotation, and it is exactly what $(\theta, \psi) \mapsto (\theta - \frac{h}{2}\psi,\ \psi + \frac{h}{2}\theta)$ is: a small rotation, slightly enlarged, since each step multiplies the distance from the equilibrium by $\sqrt{1 + h^2/4} > 1$.

The fix, in the same example. The R1 penalty (below) charges the discriminator $\frac{\gamma}{2}\|\nabla_x D_\psi(x)\|^2$ on real data. Here $\nabla_x D_\psi = \psi$, so the penalty is $\frac{\gamma}{2}\psi^2$ and the discriminator's step gains a term $-h\gamma\psi$: friction on ψ. The rotation becomes a spiral *inwards*.

Fix 1: penalize the discriminator's gradient on real data (R1). Mescheder et al. (2018) add to the discriminator's loss

$$R_1(\psi) = \frac{\gamma}{2}\,\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\big\|\nabla_{\boldsymbol{x}} D_\psi(\boldsymbol{x})\big\|^2. \tag{6.29}$$

In words: on real images, the discriminator's output should be flat. Near the equilibrium this is exactly the friction of the worked example. Unlike WGAN-GP it penalizes towards zero slope, not unit slope, and only on real data. R1 is what StyleGAN trains with on FFHQ, and what StyleGAN2 uses throughout.

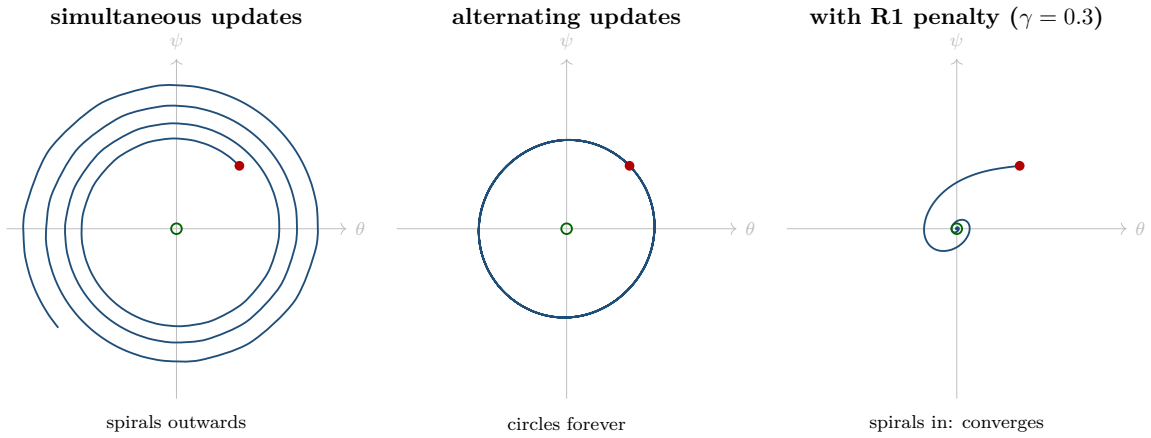


Figure 6.30. The Dirac-GAN, simulated for this book: 700 steps of size $h = 0.1$ from the red starting point $(\theta, \psi) = (1, 1)$; the green circle is the equilibrium. Plain simultaneous gradient steps drift outwards (radius $1.41 \to 2.45$), alternating steps circle at a fixed radius, and adding the R1 penalty ($\gamma = 0.3$) turns the same rotation into an inward spiral that reaches the equilibrium. The qualitative picture matches Figure 2 of Mescheder et al. (2018).

How to read it: Each panel is the path of the pair *(generator parameter, discriminator parameter) over training. A healthy optimizer would head straight for the green circle. Only the right panel gets there.*

Fix 2: limit how fast the discriminator can change (spectral normalization). Miyato et al. (2018) divide every weight matrix of D by its largest singular value, $\bar{W} = W/\sigma(W)$, so that no layer can stretch its input by more than a factor of 1 and the whole discriminator is 1-Lipschitz. $\sigma(W)$ is estimated cheaply with one step of power iteration per training step. Its one hyperparameter needs little tuning, and BigGAN uses it.

Fix 3: give the two players different step sizes (TTUR). Heusel et al. (2017) prove that updating D and G with separate learning rates converges to a local equilibrium under mild assumptions. The practical upshot is simple: tune the two learning rates separately rather than tying them.

Problem 3: mode collapse

Section 6.4.4 described the symptom and the cause: the loss rewards fooling D and never rewards variety. Every fix therefore lets D *see* variety, by looking at several samples at once.

- **Minibatch discrimination** (Salimans et al., 2016). A layer near the end of D compares each sample's features with those of the other samples in the minibatch and passes the result on. A batch of near-duplicates is now easy to catch.
- **Minibatch standard deviation** (Karras et al., 2018). A parameter-free simplification: compute the standard deviation of every feature over the minibatch, average all of them into *one number*, and append it to D as an extra constant feature map. Real batches have a typical spread; a collapsed generator's batches have almost none.

- **Feature matching** (Salimans et al., 2016). Train G to match the *average* intermediate features of real data in D, instead of maximizing D's output directly, so there is no single image to over-optimize.
- **Larger batches.** BigGAN found that increasing the batch size eight-fold improved Inception Score by 46%, and its authors conjecture that each batch then covers more modes (Brock et al., 2019). (That explanation is their conjecture, not a measured finding.)

Problem 4: the discriminator memorizes the training set

What you see. Training that works on 70,000 faces diverges on 2,000. Early samples look fine; then quality falls apart.

What is going wrong. With few images, D simply memorizes them. BigGAN measured this directly on ImageNet: its discriminators classified training images correctly more than 98% of the time, but held-out validation images only 50–55% of the time, which is chance (Brock et al., 2019). A memorizing D gives the generator feedback about those particular images, not about images in general.

The obvious fix fails. Augmenting the real images (flips, crops, colour changes) stops memorization, but the generator then learns to produce augmented-looking images, since those now count as "real". The paper calls this augmentations *leaking* into the generator.

Fix: augment everything D sees, and adapt the strength (ADA). Karras et al. (2020a) apply the same random augmentations to real *and* generated images before D sees them, choosing only augmentations that are provably non-leaking when applied with probability $p < 1$. They then adjust p automatically during training using an overfitting signal, $r_t = \mathbb{E}[\text{sign}(D_{\text{train}})]$: the fraction by which D is confident that training images are real. When r_t rises above the target 0.6, p goes up; when it falls, p goes down. The paper reports that good results become possible with only a few thousand training images, often matching StyleGAN2 with an order of magnitude fewer. DiffAugment (Zhao et al., 2020) reaches a similar conclusion with fixed, differentiable augmentations applied to both real and fake samples.

Problem 5: collapse at scale

Big models add a failure of their own. BigGAN's larger runs reached their best scores faster and then collapsed completely; the authors traced the top singular values of the generator's weights growing through training and exploding at the moment of collapse. Their paper's practical answer is to stop early, keep the checkpoint just before collapse, and regularize; it states plainly that full stability came only at a large cost in performance (Brock et al., 2019). This is a useful reminder that "GAN stability" was never fully solved, only managed.

Problem 6: no loss curve to read

Because the GAN loss scores the game, the field built separate yardsticks that score the samples.

- **Inception Score (IS)** (Salimans et al., 2016). Classify each generated image with a pretrained Inception network. A good sample gives a confident prediction $p(y \mid \boldsymbol{x})$; a good *set* of samples spreads over many classes, so the average prediction $p(y)$ is broad. $\mathrm{IS} = \exp(\mathbb{E}_{\boldsymbol{x}}\, D_{\mathrm{KL}}(p(y \mid \boldsymbol{x}) \,\|\, p(y)))$. Higher is better. It never looks at real images.
- **Fréchet Inception Distance (FID)** (Heusel et al., 2017). Take Inception features of real and generated images, fit a Gaussian to each set (means $\boldsymbol{\mu}_r, \boldsymbol{\mu}_g$, covariances Σ_r, Σ_g), and measure the distance between the Gaussians:

$$\mathrm{FID} = \|\boldsymbol{\mu}_r - \boldsymbol{\mu}_g\|^2 + \mathrm{Tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}). \tag{6.30}$$

  Lower is better, and 0 means the two feature distributions match. The first term catches wrong content, the second wrong variety, so FID penalizes mode collapse where IS does not. FID is the number reported in every table in the rest of this section.
- **Precision and recall.** FID mixes "are the samples realistic?" with "do they cover the data?". Precision and recall split the two. StyleGAN2 reports both, and uses them to argue that one of its changes traded some precision for recall.

Common Confusion

FID numbers are only comparable under the same protocol. FID depends on the dataset, the resolution, the number of samples and the feature extractor. A FID of 4.40 on FFHQ at 1024^2 and a FID of 7.4 on ImageNet at 128^2 are not two points on one scale. Compare FIDs only within one paper's table, or across papers that state the same protocol.

Key takeaway — what a modern GAN actually trains with

StyleGAN2's appendix lists its full recipe (Karras et al., 2020b): the non-saturating logistic loss with the R1 penalty; an exponential moving average of the generator's weights for the model you actually sample from; the minibatch standard deviation layer in D; equalized learning rate; a 100× lower learning rate for the mapping network; and style mixing regularization. Almost every item on that list is a fix from this section.

6.4.6 Before StyleGAN: DCGAN and progressive growing

StyleGAN is the end of a short line of papers, each fixing what stopped the previous one from scaling. Two of them explain most of StyleGAN's design.

Model at a glance: DCGAN	Radford, Metz & Chintala, 2016
In one line	The first recipe that made convolutional GANs train reliably, written as five architecture rules (Radford et al., 2016).
The rules	Replace pooling with strided convolutions (in D) and fractional-strided ("transposed") convolutions (in G). Use batch normalization in both networks. Remove fully connected hidden layers. Use ReLU in G except a Tanh output. Use LeakyReLU in D.
Why it matters	Before DCGAN, convolutional GANs were notoriously hard to train. These rules became the default template for years, and every later generator in this section is still a stack of upsampling convolutions.
SSL connection	The paper's title is *Unsupervised Representation Learning with DCGANs*: it used the trained discriminator's features for downstream classification, an early instance of this book's theme.

Model at a glance: Progressive GAN (ProGAN)	Karras, Aila, Laine & Lehtinen, 2018
In one line	Start both networks at 4×4, and add layers as training proceeds until they reach 1024×1024 (Karras et al., 2018).
Problem it solves	Learning every scale at once is hard. At low resolution there are fewer details and fewer modes, so the game is easier; growing lets each new layer build on a stable coarse model.
Key detail: fade-in	A new resolution is not switched on abruptly. Its output is blended with an upsampled copy of the old output, with a weight α that goes from 0 to 1 (Figure 6.31).
Other contributions	Minibatch standard deviation (Problem 3 above); equalized learning rate; pixelwise feature normalization in G; and the CelebA-HQ dataset at 1024^2.
Result	Megapixel faces, and a then-record Inception Score of 8.80 on unsupervised CIFAR-10.
What came later	StyleGAN kept ProGAN as its baseline (Table 6.3, row A). StyleGAN2 later *removed* progressive growing, because it causes "phase" artifacts (Section 6.4.9).

6.4.7 StyleGAN: a generator that composes styles

Moving from the vanilla GAN to StyleGAN (slides 30–35) is not a change of objective—it is a wholesale redesign of the generator. *A Style-Based Generator Architecture for Generative Adversarial Networks* by Karras, Laine, and Aila (Karras et al., 2019) borrows from style transfer to give G internal knobs with semantic meaning.

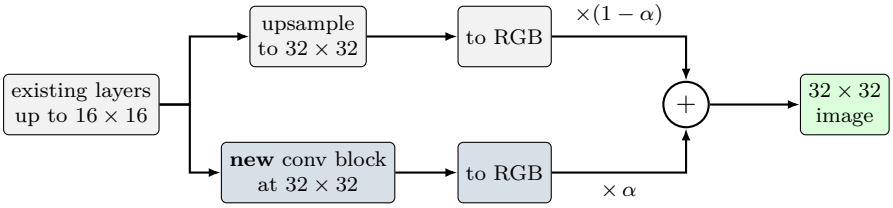


Figure 6.31. Progressive growing's fade-in, for the step from 16×16 to 32×32. The new block starts with no influence ($\alpha = 0$), so it cannot shock the already-trained layers, and takes over smoothly. The discriminator grows in mirror image.

How to read it: The upper path is the old network's answer, simply enlarged. The lower path is the new, still-untrained block. The output is a blend of the two, and the blend weight is the only thing that changes during a transition.

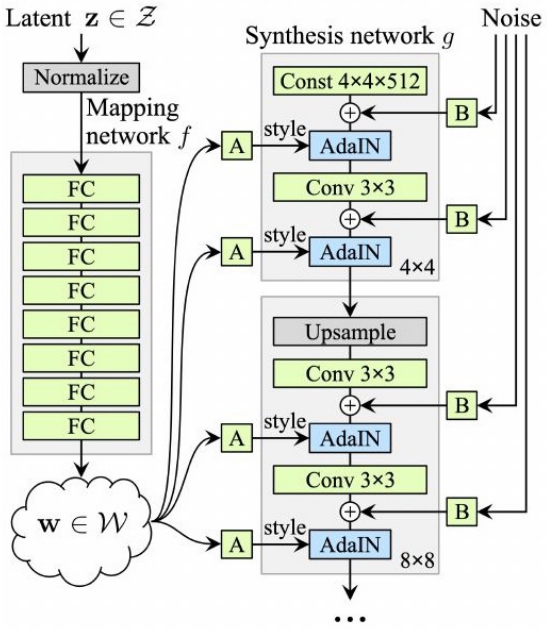


Figure 6.32. The StyleGAN generator. Three things are unusual and all three matter. The latent $\boldsymbol{z}$ does not enter the image path at all — it is first pushed through a *mapping network* f of eight fully connected layers into a second latent $\boldsymbol{w}$. The image path starts from a learned *constant* tensor, not from noise. And $\boldsymbol{w}$ is injected *repeatedly*, once per resolution, through AdaIN — which is what makes "style" a per-scale quantity rather than a single global one.

How to read it: Follow $\boldsymbol{w}$ out of the cloud: it fans out to every `A` *box, and there is one* `A` *per resolution. That fan-out is the entire idea — the same latent gets a separate say at every scale. The* `B` *boxes on the right are per-pixel noise, deliberately kept on a channel of its own so $\boldsymbol{w}$ never has to encode randomness.*

Source: Lecture 04, slide 31 (Karras et al., 2019).

Paper Analysis: StyleGAN (Karras et al., 2019)

Problem. A vanilla GAN generator maps $\boldsymbol{z} \in Z$ to pixels monolithically; there is no reason any direction in Z corresponds to anything human-interpretable. **Insight.** Learned latent structure can be *shaped* by architecture. Inject the latent at many scales with a normalization layer that makes each injection meaningfully "style-like," and let stochastic variation enter through a separate channel. **Method.** Four pieces, all visible on slide 31. (i) A *mapping network* f: eight fully connected layers mapping the initial Gaussian latent $\boldsymbol{z} \in Z$ to an intermediate latent $\boldsymbol{w} \in W$. Z must mirror the training density (it is Gaussian by construction); W is free to become entangled no further than the data demands. (ii) A *synthesis network* g starting not from $\boldsymbol{z}$ but from a learned $4 \times 4 \times 512$ *constant* tensor. (iii) A per-layer learned affine A turning $\boldsymbol{w}$ into *style* $(y_{s,i}, y_{b,i})$, injected after each conv by *AdaIN*: each feature map is normalized to zero mean and unit variance, then rescaled and shifted by the style. (iv) Per-layer, per-channel random *noise inputs* scaled by a learned B, giving the network a source of high-frequency stochasticity it does not have to smuggle through $\boldsymbol{z}$. **Results.** The style-mixing experiment (slide 32, from the paper's Figure 3) is the cleanest demonstration: take two source latents and swap the style code $\boldsymbol{w}$ at selected layers. Using B's *coarse* styles (resolutions 4^2–8^2) copies pose, general hair style, face shape, eyeglasses; B's *middle* styles control facial features' finer shape, hair detail spread; B's *fine* styles (64^2–1024^2) carry color scheme and microstructure while leaving identity untouched. The regularization of style mixing during training pushes the network to treat these scales independently. **Later work.** StyleGAN 2 removed the characteristic "droplet" artifacts by redesigning normalization; StyleGAN 3 made the generator equivariant to translation and rotation. The core idea—disentangle through architecture, not objective—persists. **What to remember.** W freed from Z's density-matching restriction is claimed to be less entangled; the authors support this with two new metrics (perceptual path length, linear separability), both worth looking up in the paper when you need to argue about "disentanglement" formally.

What "disentangled" means. Slide 33 names the goal behind StyleGAN's design. Imagine the character creator in a video game such as *The Sims*: one slider changes hair colour, another changes age, another changes the smile — and moving one slider does *not* secretly change the others. A latent space is *disentangled* when it behaves like that: each direction in the latent space (for StyleGAN, in W) controls one understandable property of the image.

Style mixing (Figure 6.34) is evidence that StyleGAN gets part of the way there: the coarse-resolution styles act like the "pose and face shape" sliders, the fine-resolution styles like the "colour and texture" sliders.

What the noise inputs are for. Slide 34 shows what happens if you remove StyleGAN's extra noise inputs: faces go smooth and painting-like. Details such as the exact placement of individual hairs, freckles and skin pores are *random* — they do not describe who the person

is. Giving the network a separate source of randomness for them means the style code does not have to waste capacity encoding them, and can stay reserved for meaningful properties.

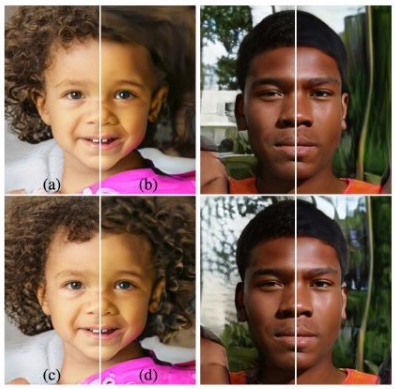


Figure 6.33. What StyleGAN's noise inputs are actually for (the paper's Figure 5). Each face is split vertically into a half-and-half comparison. Top row: (a) on the left halves, noise at all layers — the default — against (b) on the right halves, no noise at all: the hair loses its curls and the face degenerates to a featureless, "painterly" cartoon, because the network has no source of randomness left with which to invent curls, pores and stray hairs. Bottom row: the middle cases — (c) noise only at the fine layers keeps identity and hair mass correct with realistic high-frequency texture on top, while (d) noise only at the coarse layers randomises the overall head-and-hair shape but leaves a strangely smooth surface. The decomposition is not designed by hand — it *emerges* from giving each scale its own noise channel.

How to read it: Run a finger down the white seam of any face: the two halves differ only in which layers *received noise, so every visible difference can be charged to a specific scale of detail.*

Source: Lecture 04, slide 34 (Karras et al., 2019).

6.4.8 StyleGAN, one component at a time

The paper box above lists StyleGAN's pieces. This section takes each piece and asks: *what does it do, why is it there, what does it look like on a small example, and what did the paper measure when it was added?* The measurements come from the paper's own ablation study (Table 6.3), which starts from Progressive GAN and adds one change per row.

Table 6.3. StyleGAN's ablation, from Table 1 of Karras et al. (2019): FID (lower is better) after each change, on two face datasets. Each row keeps everything above it.

	Configuration	CelebA-HQ	FFHQ
A	Baseline: Progressive GAN	7.79	8.04
B	+ tuning (bilinear up/downsampling, longer training, hyperparameters)	6.11	5.25
C	+ mapping network and styles (AdaIN)	5.34	4.85
D	+ remove the traditional input (start from a learned constant)	5.07	4.88
E	+ noise inputs	5.06	4.42
F	+ mixing regularization	5.17	4.40

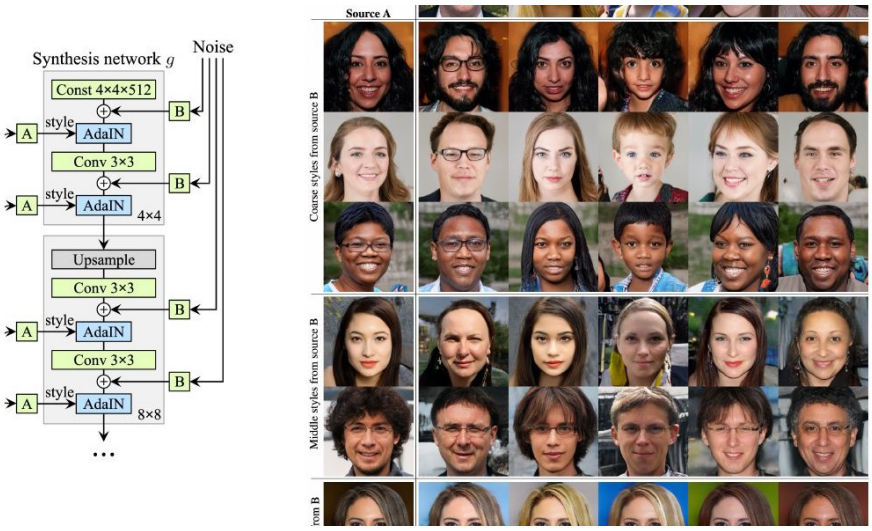


Figure 6.34. Style mixing: what the per-scale injection actually buys. Each grid takes the styles for *some* resolutions from source B and the rest from source A. Taking B's styles at coarse resolutions transfers pose, face shape and glasses; at middle resolutions it transfers facial features and hair detail; at fine resolutions only the colour scheme changes. Nobody labelled "pose" or "hair colour" anywhere in training — the separation is a by-product of injecting the same latent at many scales.

How to read it: Read one grid row: the same source-A identity, re-rendered with progressively more of source B's style. Then compare across the three grids to see which resolution controls which attribute.

Source: Lecture 04, slide 32 (Karras et al., 2019).

Read the table honestly before reading the components. The largest single gain on FFHQ (A to B) is *tuning*, not a new idea. The style machinery (C) gives a real further gain. The constant input (D) helps on CelebA-HQ and is neutral on FFHQ. Mixing regularization (F) barely moves FID at all, and slightly hurts it on CelebA-HQ: its purpose, as we will see, is something FID does not measure.

Component 1: the mapping network $f : \boldsymbol{z} \mapsto \boldsymbol{w}$

What it does. An 8-layer fully connected network turns the 512-dimensional Gaussian latent $\boldsymbol{z}$ into a second 512-dimensional latent $\boldsymbol{w}$. The image-making part of the generator never sees $\boldsymbol{z}$.

Why it is there. Suppose the training faces vary along two factors, hair length and masculinity, and one combination (long-haired men, say) is rare or absent. $\boldsymbol{z}$ is drawn from a Gaussian, which has no holes: every region of Z gets sampled. So the generator is forced to *bend* Z so that no region lands on the missing combination, and a bent space means the factors can no longer be straight directions. $\boldsymbol{w}$ is not tied to any fixed distribution, so the mapping network can "unbend" it, leaving factors closer to straight lines (Figure 6.35). This is the argument of the paper's Figure 6.

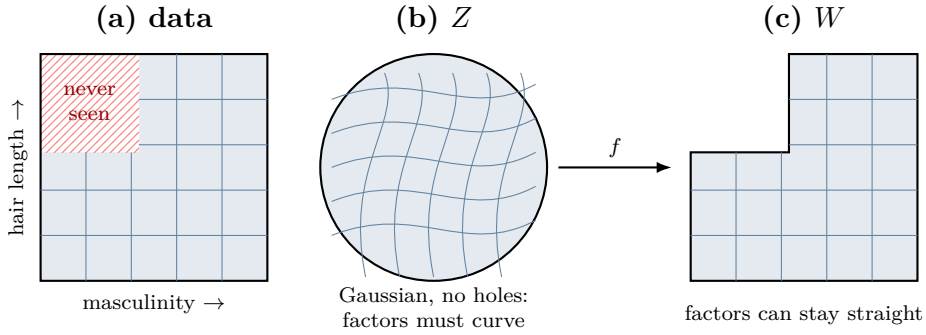


Figure 6.35. Why StyleGAN puts a mapping network in front of the generator (after Figure 6 of Karras et al. (2019)). (a) Two factors of variation, with one combination missing from the data. (b) A Gaussian Z has no holes, so the map from Z to images must curve its axes around the missing region. (c) The learned space W has no fixed shape, so it can take the data's shape and keep the factors straight.

How to read it: Follow one grid line through each panel. In (a) and (c) it is straight: moving along it changes one factor only. In (b) it bends, so moving along a straight line in Z changes both factors at once.

Evidence. The paper tests this with Table 4 (FFHQ). A style-based generator with no mapping network (latent used directly) scores FID 5.06 and *linear separability* 9.88; with an 8-layer mapping network, FID 4.40 and separability 3.79. Separability measures how well a linear boundary in the latent space can split images by attributes such as gender; lower means the factors are closer to straight directions. The same table shows adding a mapping network to a *traditional* generator also helps FID ($5.25 \to 4.87$), and that deeper mapping networks generally did better than shallow ones.

Component 2: the learned constant input

What it does. The image path starts from a fixed, learned $4 \times 4 \times 512$ tensor. It is the same for every image.

Why it is there. This was a discovery, not a plan. After adding the mapping network and styles (row C), the authors noticed that feeding the latent into the first layer no longer helped, so they removed it (row D). Every difference between two StyleGAN images therefore comes *only* through the styles and the noise. That is a strong architectural statement: the latent is not "what to draw" but "how to modulate a drawing that always starts the same".

Evidence. Row D: CelebA-HQ improves $5.34 \to 5.07$; FFHQ is essentially unchanged ($4.85 \to 4.88$). The honest summary is "simpler at no cost", not "better".

Component 3: styles through AdaIN

What it does. After each convolution, a learned affine layer ("A" in Figure 6.32) turns $\boldsymbol{w}$ into a style $\boldsymbol{y} = (\boldsymbol{y}_s, \boldsymbol{y}_b)$: one scale and one bias per feature map. Then adaptive instance normalization applies it:

$$\mathrm{AdaIN}(\boldsymbol{x}_i, \boldsymbol{y}) = y_{s,i} \frac{\boldsymbol{x}_i - \mu(\boldsymbol{x}_i)}{\sigma(\boldsymbol{x}_i)} + y_{b,i}, \tag{6.31}$$

where $\boldsymbol{x}_i$ is the i-th feature map and μ, σ are its mean and standard deviation over spatial positions. So $\boldsymbol{y}$ has twice as many numbers as the layer has feature maps. The synthesis network has 18 such layers, two per resolution from 4^2 to 1024^2.

Why it is there. Each AdaIN first *erases* a feature map's overall level and spread, then *imposes* new ones chosen by the style. So every layer's style has full control over its own feature statistics, and none of the previous style survives into the next layer's statistics. That is what makes each style local to its scale, and it is what makes style mixing possible.

Worked Example — AdaIN on one tiny feature map

Take one 2×2 feature map $\boldsymbol{x}_i = \begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix}$. Its mean is 4 and its standard deviation is $\sqrt{5} \approx 2.236$.

Normalize. $(\boldsymbol{x}_i - 4)/2.236 = \begin{pmatrix} -1.342 & -0.447 \\ 0.447 & 1.342 \end{pmatrix}$. The pattern (which pixel is brighter) survives; the level and spread are gone.

Apply the style $y_{s,i} = 2$, $y_{b,i} = 0.5$: $\begin{pmatrix} -2.183 & -0.394 \\ 1.394 & 3.183 \end{pmatrix}$, which has mean 0.5 and standard deviation 2, exactly as the style asked.

Now feed in ten times the input, $\begin{pmatrix} 10 & 30 \\ 50 & 70 \end{pmatrix}$. It normalizes to the *same* matrix, so the output is identical. AdaIN throws away the incoming magnitude completely. Remember this: it is the root cause of the artifact StyleGAN2 had to fix (Section 6.4.9).

Evidence. Row C of Table 6.3: adding the mapping network and styles together improves FID from 6.11 to 5.34 (CelebA-HQ) and 5.25 to 4.85 (FFHQ). The two are introduced in one step, so this row cannot say how much each contributes on its own.

Component 4: noise inputs

What it does. Before each AdaIN, a single-channel image of fresh Gaussian noise is added to every feature map, scaled per channel by learned factors ("B" in Figure 6.32).

Why it is there. Some detail is genuinely random: where each hair falls, freckles, pores. Without a noise source, the network has to manufacture that randomness out of $\boldsymbol{w}$, wasting

capacity and producing repetitive patterns. With it, $\boldsymbol{w}$ can describe the person and the noise can describe the accidents (Figure 6.33).

Evidence. Row E: FFHQ FID 4.88 → 4.42. The paper's Table 3 also shows the perceptual path length (below) roughly halving, from 446.2 to 200.5 (full-path, FFHQ), a sign that the latent space became smoother once it no longer had to encode randomness.

Component 5: style mixing regularization

What it does. For 90% of training images, draw *two* latents, $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$, map both, and switch from $\boldsymbol{w}_1$ to $\boldsymbol{w}_2$ at a random layer. Figure 6.34 shows the same operation done deliberately at test time.

Why it is there. Without it, neighbouring layers could learn to rely on each other's styles being *correlated*, since in ordinary training they all come from one $\boldsymbol{w}$. Mixing breaks that correlation, so each layer's style must make sense on its own.

Evidence. This is where FID alone misleads. On ordinary samples, mixing barely matters (FID 4.42 → 4.40). The paper's Table 2 stress-tests the networks by mixing several latents at test time: with two latents, FID is 8.22 without mixing regularization and 5.11 with it at 90%; with four latents, 17.41 against 9.03. The regularizer buys robustness to exactly the editing operation StyleGAN is used for.

Component 6: the truncation trick in W

What it does. At sampling time only, pull each $\boldsymbol{w}$ towards the average $\bar{\boldsymbol{w}}$ (estimated by mapping many random $\boldsymbol{z}$):

$$\boldsymbol{w}' = \bar{\boldsymbol{w}} + \psi\,(\boldsymbol{w} - \bar{\boldsymbol{w}}), \qquad \psi < 1. \tag{6.32}$$

Why it is there. Rare regions of latent space are poorly trained, and that is where bad samples come from. Shrinking towards the centre trades variety for quality. At $\psi = 0$ every sample becomes the same "average face"; at $\psi = 1$ nothing changes. The paper's uncurated FFHQ samples use $\psi = 0.7$, applied to the low-resolution layers only.

Worked Example — truncating one latent

Pretend W is two-dimensional, with $\bar{\boldsymbol{w}} = (0.2, -0.1)$, and a sample lands at $\boldsymbol{w} = (1.4, 0.9)$. With $\psi = 0.7$: $\boldsymbol{w}' = (0.2, -0.1) + 0.7\,\big((1.4, 0.9) - (0.2, -0.1)\big) = (0.2 + 0.84,\ -0.1 + 0.70) = (1.04,\ 0.60)$. The sample keeps its *direction* from the average (which attributes it has) but moves 30% of the way back towards the centre (how extreme those attributes are).

How the paper measured "disentanglement"

Two metrics were introduced alongside the architecture, and both reappear in StyleGAN2.

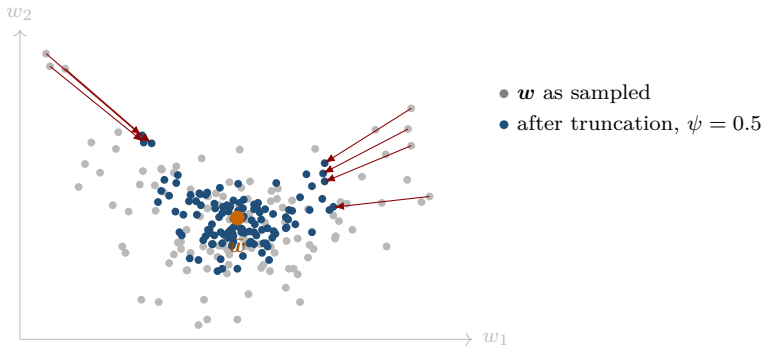


Figure 6.36. The truncation trick in two dimensions. Every sample slides along the line to the average $\bar{\boldsymbol{w}}$, halving its distance. The outlying samples, where the generator was trained least, are pulled into the well-trained core; the price is that the cloud, and so the variety of images, shrinks.

How to read it: The red arrows show the seven samples farthest from the centre. Truncation never changes which side of $\bar{\boldsymbol{w}}$ a sample is on, only how far out it sits.

- **Perceptual path length (PPL).** Walk a tiny step along an interpolation in latent space and measure how much the image changes perceptually. Average over many steps and pairs. A space where small steps sometimes cause sudden jumps (an extra ear appearing, say) is entangled; lower PPL means smoother.
- **Linear separability.** Label generated images with attribute classifiers (e.g. gender), then check how well a linear boundary in latent space can split them. Lower means the attribute is closer to a single straight direction.

6.4.9 StyleGAN2 and StyleGAN3: fixing the artifacts

StyleGAN's images were good enough that its remaining flaws became the research problem. Two follow-ups each start from a specific, visible artifact and trace it to a specific design choice.

StyleGAN2, problem 1: water droplets

What you see. Almost every StyleGAN image, and nearly every intermediate feature map, contains a blob-shaped artifact resembling a water droplet (Karras et al., 2020b).

What is going wrong. Recall the AdaIN worked example: normalization throws away each feature map's magnitude. The authors' hypothesis is that the generator *sneaks* magnitude information past the normalization by creating one strong, localized spike that dominates the statistics; with the spike setting the mean and variance, the rest of the map can be scaled as the generator likes. Their support: removing the normalization makes the droplets disappear completely.

The fix: modulate and demodulate the weights instead. Rather than normalizing the *activations* using their actual statistics, StyleGAN2 scales the *convolution weights* by the style,

$$w'_{ijk} = s_i \cdot w_{ijk}, \qquad w''_{ijk} = \frac{w'_{ijk}}{\sqrt{\sum_{i,k}(w'_{ijk})^2 + \epsilon}}, \tag{6.33}$$

where i indexes input feature maps, j output feature maps and k the kernel's spatial positions. The first step (modulation) applies the style. The second (demodulation) rescales each output map back to unit standard deviation, *assuming* unit-variance independent inputs. Because it uses this expected statistic rather than the actual content of the feature map, a spike no longer changes anything, and there is nothing to be gained by making one.

Worked Example — modulation and demodulation for one output map

Take a 1×1 convolution with two input maps and one output map, weights $\boldsymbol{w} = (0.5,\ 1.0)$, and a style $\boldsymbol{s} = (2,\ 0.5)$.

Modulate: $\boldsymbol{w}' = (2 \times 0.5,\ 0.5 \times 1.0) = (1.0,\ 0.5)$. The style has made input map 1 twice as influential as input map 2, reversing the original weights.

Demodulate: if both inputs have unit variance, the output's standard deviation would be $\sqrt{1.0^2 + 0.5^2} = \sqrt{1.25} \approx 1.118$. Divide by it: $\boldsymbol{w}'' \approx (0.894,\ 0.447)$, whose squared entries sum to 1.

The *relative* emphasis the style chose (2 to 1) is kept; only the overall scale is reset. Compare AdaIN, which reset the scale by measuring the actual activations, and so could be gamed by them.

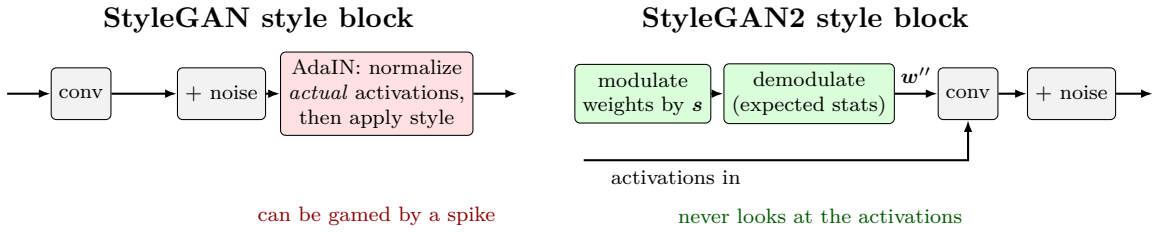


Figure 6.37. Where the style enters, before and after StyleGAN2. StyleGAN normalizes the activations themselves; StyleGAN2 folds the same scaling into the convolution's weights, using only an assumption about the input statistics.

How to read it: In the right-hand block, the activations only ever pass through the convolution and the noise. The style touches the weights, never the data, so nothing the data does can change the normalization.

StyleGAN2, problem 2: features stuck to the screen

What you see. When a face turns, some details (teeth, eyes) stay fixed relative to the image frame instead of moving with the face, then jump to the next "preferred" location. The paper calls these *phase artifacts*.

What is going wrong. The authors trace it to progressive growing. During training, each resolution is for a while the *output* resolution, so it learns to produce maximum-frequency detail; that leaves the final network with too much high frequency in its middle layers, which compromises shift invariance.

The fix. Drop progressive growing. Keep its benefit (coarse structure first) with a different architecture: a generator whose output is the sum of RGB images produced at every resolution ("output skips"), and a residual discriminator. The paper shows that such a network still concentrates on low resolutions early in training and shifts to high resolutions later, without any schedule.

StyleGAN2, problem 3: an uneven latent space

The fix: path length regularization. StyleGAN2 wants a fixed-size step in W to cause a fixed-size change in the image, whatever the direction. With $\mathbf{J}_{\boldsymbol{w}} = \partial g(\boldsymbol{w})/\partial \boldsymbol{w}$ the generator's Jacobian and $\boldsymbol{y}$ a random image-space direction, it penalizes

$$\mathcal{L}_{\text{pl}} = \mathbb{E}_{\boldsymbol{w},\boldsymbol{y}}\big(\|\mathbf{J}_{\boldsymbol{w}}^{\top}\boldsymbol{y}\|_2 - a\big)^2, \tag{6.34}$$

where a is a running average of the lengths themselves. It is the flow chapter's Jacobian (Section 6.3.2) put to a new use: not for a likelihood, but to keep the map well-conditioned. To save compute, both this and R1 are computed only once every 16 minibatches ("lazy regularization"), which the paper found did not hurt.

Table 6.4. StyleGAN2's ablation on FFHQ at 1024^2, from Table 1 of Karras et al. (2020b). Lower is better for both. Each row keeps everything above it.

	Configuration	FID	Path length
A	Baseline StyleGAN	4.40	212.1
B	+ weight demodulation	4.39	175.4
C	+ lazy regularization	4.38	158.0
D	+ path length regularization	4.34	122.5
E	+ no growing, new G and D architecture	3.31	124.5
F	+ large networks (= StyleGAN2)	2.84	145.0

Again, read the table honestly. Demodulation and path length regularization barely move FID: they were introduced to remove artifacts and smooth the latent space, and FID does not see either. The FID gains come from the new architecture without progressive growing (E) and from larger networks (F).

StyleGAN3: texture sticking and aliasing

What you see. In StyleGAN2 videos and interpolations, fine texture such as hair or stubble appears glued to the image coordinates while the face moves underneath it: "texture sticking".

What is going wrong. Karras et al. (2021) trace it to *aliasing*: careless signal processing (upsampling, pointwise nonlinearities) lets information about absolute pixel positions leak into the network, which then uses it.

The fix. Treat every signal in the network as a continuous image that merely happens to be sampled on a grid, and redesign each layer so that it never creates frequencies the grid cannot represent: careful low-pass filtering around upsampling and around the nonlinearities.

Result. The paper reports FID matching StyleGAN2, with networks that are fully equivariant to translation and rotation, even at sub-pixel scales. Its goal was not better still images but generators better suited to video and animation.

6.4.10 BigGAN: the same game at ImageNet scale

Model at a glance: BigGAN	Brock, Donahue & Simonyan, 2019
In one line	A class-conditional GAN on all of ImageNet, improved mainly by making it much larger, plus three tricks that make large scale work (Brock et al., 2019).
Conditioning	The class label enters G through class-conditional batch normalization (each class has its own scale and shift, much like a StyleGAN style) and enters D through a projection discriminator; both are explained in Section 6.4.12.
Scale	Batch size 256 → 2048 improves Inception Score by 46%; widening every layer by 50% (about twice the parameters) adds a further 21%.
Trick 1: shared embeddings	One class embedding shared across all conditional batch-norm layers, instead of a separate one per layer.
Trick 2: skip-z	The noise vector is split into chunks fed to several layers, not only the first: the same "inject the latent at every scale" idea as StyleGAN, in a simpler form.
Trick 3: truncation	Sample $\boldsymbol{z}$ from a truncated normal (values beyond a threshold are resampled), trading variety for fidelity. Large models often break under truncation; with *orthogonal regularization* of G, 60% of the paper's runs tolerated it, against 16% without.
Result	At 128 × 128 on ImageNet, IS 166.5 and FID 7.4, against the previous best IS 52.52 and FID 18.65.
Problems it exposed	Training collapse at scale, and a discriminator that memorizes ImageNet (Problems 4 and 5 of Section 6.4.5).
Relation to StyleGAN	Both put the latent into many layers and both use a truncation trick. BigGAN truncates $\boldsymbol{z}$ and conditions on a class; StyleGAN truncates $\boldsymbol{w}$ and is unconditional.

Table 6.5. BigGAN's ablation on ImageNet at 128^2, from Table 1 of Brock et al. (2019). Rows are cumulative. The first two rows are single runs; the others are means over eight runs. "Collapse" gives the iteration (in thousands) at which training fell apart; "stable" means it survived 10^6 iterations.

Change	Batch	FID	IS	Training
SA-GAN baseline	256	18.65	52.52	stable
larger batch	2048	12.39	76.85	collapses at 732
+ 50% wider	2048	9.54	92.98	collapses at 295
+ shared embeddings	2048	9.18	94.94	collapses at 185
+ skip-$\boldsymbol{z}$	2048	8.73	98.76	collapses at 152
+ orthogonal regularization	2048	8.51	99.31	collapses at 165

The last column is the lesson. Almost every change that improved quality also made training collapse *sooner* (orthogonal regularization delayed it slightly); the reported scores come from the checkpoint saved just before collapse.

6.4.11 The StyleGAN family and its relatives, side by side

Table 6.6. How each model in this section changed its predecessor. Read down the "problem" column: each row exists because of a limitation in a row above it.

Model	Year	Problem it attacked	Key idea
GAN	2014	density-free generative modeling	two-player game
DCGAN	2016	convolutional GANs would not train	five architecture rules
WGAN / -GP	2017	no gradient when supports do not overlap	Wasserstein distance; gradient penalty
ProGAN	2018	high resolutions unstable	grow from 4^2 to 1024^2 with fade-in
SN-GAN	2018	discriminator changes too fast	spectral normalization
BigGAN	2019	ImageNet-scale class-conditional generation	scale, shared embeddings, skip-$\boldsymbol{z}$, truncation
StyleGAN	2019	no control over what the latent does	mapping network, per-layer styles (AdaIN), noise, mixing
StyleGAN2	2020	droplet and phase artifacts	weight demodulation; no progressive growing; path length regularization
StyleGAN2 + ADA	2020	small datasets overfit D	adaptive, non-leaking augmentation
StyleGAN3	2021	texture sticking	alias-free signal processing

Key takeaway — two threads through the whole family

Two ideas recur in every row of Table 6.6. First, *inject the latent at every scale* (skip-$\boldsymbol{z}$ in BigGAN, styles in StyleGAN), so coarse and fine structure get separate controls. Second,

control the discriminator's slope (gradient penalties, spectral normalization, R1), because a discriminator that changes too sharply is what breaks the game. Most of the progress from 2016 to 2021 is careful engineering of those two ideas, measured one ablation row at a time.

6.4.12 Conditional GANs: telling the generator what to make

Every GAN so far samples from $p(\boldsymbol{x})$: you draw noise and get *some* image. You cannot ask for a seven, a cat, or "a small yellow bird with black wings". A *conditional* GAN adds a second input c, the condition, and learns $p(\boldsymbol{x} \mid c)$ instead: draw noise, choose c, and get an image *of that kind*.

The condition can be almost anything: a class label, a sentence, or a whole image (Pix2Pix, Section 6.4.13). Two questions organize this section. *What does the game optimize* once a condition is added? And *where in the networks should the condition go*? The second question has at least four standard answers, and they perform very differently.

Symbol	Meaning
$\boldsymbol{x}$	an image (real or generated)
c	the condition: a class, a sentence, an image. For a class out of K, written as a one-hot vector $\boldsymbol{c} \in \{0,1\}^K$
$\boldsymbol{z}$	the noise, as before
$G(\boldsymbol{z}, c)$	the generator: noise plus condition in, image out
$D(\boldsymbol{x}, c)$	the discriminator: the probability that $\boldsymbol{x}$ is a real image *of condition* c
$p(\boldsymbol{x} \mid c)$, $p_G(\boldsymbol{x} \mid c)$	real images of condition c, and the generator's images when asked for c

The objective and what it optimizes

Mirza and Osindero (2014) add the condition to both players and change nothing else:

$$\min_G \max_D \; \mathbb{E}_{(\boldsymbol{x},c)\sim p_{\text{data}}}\big[\log D(\boldsymbol{x}, c)\big] + \mathbb{E}_{c\sim p(c),\,\boldsymbol{z}}\big[\log\big(1 - D(G(\boldsymbol{z}, c), c)\big)\big]. \tag{6.35}$$

Real pairs come from the dataset. Fake pairs are made by drawing a condition c with the same frequencies as in the data, then generating an image for it.

The best discriminator. Real pairs have density $p(c)\,p(\boldsymbol{x} \mid c)$ and fake pairs $p(c)\,p_G(\boldsymbol{x} \mid c)$. So

$$V(G, D) = \sum_c p(c) \int \Big[p(\boldsymbol{x} \mid c) \log D(\boldsymbol{x}, c) + p_G(\boldsymbol{x} \mid c) \log\big(1 - D(\boldsymbol{x}, c)\big)\Big]\, d\boldsymbol{x}.$$

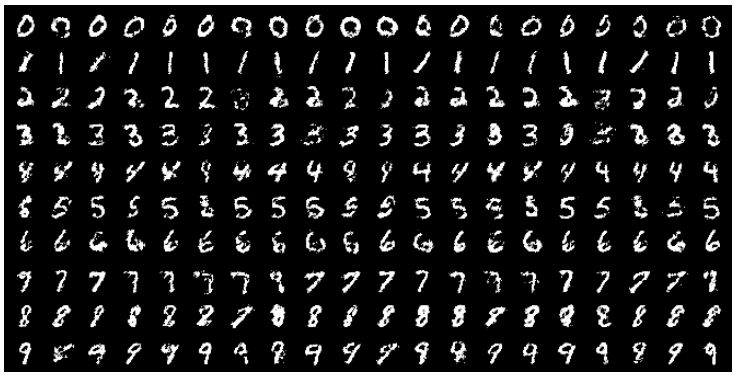


Figure 6.38. The first conditional GAN's output (Mirza and Osindero, 2014). Each row was generated by asking for one digit, 0 at the top to 9 at the bottom; within a row only the noise $\boldsymbol{z}$ changes. The condition decides *which* digit; the noise decides the handwriting.

How to read it: Read across a row to see what the noise controls (slant, thickness, style), and down a column to see what the condition controls (the digit's identity).

Source: Figure 2 of Mirza and Osindero (2014).

Each $(\boldsymbol{x}, c)$ can be maximized separately with the same one-line calculus as Equation (6.24), giving

$$D^*(\boldsymbol{x}, c) = \frac{p(\boldsymbol{x} \mid c)}{p(\boldsymbol{x} \mid c) + p_G(\boldsymbol{x} \mid c)}. \tag{6.36}$$

Substituting back, the "$-2\log 2$ plus two KLs" algebra of Section 6.4.2 goes through once per condition:

$$V(G, D^*) = 2 \sum_c p(c)\, \mathrm{JS}\big(p(\cdot \mid c) \,\|\, p_G(\cdot \mid c)\big) - 2\log 2. \tag{6.37}$$

In words: one JS divergence per condition, weighted by how common the condition is. The minimum is reached only when $p_G(\cdot \mid c) = p(\cdot \mid c)$ for *every* condition. Good images are not enough; they must be good images *of what was asked for*.

Why the discriminator must see c. Suppose only the generator gets the condition and the discriminator judges $D(\boldsymbol{x})$ alone. Then fake images are distributed as the mixture $p_G(\boldsymbol{x}) = \sum_c p(c)\, p_G(\boldsymbol{x} \mid c)$, and the same derivation gives $V = 2\,\mathrm{JS}\big(p(\boldsymbol{x}) \,\|\, p_G(\boldsymbol{x})\big) - 2\log 2$. Only the *overall* mix of images is checked.

Worked Example — a generator that ignores the label

MNIST, with the ten digits equally common. A lazy generator draws a *random* digit whatever it is asked for: $p_G(\boldsymbol{x} \mid c) = p(\boldsymbol{x})$ for every c.

Discriminator without c. The generator's overall mix is $\sum_c \frac{1}{10} p(\boldsymbol{x}) = p(\boldsymbol{x})$, exactly the data. JS is 0: a perfect score, although the label is completely ignored.

Discriminator with c. Asked for a 7, the generator produces a 7 only a tenth of the time. The pair (an image of a 3, label 7) never occurs in the real data, so D^* gives it 0. Now each per-condition JS is large, and the generator is pushed to respect the label.

The discriminator's score, split into two parts. Write the discriminator as a sigmoid of a score, $D = \sigma(f)$. The optimal score is the log-ratio of real to fake pair densities, and Bayes' rule splits it (Miyato and Koyama, 2018):

$$f^*(\boldsymbol{x}, c) = \log \frac{p(\boldsymbol{x}, c)}{p_G(\boldsymbol{x}, c)} = \underbrace{\log \frac{p(c \mid \boldsymbol{x})}{p_G(c \mid \boldsymbol{x})}}_{\substack{\text{does the label} \\ \text{fit the image?}}} + \underbrace{\log \frac{p(\boldsymbol{x})}{p_G(\boldsymbol{x})}}_{\substack{\text{does the image} \\ \text{look real?}}}. \tag{6.38}$$

Every design below is a different way of building a network that can represent these two parts. Keep Equation (6.38) in mind: it is what makes the designs comparable.

Where the condition goes: four designs

Figure 6.39 shows the options side by side. The first three are about the discriminator; the fourth is about the generator, and is usually combined with one of the first three.

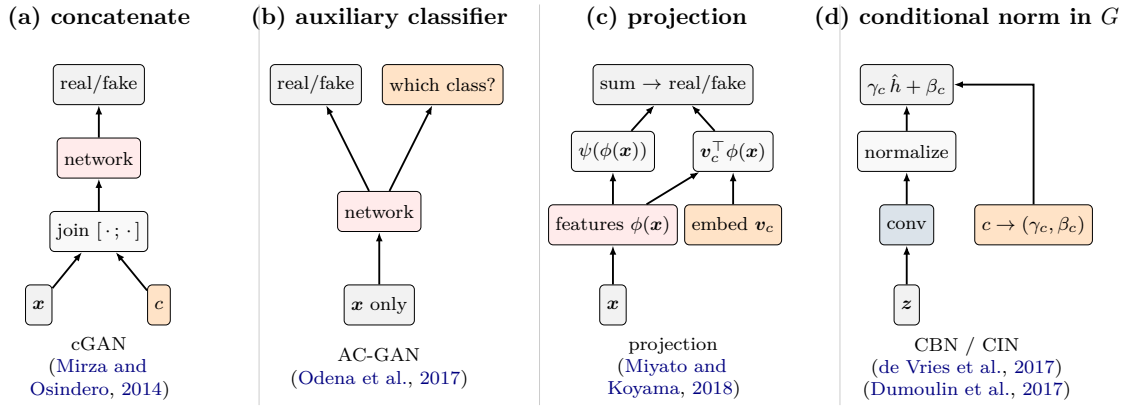


Figure 6.39. Four places to put the condition (orange). (a) Join it to the input or to a hidden layer. (b) Give the discriminator no condition, but a second head that must predict the class. (c) Take an inner product between a learned class vector and the image features, and add a class-free realism score. (d) In the generator, let the condition choose the scale and shift after each normalization layer.

How to read it: Follow the orange box in each panel. In (a) it enters at the bottom and mixes with everything; in (b) it appears only as a training target; in (c) it enters at the very end, in one multiplication; in (d) it sets two numbers per channel.

Design 1: concatenation (the original cGAN)

What it does. Append the condition to the input, or to a hidden layer, and let the network figure out the rest.

The original architecture. Mirza and Osindero (2014) used small fully connected networks on MNIST, with every size stated in the paper (Figure 6.40). In the generator, 100-dimensional noise drawn uniformly from the unit cube goes to a ReLU layer of 200 units; the one-hot label goes to its own ReLU layer of 1000 units; the two are joined into a combined ReLU layer of 1200 units, and a sigmoid layer outputs the $784 = 28 \times 28$ pixels. The discriminator mirrors this with maxout layers: the image to 240 units, the label to 50, then a joint layer of 240 units and a sigmoid.

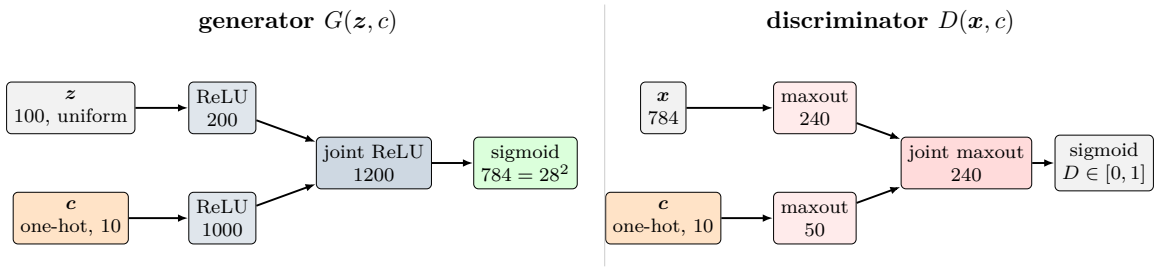


Figure 6.40. The conditional GAN of Mirza and Osindero (2014) on MNIST, with the layer sizes reported in the paper. In both networks the label gets its own first layer, and the two streams are joined in a shared hidden layer.

How to read it: The orange boxes are the only change from an unconditional GAN. Delete them and their arrows and you have the original GAN on MNIST.

Worked Example — what joining a one-hot label actually does

Take a layer that computes $W[\boldsymbol{h}; \boldsymbol{c}]$, where $\boldsymbol{h}$ is the rest of its input and $\boldsymbol{c}$ a one-hot label for $K = 3$ classes. Split the weight matrix into the part for $\boldsymbol{h}$ and the part for $\boldsymbol{c}$:

$$W[\boldsymbol{h}; \boldsymbol{c}] = W_h \boldsymbol{h} + W_c \boldsymbol{c}.$$

With $W_c = \begin{pmatrix} 0.5 & -1.0 & 2.0 \\ 0.0 & 0.3 & -0.4 \end{pmatrix}$ and $\boldsymbol{c} = (0, 1, 0)$ (class 2), $W_c \boldsymbol{c} = (-1.0,\ 0.3)$: just the second column. So a joined one-hot label is nothing more than a *learned bias vector per class*, added before the nonlinearity. Everything the condition can do has to be built up from those per-class biases by later layers.

That is the weakness Miyato and Koyama (2018) point out: nothing about concatenation reflects the structure of Equation (6.38), so the network must discover how to compare label and image on its own. For convolutional networks the condition is joined to a hidden feature map by copying it to every spatial position and stacking it as extra channels ("depth concatenation"), as the text-to-image GAN below does.

Design 2: an auxiliary classifier (AC-GAN)

What it does. Odena et al. (2017) give the discriminator no condition at all. Instead it has two output heads: $P(S \mid \boldsymbol{x})$, the probability that the image is real (S for source), and $P(C \mid \boldsymbol{x})$, a distribution over the classes. The generator still receives the class. Two log-likelihoods are defined,

$$L_S = \mathbb{E}\big[\log P(S = \text{real} \mid \boldsymbol{x}_{\text{real}})\big] + \mathbb{E}\big[\log P(S = \text{fake} \mid \boldsymbol{x}_{\text{fake}})\big], \tag{6.39}$$

$$L_C = \mathbb{E}\big[\log P(C = c \mid \boldsymbol{x}_{\text{real}})\big] + \mathbb{E}\big[\log P(C = c \mid \boldsymbol{x}_{\text{fake}})\big], \tag{6.40}$$

and the discriminator maximizes $L_S + L_C$ while the generator maximizes $L_C - L_S$. Both players want every image, real or fake, classified as its intended class; they disagree only about real versus fake.

Worked Example — the generator's AC-GAN reward on one sample

The generator is asked for class c and produces $\boldsymbol{x}$. The discriminator says $P(S = \text{fake} \mid \boldsymbol{x}) = 0.7$ and $P(C = c \mid \boldsymbol{x}) = 0.9$. This sample's contribution to the generator's objective $L_C - L_S$ is

$$\log 0.9 - \log 0.7 \approx -0.105 + 0.357 = 0.252.$$

Now compare a second sample that is *more realistic* ($P(\text{fake}) = 0.5$) but less obviously of its class ($P(C = c) = 0.6$): $\log 0.6 - \log 0.5 \approx -0.511 + 0.693 = 0.182$. The objective prefers the first one. Nothing stops the generator from trading realism and variety for images that are simply *easy to classify*.

That is the known weakness. The projection paper argues that auxiliary classifiers encourage images that are particularly easy to classify, which pulls $p_G(\boldsymbol{x} \mid c)$ away from the true conditional. It notes that the AC-GAN authors saw the problem grow with the number of classes, and reports that their own AC-GAN on all 1000 ImageNet classes ended up generating only one image for most classes (Miyato and Koyama, 2018). The original AC-GAN results on ImageNet in fact used 100 separate models, each trained on 10 classes (Odena et al., 2017).

Design 3: the projection discriminator

The derivation. Miyato and Koyama (2018) start from Equation (6.38) and ask what form the first part takes if the class is predicted from features the usual way, by a softmax over linear scores. Suppose both the real and the generated distributions have

$$\log p(c \mid \boldsymbol{x}) = \boldsymbol{v}_c^{\text{real}\,\top} \phi(\boldsymbol{x}) - \log Z^{\text{real}}(\boldsymbol{x}), \qquad \log p_G(c \mid \boldsymbol{x}) = \boldsymbol{v}_c^{G\,\top} \phi(\boldsymbol{x}) - \log Z^{G}(\boldsymbol{x}),$$

where $\phi(\boldsymbol{x})$ is a feature vector, $\boldsymbol{v}_c$ is one weight vector per class and $Z(\boldsymbol{x})$ the softmax's normalizer. Subtract:

$$\log \frac{p(c \mid \boldsymbol{x})}{p_G(c \mid \boldsymbol{x})} = \underbrace{(\boldsymbol{v}_c^{\text{real}} - \boldsymbol{v}_c^{G})^\top}_{=:\, \boldsymbol{v}_c^\top} \phi(\boldsymbol{x}) \; - \; \underbrace{\left(\log Z^{\text{real}}(\boldsymbol{x}) - \log Z^{G}(\boldsymbol{x})\right)}_{\text{depends on } \boldsymbol{x} \text{ only}}.$$

The normalizer term has no c in it, so it can be merged with the second part of Equation (6.38) into a single class-free function $\psi(\phi(\boldsymbol{x}))$. With $\boldsymbol{c}$ one-hot and V the matrix whose rows are the $\boldsymbol{v}_c$:

$$f(\boldsymbol{x}, c) = \boldsymbol{c}^\top V \phi(\boldsymbol{x}) + \psi(\phi(\boldsymbol{x})) = \boldsymbol{v}_c^\top \phi(\boldsymbol{x}) + \psi(\phi(\boldsymbol{x})). \tag{6.41}$$

The label enters in exactly one place: an inner product (a *projection*) of the image features onto the class's learned vector.

Worked Example — the same image under two labels

Let an image have features $\phi(\boldsymbol{x}) = (2,\, 1)$ and realism score $\psi(\phi(\boldsymbol{x})) = 0.3$, with learned class vectors $\boldsymbol{v}_{\text{cat}} = (1,\, 0.5)$ and $\boldsymbol{v}_{\text{dog}} = (-0.5,\, 1)$.

$$f(\boldsymbol{x}, \text{cat}) = 2 + 0.5 + 0.3 = 2.8, \qquad D = \sigma(2.8) \approx 0.94;$$

$$f(\boldsymbol{x}, \text{dog}) = -1 + 1 + 0.3 = 0.3, \qquad D = \sigma(0.3) \approx 0.57.$$

The realism part is shared; only the projection changes. The discriminator calls this a convincing *cat*, and a far less convincing *dog*.

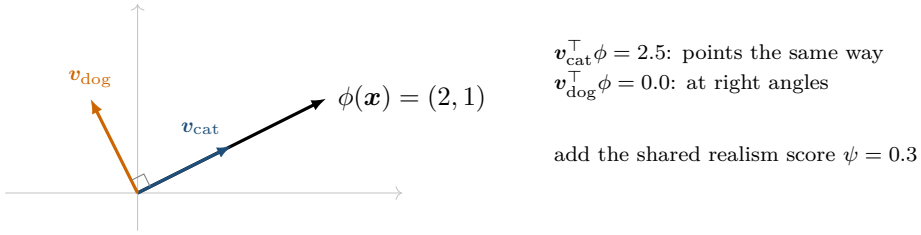


Figure 6.41. The projection discriminator's label term, drawn for the worked example. The score for a class is how far the image's feature vector points along that class's learned direction. Here $\phi(\boldsymbol{x})$ is exactly perpendicular to $\boldsymbol{v}_{\text{dog}}$, so the "dog" label adds nothing.

How to read it: Rotate $\phi(\boldsymbol{x})$ in your head towards $\boldsymbol{v}_{dog}$: the cat score falls and the dog score rises, while the realism score ψ stays put.

Evidence. The paper compares the three discriminator designs with the same ResNet backbone on class-conditional ImageNet. Besides the Inception Score it reports *intra-FID*: FID computed separately within each class and then averaged, so a model that collapses within a class is penalized even if its classes look right overall.

Read both columns. AC-GAN's Inception Score is close to projection's, but its intra-FID is by far the worst: its images are easy to classify (which IS rewards) but not diverse within a

Table 6.7. Class-conditional ImageNet, from Table 1 of Miyato and Koyama (2018): higher Inception Score and lower intra-FID are better. The AC-GAN row is its best checkpoint, since its training collapsed before the end.

Discriminator design	Inception Score	Intra-FID
AC-GAN (auxiliary classifier)	28.5 ± 0.20	260.0
Concatenation (hidden layer)	21.1 ± 0.35	141.2
Projection	29.7 ± 0.61	103.1
Projection, trained longer (850K iterations)	36.8 ± 0.44	92.4

class (which intra-FID catches), exactly the failure the worked example predicts. The paper also gives the method's limit honestly: the derivation assumes a simple, unimodal $p(c \mid \boldsymbol{x})$, and when that fails, projection may not be the right choice. BigGAN's discriminator uses projection (Section 6.4.10).

Design 4: conditional normalization in the generator

The first three designs are about the discriminator. The generator also has to use the condition, and the most effective way turned out to be through normalization layers.

What it does. A normalization layer standardizes each channel and then applies a learned scale γ and shift β. *Conditional* normalization gives each condition its own:

$$\mathrm{CN}(\boldsymbol{h}, c) = \gamma_c \frac{\boldsymbol{h} - \mu(\boldsymbol{h})}{\sigma(\boldsymbol{h})} + \beta_c. \tag{6.42}$$

Dumoulin et al. (2017) introduced this for style transfer, storing γ and β as $N \times C$ matrices (N styles, C channels); they report that one network with all convolution weights shared could learn 32 styles at once, with only these per-style parameters differing. de Vries et al. (2017) predicted the changes to γ and β from a sentence embedding with a small network, to let language modulate a vision network. BigGAN uses class-conditional batch normalization, with one shared class embedding linearly projected to every layer's gains and biases (Brock et al., 2019).

Worked Example — one channel, two classes

Reuse the AdaIN example's feature map, which standardizes to $(-1.342, -0.447, 0.447, 1.342)$. With class A's parameters $\gamma_A = 2$, $\beta_A = 0.5$ the output is $(-2.18, -0.39, 1.39, 3.18)$: a strong, bright channel. With class B's $\gamma_B = 0.5$, $\beta_B = -1$ it is $(-1.67, -1.22, -0.78, -0.33)$: a faint, dark one. The convolution weights are identical; the class only decides how loud each channel is and where it is centred.

Common Confusion

Conditional normalization and StyleGAN's AdaIN are the same operation. Compare Equation (6.42) with Equation (6.31). The only difference is where γ and β come from: a lookup by class here, a learned affine map of $\boldsymbol{w}$ in StyleGAN. StyleGAN is, in this sense, a generator conditioned on its own latent code at every layer.

Conditioning on text: the matching-aware discriminator

Reed et al. (2016) condition on a sentence. A pretrained text encoder produces an embedding $\varphi(t)$, which is compressed to 128 dimensions and joined to 100-dimensional noise in the generator; in the discriminator it is copied across the 4×4 feature map and stacked as extra channels (Design 1). Images are 64×64.

Their key change is to the *training data* of the discriminator. A plain conditional discriminator sees two kinds of pairs: real image with its text (real), and generated image with its text (fake). It then has to learn "does this image match this text?" only from fake images, which early in training are bad at everything, so the two kinds of error are mixed up. GAN-CLS adds a third kind:

Pair shown to D	Score	Target
real image, matching text	$s_r = D(\boldsymbol{x}, h)$	real
real image, *mismatched* text	$s_w = D(\boldsymbol{x}, \hat{h})$	fake
generated image, matching text	$s_f = D(\hat{\boldsymbol{x}}, h)$	fake

and the discriminator maximizes

$$L_D = \log s_r + \tfrac{1}{2}\big(\log(1 - s_w) + \log(1 - s_f)\big), \tag{6.43}$$

while the generator maximizes $\log s_f$. The middle row is the point: a perfectly realistic image can still be wrong *for this text*, and now the discriminator is taught that directly. It is the same lesson as the label-ignoring worked example above, applied as a data trick.

The designs side by side

Table 6.8. Ways to condition a GAN. The first three are discriminator designs; the fourth is a generator design, usually combined with one of them.

Design	How the condition enters	Known issue	Used in
Concatenation	joined to the input or a hidden layer, i.e. a per-class bias	no built-in structure; weakest in the comparison above	cGAN, text-to-image GAN, Pix2Pix
Auxiliary classifier	only as a classification target for a second head	rewards easy-to-classify images; loses diversity with many classes	AC-GAN
Projection	inner product $\boldsymbol{v}_c^\top \phi(\boldsymbol{x})$ added to a realism score	assumes a simple $p(c \mid \boldsymbol{x})$	BigGAN
Conditional norm (in G)	per-condition γ_c, β_c after each normalization	none specific; needs one pair per layer per condition, or an embedding	BigGAN; style transfer; StyleGAN's AdaIN is the same form

Key takeaway — conditional GANs in three lines

(1) The objective is one JS divergence per condition (Equation (6.37)), so the generator must be right for each condition, not just on average. (2) That only works if the discriminator sees the condition; a discriminator that does not can be fooled by a generator that ignores it. (3) How the condition is fed in matters as much as whether it is: the optimal score splits into "does the label fit?" plus "does it look real?", and the projection discriminator is built to match that split.

Common Confusion

A conditional GAN is only as self-supervised as its condition. Class labels and captions are human annotations, so class- and text-conditional GANs are supervised in the sense of this book. Conditions that can be computed from the data itself (a low-resolution copy, an edge map, a greyscale version) keep the method self-supervised. The next section meets the same distinction in Pix2Pix.

6.4.13 Pix2Pix: a conditional GAN for image-to-image translation

Section 6.3.12 added conditions to flows. Slides 36–38 add conditions to GANs, via *Image-to-Image Translation with Conditional Adversarial Networks*—Pix2Pix—by Isola, Zhu, Zhou, and Efros (Isola et al., 2017) (slide 36; the idea of a conditional GAN itself is due to Mirza and Osindero (Mirza and Osindero, 2014)). The crucial contrast, drawn explicitly on slide 37,

is with conditional flows: *the GAN generator is not architecturally restricted.* No invertibility, no Jacobian, no dimension preservation. The condition $\boldsymbol{x}$ is just another input tensor, concatenated with noise (or used alone, deterministically).

Two changes distinguish the conditional setup from the vanilla game. First, the discriminator also sees the condition: it scores pairs $(G(\boldsymbol{x}), \boldsymbol{x})$ as "fake" versus real pairs $(\boldsymbol{y}, \boldsymbol{x})$ as "real"—enough to penalize a generator that produces a plausible output that *ignores* $\boldsymbol{x}$. Second, the generator gets an additional reconstruction term against the paired ground truth:

$$G^* = \arg\min_G \max_D \; \mathcal{L}_{\text{cGAN}}(G, D) + \lambda \, \mathcal{L}_{L1}(G), \tag{6.44}$$

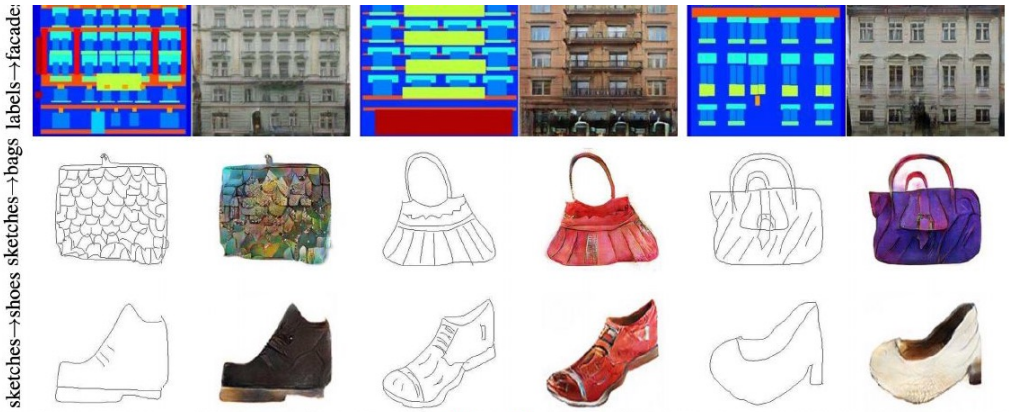


Figure 6.42. Pix2Pix across several image-to-image tasks. The same architecture and the same loss handle all of them; only the training pairs change.

How to read it: Ask, for each task, where the training pairs would have come from. Edges→photo pairs can be produced automatically by running an edge detector over photographs, so that task is genuinely self-supervised. Labels→street-scene pairs need a human-annotated segmentation map, so that one is not — which is exactly the caveat the lecture flags in bold on this slide.

Source: Lecture 04, slide 39 (Isola et al., 2017).

where $\mathcal{L}_{L1}(G) = \mathbb{E}\|\boldsymbol{y} - G(\boldsymbol{x}, \boldsymbol{z})\|_1$. The adversarial term encourages sharp realism; the L_1 term pins the output to the conditioning input. (The published architecture uses a U-Net generator (Ronneberger et al., 2015) and a *PatchGAN* discriminator classifying small patches rather than whole images. The lecture mentions the former through the U-Net reference; both are developed in full below.) Slide 39 shows the teaser grid: labels→street scene, labels→facade, BW→color, aerial→map, day→night, edges→photo, edges→shoes.

Notation: which letter is the condition?

Pix2Pix writes $\boldsymbol{x}$ for the *input* image (the label map, the edges, the greyscale photo) and $\boldsymbol{y}$ for the *target* image it should turn into. The generator is $G(\boldsymbol{x}, \boldsymbol{z})$, and a training set is a list of pairs $(\boldsymbol{x}, \boldsymbol{y})$. Be careful when reading older papers: Mirza and Osindero (2014) write the condition as $\boldsymbol{y}$ and the data as $\boldsymbol{x}$, the opposite way round. This section uses Pix2Pix's convention throughout.

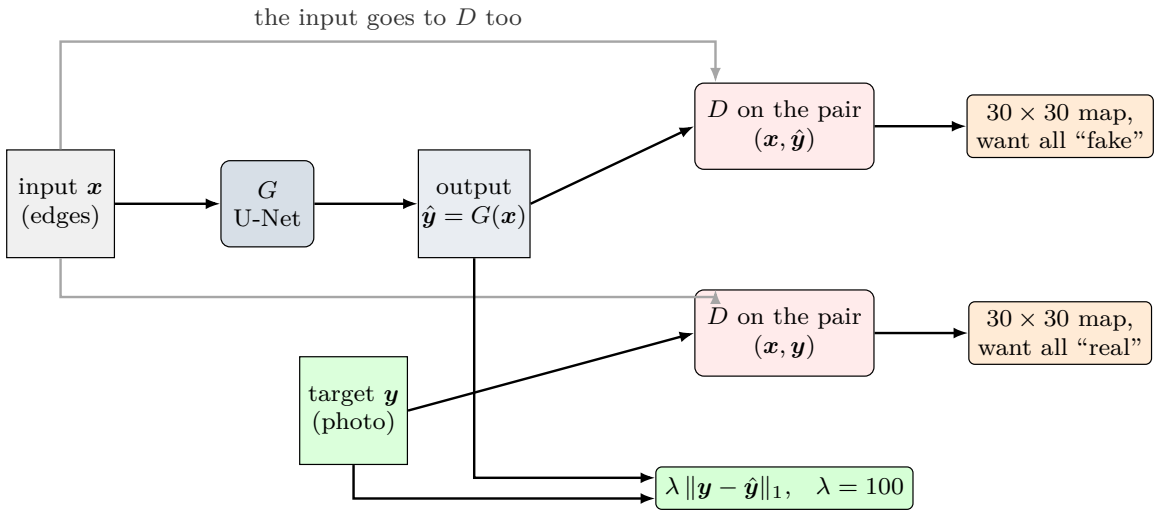


Figure 6.43. One Pix2Pix training step. The generator turns the input into $\hat{\boldsymbol{y}}$. The discriminator never judges an image alone: it judges *pairs*, input plus output, and it answers once per patch rather than once per image. The generator is trained to make the fake pair's patches look real *and* to stay close to the target in L_1.

How to read it: Follow the grey arrows: the input $\boldsymbol{x}$ reaches the discriminator on both paths. That wire is what makes the GAN conditional. *Remove it and D could no longer tell whether the output belongs to this input.*

The conditional GAN objective, in Pix2Pix's letters

Pix2Pix is the conditional GAN of Section 6.4.12 with the condition being an image. Renaming $c \to \boldsymbol{x}$ (the input) and $\boldsymbol{x} \to \boldsymbol{y}$ (the output), Equation (6.35) becomes the paper's

$$\mathcal{L}_{\text{cGAN}}(G, D) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}}\big[\log D(\boldsymbol{x}, \boldsymbol{y})\big] + \mathbb{E}_{\boldsymbol{x},\boldsymbol{z}}\big[\log(1 - D(\boldsymbol{x}, G(\boldsymbol{x}, \boldsymbol{z})))\big], \tag{6.45}$$

and everything derived there carries over: the game is one JS divergence per input image, so the generator must produce the right output *for each input*, and a discriminator that sees only $\boldsymbol{y}$ checks nothing but the overall mix of outputs. With images, that failure looks like this.

Worked Example — a generator that fools the unconditional discriminator

Suppose there are two inputs, equally common: $\boldsymbol{x}_a$ ("road on the left") and $\boldsymbol{x}_b$ ("road on the right"), with correct outputs $\boldsymbol{y}_a$ and $\boldsymbol{y}_b$. A lazy generator outputs $\boldsymbol{y}_b$ when given $\boldsymbol{x}_a$ and $\boldsymbol{y}_a$ when given $\boldsymbol{x}_b$.

Unconditional D. Real outputs are $\boldsymbol{y}_a$ half the time and $\boldsymbol{y}_b$ half the time. So are the generator's. The two distributions are identical, JS is 0, and the generator gets a perfect score while being wrong on every single input.

Conditional D. Real pairs are $(\boldsymbol{x}_a, \boldsymbol{y}_a)$ and $(\boldsymbol{x}_b, \boldsymbol{y}_b)$. Fake pairs are $(\boldsymbol{x}_a, \boldsymbol{y}_b)$ and $(\boldsymbol{x}_b, \boldsymbol{y}_a)$, which never occur in the data. So $D^* = 0$ on every fake pair, the per-input JS is at its maximum $\log 2$, and the generator is penalized as hard as possible.

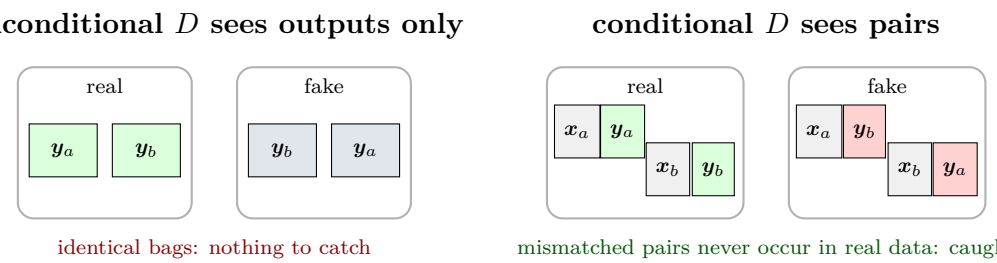


Figure 6.44. The swapped-output generator of the worked example, as each discriminator sees it. Judged on outputs alone, fake and real are the same collection. Judged as pairs, every fake is a combination the data never contains.

How to read it: Compare the two right-hand bags. They hold the same images; only the pairing with the input differs. The conditional discriminator is the only one that looks at pairing.

Why an L_1 loss alone gives blurry images

Pix2Pix adds $\lambda \mathcal{L}_{L1}$ to the adversarial loss. To see why both terms are needed, look at what a pixel-wise loss does on its own when the answer is uncertain.

Mean versus median. Fix one pixel and let v be its true value, which is random because many targets are plausible for the same input. If the network must output a single number c:

- the *squared* loss $\mathbb{E}[(v-c)^2]$ is minimized by the **mean** of v;
- the *absolute* loss $\mathbb{E}|v-c|$ is minimized by the **median** of v.

(For the squared loss, set the derivative $-2\mathbb{E}[v-c]$ to zero. For the absolute loss, the derivative is $P(v<c) - P(v>c)$, which is zero where half the probability lies on each side.)

Worked Example — one uncertain pixel on an edge

Say an edge might fall just left or just right of a pixel, so the pixel is black ($v=0$) with probability 0.6 and white ($v=1$) with probability 0.4.

Squared loss: $\mathbb{E}[(v-c)^2] = 0.6c^2 + 0.4(1-c)^2$, minimized at $c = 0.4$, a grey that *never* occurs in the data.

Absolute loss: for $0 \le c \le 1$, $\mathbb{E}|v-c| = 0.6c + 0.4(1-c) = 0.4 + 0.2c$, minimized at $c = 0$, the median: a value that does occur.

Now make it 0.5 and 0.5. The squared loss still picks 0.5 grey. The absolute loss becomes 0.5 for *every* $c \in [0, 1]$: it is completely indifferent. L_1 gives no reason to prefer a sharp answer over a grey one. That flat stretch is exactly where the adversarial term takes over.

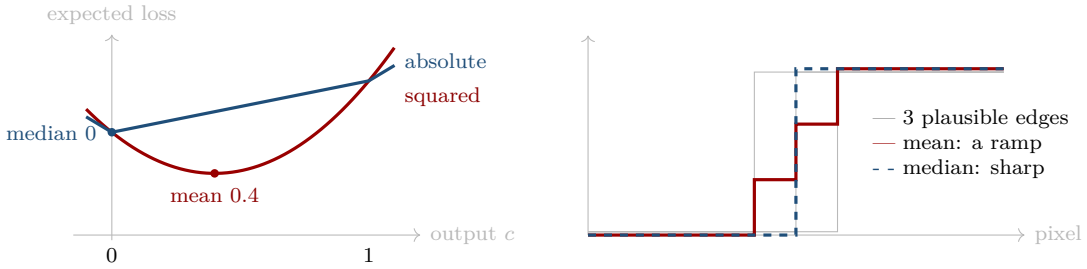


Figure 6.45. Left: the expected loss for the 0.6/0.4 pixel of the worked example. The squared loss bottoms out at a grey that never occurs; the absolute loss at the median, a real value. Right: an edge that could be at pixel 4, 5 or 6, equally likely. The per-pixel mean is a blurred ramp; the per-pixel median is a sharp edge. In a single dimension the median is sharp; across a whole image the per-pixel medians of many plausible images need not form a plausible image, which is why L_1 results are still blurry in practice.

How to read it: On the left, find each curve's lowest point. On the right, the three grey steps are three equally good answers; averaging them (red) produces something none of them is.

Pix2Pix's own evidence agrees: L_1 alone gives "reasonable but blurry" results, and on colorization L_1 produces a narrower spread of colours than the ground truth, which the authors read as confirming that L_1 encourages "average, grayish colors" when unsure (Isola et al., 2017).

What the adversarial term adds. The discriminator has seen real images, and a grey edge or a greyish sky does not look like them. So wherever L_1 is indifferent or pulls towards an average, the adversarial term pulls towards a value that actually occurs: a sharp edge, a real colour. The division of labour the paper proposes is:

- L_1 gets the *low frequencies* right: overall layout, where things are, rough colour.
- The discriminator gets the *high frequencies* right: texture, sharpness, fine detail.

The weight $\lambda = 100$ in Equation (6.44) reflects that the L_1 term does most of the steering on layout; the adversarial term is decisive exactly where L_1 has no preference.

How to read an FCN-score. Generated street photos are fed to a segmentation network (FCN-8s) trained on real Cityscapes photos, and its predicted labels are compared with the label map the photo was generated from. If the photo is realistic *and* faithful to its input, the

Table 6.9. Which loss terms matter: FCN-scores for Cityscapes labels→photo, from Table 1 of Isola et al. (2017). Higher is better. "GAN" uses a discriminator that does not see the input (Equation (6.45) with $D(\boldsymbol{y})$ in place of $D(\boldsymbol{x}, \boldsymbol{y})$).

Loss	Per-pixel acc.	Per-class acc.	Class IoU
L_1	0.42	0.15	0.11
GAN (unconditional)	0.22	0.05	0.01
cGAN	0.57	0.22	0.16
L_1 + GAN	0.64	0.20	0.15
L_1 + cGAN	**0.66**	**0.23**	**0.17**
Ground truth photos	0.80	0.26	0.21

segmenter recovers the input. Even real photos score only 0.80, because the segmenter itself is imperfect; that row is the ceiling. Two readings stand out. The unconditional GAN is by far the worst (0.22), just as the swapped generator example predicts: it produces realistic-looking photos that ignore the label map. And the full objective, L_1 + cGAN, is best on all three columns.

The generator: a U-Net

The problem it solves. In image-to-image translation the input and output share a great deal of low-level structure: the edges in an edge map are exactly where the object boundaries in the photo should be. A plain encoder–decoder squeezes everything through a tiny bottleneck and must rebuild those locations from scratch. A U-Net (Ronneberger et al., 2015) adds *skip connections* that copy each encoder layer's activations directly to the decoder layer of the same resolution, so location detail never has to pass through the bottleneck.

The architecture, layer by layer. Every layer is a 4×4 convolution with stride 2, followed by batch normalization and a ReLU (leaky, slope 0.2, in the encoder). "C64" means such a layer with 64 filters; "CD" adds 50% dropout. For a 256×256 RGB input:

Worked Example — why the plain encoder–decoder fails at batch size 1

The paper notes a trap that shows how much the skips carry. Batch normalization standardizes each channel using the mean and standard deviation over the batch *and* all spatial positions. At the bottleneck the map is 1×1. With a batch of one image, each channel therefore has exactly *one* value a, its mean is a, and

$$\frac{a - \mu}{\sigma + \epsilon} = \frac{a - a}{\epsilon} = 0.$$

Every bottleneck activation becomes zero, then the learned shift β is added, whatever the input. A plain encoder–decoder, whose only route from input to output runs through that bottleneck, now produces the same output for every input. The U-Net still works, because the skips route around the bottleneck. This is why the paper compares the two

Table 6.10. The Pix2Pix U-Net generator for a $256 \times 256 \times 3$ input, from the appendix of Isola et al. (2017). Each decoder layer's output is concatenated with the encoder activation of the same size (the "skip" column) before being passed on, which doubles the channels.

Layer	Type	Output size	Skip: concatenated with
e1	C64 (no batch norm)	$128 \times 128 \times 64$	
e2	C128	$64 \times 64 \times 128$	
e3	C256	$32 \times 32 \times 256$	
e4	C512	$16 \times 16 \times 512$	
e5	C512	$8 \times 8 \times 512$	
e6	C512	$4 \times 4 \times 512$	
e7	C512	$2 \times 2 \times 512$	
e8	C512 (bottleneck)	$1 \times 1 \times 512$	
d1	CD512 (upsampling)	$2 \times 2 \times 512$	e7 $\rightarrow 2 \times 2 \times 1024$
d2	CD512	$4 \times 4 \times 512$	e6 $\rightarrow 4 \times 4 \times 1024$
d3	CD512	$8 \times 8 \times 512$	e5 $\rightarrow 8 \times 8 \times 1024$
d4	C512	$16 \times 16 \times 512$	e4 $\rightarrow 16 \times 16 \times 1024$
d5	C256	$32 \times 32 \times 256$	e3 $\rightarrow 32 \times 32 \times 512$
d6	C128	$64 \times 64 \times 128$	e2 $\rightarrow 64 \times 64 \times 256$
d7	C64	$128 \times 128 \times 64$	e1 $\rightarrow 128 \times 128 \times 128$
out	conv + Tanh	$256 \times 256 \times 3$	

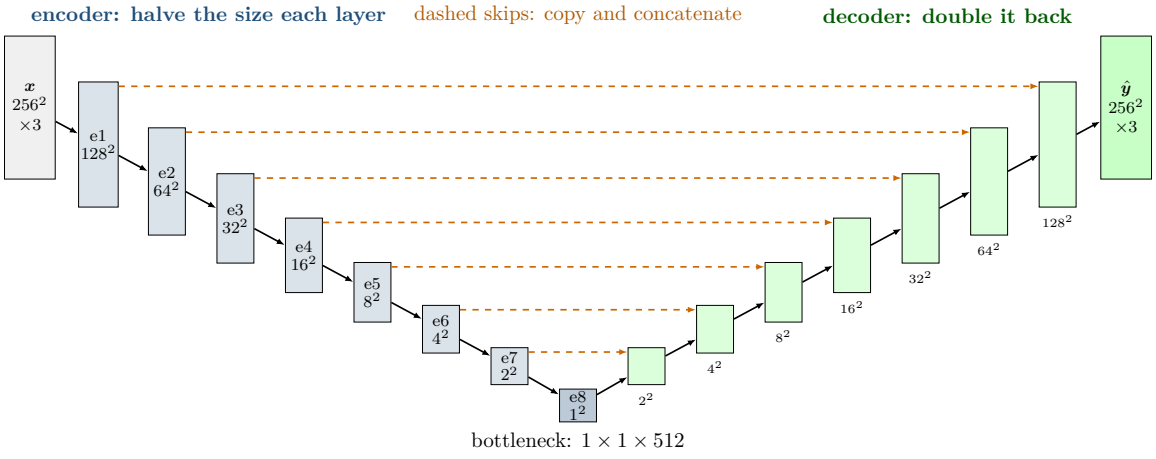


Figure 6.46. The Pix2Pix U-Net. Eight stride-2 convolutions shrink a 256×256 image to a $1 \times 1 \times 512$ bottleneck, and eight upsampling layers grow it back. The dashed skips hand each resolution's encoder features straight to the decoder layer of the same size. Remove them and you get the plain encoder–decoder of Table 6.11.

How to read it: Box heights shrink with resolution. Each dashed arrow joins two boxes of the same size: that pairing (layer i with layer $n - i$) is the whole U-Net idea.

architectures at batch size 10, and why it notes that removing batch norm from the bottleneck is an alternative fix.

Table 6.11. Generator architecture, FCN-scores on Cityscapes labels→photo, from Table 2 of Isola et al. (2017) (batch size 10, which is why the U-Net scores differ slightly from Table 6.9).

Generator (loss)	Per-pixel acc.	Per-class acc.	Class IoU
Encoder–decoder (L_1)	0.35	0.12	0.08
Encoder–decoder (L_1 + cGAN)	0.29	0.09	0.05
U-Net (L_1)	0.48	0.18	0.13
U-Net (L_1 + cGAN)	**0.55**	**0.20**	**0.14**

The U-Net wins with either loss, so its benefit is not specific to GANs. Notice also that adding the cGAN term made the encoder–decoder *worse*: an adversarial loss sharpens whatever structure the generator produces, and without skips that structure is poor.

Where is the noise? A conditional GAN normally takes noise $\boldsymbol{z}$ as well as $\boldsymbol{x}$, so that one input can map to many outputs. Pix2Pix tried this and found the generator simply learned to ignore $\boldsymbol{z}$. The final models instead get randomness only from dropout, kept on at test time, and the authors report that the outputs vary only slightly. So in practice Pix2Pix is close to deterministic: it learns a sharp, realistic *single* answer per input rather than the full conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$. The paper names this an open problem.

The discriminator: PatchGAN

The idea. Since L_1 already handles the low frequencies, the discriminator only needs to judge local texture and sharpness. So instead of one real/fake verdict for the whole image, a PatchGAN outputs a verdict for every $N \times N$ patch, and the loss averages them. The paper describes this as modelling the image as a Markov random field (pixels farther apart than a patch are treated as independent), which makes it a learned texture or style loss.

The architecture. The input is the pair stacked on the channel axis, $256 \times 256 \times 6$ (three channels of $\boldsymbol{x}$, three of $\boldsymbol{y}$ or $\hat{\boldsymbol{y}}$). The 70×70 version is C64–C128–C256–C512 followed by a one-channel convolution and a sigmoid, all 4×4 kernels. The paper's appendix says every convolution has stride 2; the authors' released code uses stride 2 for the first three and stride 1 for the last two, and that is the setting that gives exactly 70:

Worked Example — computing the receptive field

Work backwards from one output value. A layer with kernel k and stride s turns a receptive field of r (measured at its output) into

$$r_{\text{in}} = s\,(r_{\text{out}} - 1) + k$$

Table 6.12. The 70×70 PatchGAN on a 256×256 pair, with the strides of the released code (padding 1). "Receptive field" is the size of the input patch that one output value depends on.

Layer	Kernel, stride	Output size	Receptive field
input pair		$256 \times 256 \times 6$	
C64 (no batch norm)	4, 2	$128 \times 128 \times 64$	70
C128	4, 2	$64 \times 64 \times 128$	34
C256	4, 2	$32 \times 32 \times 256$	16
C512	4, 1	$31 \times 31 \times 512$	7
conv to 1 channel + sigmoid	4, 1	$30 \times 30 \times 1$	4

at its input. Start at $r = 1$ and apply the five layers from last to first:

$$1 \xrightarrow{k=4,\,s=1} 4 \xrightarrow{k=4,\,s=1} 7 \xrightarrow{k=4,\,s=2} 16 \xrightarrow{k=4,\,s=2} 34 \xrightarrow{k=4,\,s=2} 70.$$

So each of the $30 \times 30 = 900$ outputs looks at a 70×70 patch of the pair. The strides multiply to $2 \cdot 2 \cdot 2 = 8$, so neighbouring outputs' patches start 8 pixels apart and overlap heavily: every pixel is judged by many patches. The output size follows from the usual formula $\lfloor (n + 2p - k)/s \rfloor + 1$: $256 \to 128 \to 64 \to 32 \to 31 \to 30$.

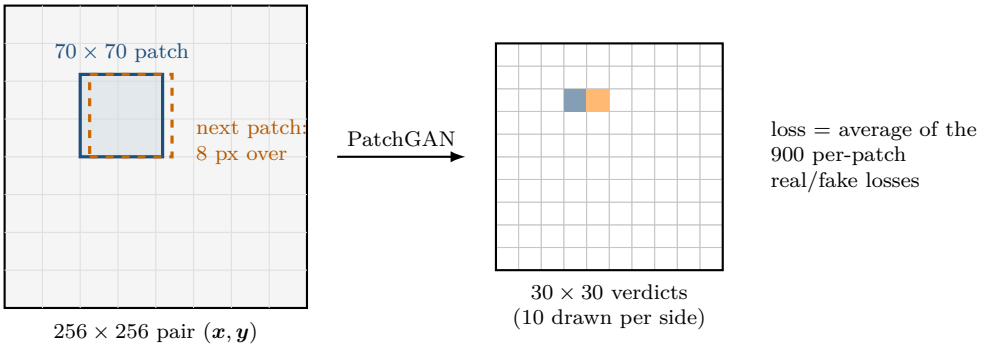


Figure 6.47. What a PatchGAN outputs. Each cell of the 30×30 output is a real/fake probability for one 70×70 patch of the input pair. Adjacent cells look at patches shifted by 8 pixels, so the patches overlap and every region of the image is judged many times.

How to read it: The blue patch produces the blue cell; the dashed orange patch, eight pixels to the right, produces the orange cell next to it. The whole grid is one convolutional pass, not 900 *separate runs.*

Why patches, and how big? A patch discriminator has fewer parameters, runs faster and can be applied to images of any size, since it is fully convolutional. The paper tested the patch size directly:

The pattern is not "bigger is better". A 1×1 discriminator can only judge colour, not structure. 16×16 already gives sharp results; the paper reports it also produces tiling artifacts, which 70×70 reduces. The full-image discriminator is worse, which the paper attributes to it having many more parameters and greater depth, making it harder to train.

Table 6.13. Discriminator receptive field, FCN-scores on Cityscapes labels→photo, from Table 3 of Isola et al. (2017). Inputs are 256 × 256; the 286 × 286 "ImageGAN" sees the whole image (with zero padding).

Receptive field	Per-pixel acc.	Per-class acc.	Class IoU
1 × 1 ("PixelGAN")	0.39	0.15	0.10
16 × 16	0.65	0.21	0.17
70 × 70	**0.66**	**0.23**	**0.17**
286 × 286 ("ImageGAN")	0.42	0.16	0.11

Training Pix2Pix in practice

Step by step — one iteration, as in the paper

1. Take a training pair $(\boldsymbol{x}, \boldsymbol{y})$. Apply random jitter (resize 256 → 286, random crop back to 256) and random mirroring.
2. Compute $\hat{\boldsymbol{y}} = G(\boldsymbol{x})$, with dropout on.
3. **Discriminator step.** Run D on $(\boldsymbol{x}, \boldsymbol{y})$ and on $(\boldsymbol{x}, \hat{\boldsymbol{y}})$. Its loss is the average binary cross-entropy of "real" on the first map and "fake" on the second, *divided by* 2 so D learns more slowly relative to G.
4. **Generator step.** Run D on $(\boldsymbol{x}, \hat{\boldsymbol{y}})$ again. The generator's loss is the non-saturating adversarial loss (make the fake map look real) plus $\lambda = 100$ times the L_1 distance to $\boldsymbol{y}$.

Adam with learning rate 0.0002 and $\beta_1 = 0.5$, $\beta_2 = 0.999$; weights initialized from $\mathcal{N}(0, 0.02^2)$; batch size between 1 and 10; typically 200 epochs.

Worked Example — one training step with numbers

Shrink everything so it fits on the page: the discriminator outputs a 2×2 map instead of 30×30, and the image has two pixels.

Discriminator. On the real pair it outputs $\left(\begin{smallmatrix} 0.9 & 0.8 \\ 0.7 & 0.95 \end{smallmatrix}\right)$, on the fake pair $\left(\begin{smallmatrix} 0.2 & 0.4 \\ 0.1 & 0.3 \end{smallmatrix}\right)$.

$$\begin{aligned} \text{real part} &= \tfrac{1}{4}(-\log 0.9 - \log 0.8 - \log 0.7 - \log 0.95) \\ &= \tfrac{1}{4}(0.105 + 0.223 + 0.357 + 0.051) \approx 0.184, \\ \text{fake part} &= \tfrac{1}{4}(-\log 0.8 - \log 0.6 - \log 0.9 - \log 0.7) \\ &= \tfrac{1}{4}(0.223 + 0.511 + 0.105 + 0.357) \approx 0.299, \end{aligned}$$

where the fake part scores $1 - D$, since "fake" is the right answer there, and $\mathcal{L}_D = \frac{1}{2}(0.184 + 0.299) \approx 0.242$. The cell that says 0.4 for a fake patch contributes the most: that patch was nearly convincing.

Generator. Adversarial part (non-saturating): $\frac{1}{4}(-\log 0.2 - \log 0.4 - \log 0.1 - \log 0.3) = \frac{1}{4}(1.609 + 0.916 + 2.303 + 1.204) \approx 1.508$. For L_1, suppose the target pixels are $\boldsymbol{y} =$

$(0.8, -0.2)$ and the output is $\hat{\boldsymbol{y}} = (0.5, 0.1)$; the mean absolute error is 0.3, and with $\lambda = 100$ it contributes 30. So $\mathcal{L}_G \approx 1.508 + 30 = 31.5$.

Early in training the L_1 term dominates, pulling the output towards the right layout. The adversarial term's gradient is smaller but points somewhere L_1 cannot: towards whatever makes the 0.1 patch (the least convincing one) look like real data.

At test time, the generator runs exactly as in training: dropout stays on, and batch normalization uses the statistics of the test batch rather than running averages. With a batch size of 1 that is the same as instance normalization. The paper also stresses how little data it needs: the facades model trains on 400 images in under two hours on one GPU, and all models run in well under a second per image.

What the results show, including where it does not help

The paper checks realism with human judges on Amazon Mechanical Turk, who see real and generated images and must say which is real:

Table 6.14. Percentage of generated images that human judges labelled as real, on the maps↔aerial photos task, from Isola et al. (2017). Higher means more convincing; 50% would mean indistinguishable.

Loss	Photo → map	Map → photo
L_1	$2.8\% \pm 1.0\%$	$0.8\% \pm 0.3\%$
L_1 + cGAN	$6.1\% \pm 1.3\%$	$18.9\% \pm 2.5\%$

The adversarial term helps far more in the map→photo direction, where the output (a photo) is rich and ambiguous, than in photo→map, where the output is simple. The paper makes the same point with the reverse Cityscapes task, photo→labels: there, plain L_1 regression beats the cGAN (per-pixel accuracy 0.86 against 0.83 for L_1 + cGAN). The authors' reading is that for vision problems, where the right output is less ambiguous than in graphics tasks, a reconstruction loss is mostly enough.

Key takeaway — Pix2Pix in four lines

(1) Condition the *discriminator* on the input, or the generator can ignore it. (2) Add L_1 for the layout; it alone gives the median, which blurs when the answer is ambiguous. (3) Use a U-Net so low-level detail skips the bottleneck. (4) Use a patch discriminator, because L_1 already covers the large scale and only local texture needs a critic. The adversarial term earns its keep exactly when the output is ambiguous.

The caveat: pairs are labels

But the lecture immediately inserts a *critical caveat in bold* on that results slide: **some of these tasks require labelled training data**. Pix2Pix learns a conditional distribution

from paired $(\boldsymbol{x}, \boldsymbol{y})$ examples. The pairs *are* labels—they are ground-truth mappings somebody had to collect. The lecture's taxonomy (used again in Chapter 12) therefore calls training on human-made pairs *strictly semi-supervised*: the method is not self-supervised whenever someone had to produce the pairs, even though nothing in its architecture needs labels. Only tasks whose input can be computed from the target (edges, greyscale, a blurred copy) keep it self-supervised. The natural counterpart—CycleGAN, which discards paired data and instead imposes cycle consistency on unpaired sets—is deferred to Chapter 12, where that story belongs. For now, hold both facts: conditional GANs are architecturally free, and that freedom is paid for in supervision requirements.

6.4.14 Why generative models matter beyond the course

Slides 41–42 step back: the generative approach is how one models *ambiguity*. When many outputs $\boldsymbol{x}$ plausibly correspond to one input $\boldsymbol{y}$, the right object to model is $p(\boldsymbol{x} \mid \boldsymbol{y})$, not a point estimate $\hat{\boldsymbol{x}}(\boldsymbol{y})$. The lecture picks two modern witnesses. Language modeling, where the input is a text prompt and the output is continuation text (the slide's prompt: "Write me a short rhyming poem about generative models"—the poem at the head of this chapter is what the model returned). Text-to-image, where the input is a caption and the output a photo. Both framings—conditional generative modeling—run through this chapter's taxonomy and through the rest of the book.

6.5 Taxonomy of Generative Models

The lecture's last slide (43) closes with a tree drawn originally by Ian Goodfellow (Tutorial on Generative Adversarial Networks, 2017) and adapted by CS231n: the space of generative models organized by whether $p(\boldsymbol{x})$ is computable and, if so, how, and if not, how sampling proceeds. We redraw it as Table 6.15, which is more compact and easier to extend as new families appear.

Two observations turn the taxonomy from taxonomy into analysis. First, the columns trade against each other: flows buy exact densities with architectural restriction; GANs buy architectural freedom by giving up densities; diffusion buys freedom and sample quality by giving up one-pass sampling. No family wins on all three. Second, the leaves are not isolated: the next chapter's latent diffusion is, in the lecture's words, a composition of VAEs, GANs, flows and diffusion, and Chapter 12's instruct-pix2pix successor (Brooks et al., 2023) grafts a conditional diffusion model onto Pix2Pix's image-to-image framing—the taxonomy's boxes begin to blend at the modern edge.

6.6 Summary

Modeling $p(X)$ from raw data is self-supervision: the data provides its own target, and the structure needed to assign density is the structure that transfers (slide 3's callback to MAE,

Table 6.15. Taxonomy of generative models, adapted to table form from the tree of Goodfellow's 2017 GAN tutorial as presented in CS231n (slide 43). "Exact" means the model's own $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ is computable in closed form. Position of diffusion matches the lecture's placement; Chapter 7 picks it up.

Family	Density $p(\boldsymbol{x})$	Sampling	Canonical example
Explicit, tractable	Exact, cheap	Sequential (D steps)	Autoregressive models (van den Oord et al., 2016)
Explicit, tractable	Exact, cheap	One pass	Normalizing flows (§6.3) (Dinh et al., 2017; Kingma and Dhariwal, 2018)
Explicit, approximate	Tractable bound only	One pass	Variational autoencoder (Ch. 4) (Kingma and Welling, 2014)
Implicit, direct	Not computable	One pass	GAN (§6.4) (Goodfellow et al., 2014)
Implicit, indirect	Not computable	Iterative	Diffusion models (Ch. 7) (Ho et al., 2020)

Chapter 5, holds for every method in this chapter). KDE is the simplest member—a Gaussian bump per training point—and its blurry digits (slide 8) show the distance between *a working sampler* and *a learned representation*. Normalizing flows close that distance by re-expressing change-of-variables as a neural architecture: bijectivity and triangular Jacobians buy exact likelihood (Real NVP, Glow), and adding conditions (Winkler et al.'s CNF) gives super-resolution that samples sharp detail instead of averaging it. Their price is dimensional and topological rigidity, plus a training brittleness the lecture flags with its DANGER box. GANs exit the likelihood game entirely, replacing a scalar density with a classifier adversary; their failure modes are game-theoretic (mode collapse, instability), their embodiment of *control* is StyleGAN's mapping network, AdaIN injection, style mixing, and noise-based stochastic detail, and their conditional form (Pix2Pix) is architecturally free but strictly semi-supervised because the pairs are labels. Each training pathology has a named fix (non-saturating and Wasserstein losses, R1 and spectral normalization, minibatch statistics, adaptive augmentation), and the StyleGAN line from ProGAN to StyleGAN3 shows the architecture side of the same story, one measured ablation at a time. Goodfellow's taxonomy frames the families by whether $p(\boldsymbol{x})$ is computable; Chapter 7 takes up the last empty leaf—diffusion.

Check Your Understanding

Level 1. 1. Write down the four-step generative recipe from slide 2, and identify what changes step (a) if the data are not i.i.d. 2. Compute $\hat{f}_h(\boldsymbol{q})$ by hand for the five-point example of Section 6.2.3 with $h = 2$, verifying that the answer lies between the $h = 1$ and $h = 5$ values reported there. 3. Explain in one sentence each why normalizing flows require (a) invertibility and (b) a tractable log-determinant.

Level 2. 4. Show that the inverse of the affine coupling layer (6.7) requires only element-wise subtraction and division, no matrix inversion. Where does s (resp. t) need to be evaluated during inversion, and why does it not need to be invertible itself? 5. In the coupling-layer worked example, compute the log-determinant $\log|\det J|$ for $s(x_1) = (0.2, -0.5)$ and check that it equals $\sum_j s_j$. 6. Why does Glow parameterize W via $PL(U + \mathrm{diag}(\boldsymbol{s}))$ rather than a learned permutation matrix? What happens to the gradient if P were learned?

Level 3. 7. Derive Equation (6.5) from Equation (6.4) and the chain rule for determinants under composition. Then use it to re-derive Equation (6.6) for a composition of T flows. 8. The GAN's fixed-G optimal discriminator is $D^*(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})/(p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x}))$. Derive it by maximizing Equation (6.23) pointwise in $D(\boldsymbol{x})$. 9. Show that substituting D^* into Equation (6.23) yields a function of p_G whose minimum over G is attained at $p_G = p_{\text{data}}$, and identify the divergence that appears.

Level 4. 10. Explain precisely why the saturating generator loss has vanishing gradients early in training, and show that Equation (6.26) has no such plateau by computing both gradients in a regime $D(G(\boldsymbol{z})) \approx \varepsilon$. 11. The text claims flows cannot model data on a lower-dimensional manifold inside $\mathbb{R}^d$ exactly. State the geometric/topological reason and sketch what pathologies arise when a flow tries anyway (think singular Jacobians, ill-conditioning). 12. StyleGAN's mapping network $f : Z \to W$ is described as freeing W from a "density-matching restriction." Make this precise: what constraint does drawing $\boldsymbol{z} \sim \mathcal{N}(0, I)$ impose on Z, and why is W exempt?

Level 5. 13. Pix2Pix is labeled "semi-supervised strictly" in this book's organization. Argue both sides: in what sense is paired image-to-image translation self-supervised, and in what sense is it supervised? Which of the lecture's own methods is the closest truly unpaired counterpart, and which chapter covers it? 14. Suppose a flow, a GAN, and a diffusion model each reach the same sample quality on CelebA. Using Table 6.15, rank them by (a) exactness of density evaluation, (b) sampling cost, (c) architectural freedom. Justify each ranking. 15. (Open ended.) The taxonomy treats the families as disjoint. Latent diffusion is described in Chapter 7 as "a composition of VAEs + GANs + flows + diffusion." Sketch how you would extend Table 6.15 to accommodate hybrid models without losing its discriminative power.

GAN training and the StyleGAN family. 16. (Level 2.) Apply AdaIN (Equation (6.31)) to the feature map $(2, 4, 6, 8)$ with style $y_s = 3$, $y_b = -1$. Then explain why the result would be the same for $(20, 40, 60, 80)$, and why that property led to StyleGAN2's droplet artifact. 17. (Level 2.) Modulate and demodulate (Equation (6.33)) the weights $\boldsymbol{w} = (1, 1, 1)$ with style $\boldsymbol{s} = (1, 2, 2)$. Check that the result's squared entries sum to 1. 18. (Level 3.) In the Dirac-GAN, the linearized simultaneous update is $(\theta, \psi) \mapsto (\theta - \frac{h}{2}\psi,\ \psi + \frac{h}{2}\theta)$. Show that it multiplies the distance to the origin by $\sqrt{1 + h^2/4}$. Then add the R1 term $-h\gamma\psi$ to the ψ update and show that for small $h > 0$ the distance now shrinks. 19. (Level 4.) A generator outputs perfect, confident images of exactly one

class per random seed, but only ever 10 of ImageNet's 1000 classes. Which of IS and FID detects the problem more directly, and why? (Use the formulas in Section 6.4.5.)

Conditional GANs and Pix2Pix. 20. (Level 2.) A PatchGAN uses 4×4 kernels with strides $2, 2, 1, 1$ (one fewer stride-2 layer than Table 6.12). Compute its receptive field with $r_{\text{in}} = s(r_{\text{out}} - 1) + k$, and its output size on a 256×256 input with padding 1. 21. (Level 3.) A pixel's true value is 0 with probability 0.3, 0.5 with probability 0.3, and 1 with probability 0.4. Find the output that minimizes the expected squared loss and the one that minimizes the expected absolute loss. Which of them is a value that actually occurs? 22. (Level 3.) Starting from Equation (6.35), derive Equations (6.36) and (6.37). Then show that if the discriminator does not see c, a generator with $p_G(\boldsymbol{x}) = p(\boldsymbol{x})$ reaches the optimum even when $p_G(\boldsymbol{x} \mid c) \neq p(\boldsymbol{x} \mid c)$. 23. (Level 2.) Show that joining a one-hot label $\boldsymbol{c}$ (class k) to a layer's input, $W[\boldsymbol{h}; \boldsymbol{c}]$, is the same as adding a class-specific bias. What does that imply about how much the label can change the layer's computation? 24. (Level 2.) A projection discriminator has $\phi(\boldsymbol{x}) = (3, -1)$, $\psi = -0.2$, $\boldsymbol{v}_{\text{cat}} = (0.5, 0.5)$ and $\boldsymbol{v}_{\text{dog}} = (-0.5, 1)$. Compute f and D for both labels.

Solution sketches

1. Samples → same distribution → fit $q(X)$ → sample from $q(X)$. Without i.i.d., step (a) needs explicit modeling of dependence (e.g. time, context).
2. Each kernel is $\frac{1}{8\pi} e^{-d^2/8}$. Contributions: $3 \cdot 0.0374 + 0.0227 + 0.0003 \approx 0.135$; divided by 5 gives $\hat{f}_2(\boldsymbol{q}) \approx 0.027$, between 0.078 ($h = 1$) and 0.0055 ($h = 5$).
3. (a) Invertibility enables both likelihood ($\boldsymbol{x} \to \boldsymbol{z}$) and sampling ($\boldsymbol{z} \to \boldsymbol{x}$). (b) Without a cheap log-det, Equation (6.5) is not computable at image dimension.
4. $\boldsymbol{x}_{d+1:D} = (\boldsymbol{y}_{d+1:D} - t(\boldsymbol{x}_{1:d})) \odot e^{-s(\boldsymbol{x}_{1:d})}$; s, t are evaluated at $\boldsymbol{x}_{1:d} = \boldsymbol{y}_{1:d}$, which is read off directly. They appear analytically, so their own invertibility is irrelevant.
5. $\log\det J = \log 1 + 0.2 + (-0.5) = -0.3$; equals $\sum_j s_j$.
6. P is discrete/permutation—non-differentiable; fixing it while learning $L, U, \boldsymbol{s}$ keeps the parameterization smooth and the determinant $\prod_i s_i$.
7. Take logs of (6.4); for composed maps use $\det(AB) = \det A \det B$ and the Jacobian chain rule $\prod_t J_{g_t}$; summing the per-layer log-determinants yields (6.6).
8. Pointwise, maximize $a \log D + b \log(1 - D)$, $a = p_{\text{data}}(\boldsymbol{x}), b = p_G(\boldsymbol{x})$; set derivative to zero: $D^* = a/(a + b)$.
9. Substitute and rearrange to get $-\log 4 + 2\,\text{JS}(p_{\text{data}} \| p_G)$; JS is minimized at 0 iff the distributions match.
10. Write $D = \sigma(l)$ with l the discriminator's logit and $D(G(\boldsymbol{z})) = \varepsilon$. Saturating: $\partial_l \log(1 - \sigma(l)) = -\sigma(l) = -\varepsilon$, so the gradient is $\approx -\varepsilon\, \partial l / \partial \theta_G$, nearly zero. Non-saturating: $\partial_l \log \sigma(l) = 1 - \varepsilon \approx 1$, so the gradient is $\approx \partial l / \partial \theta_G$ — about $1/\varepsilon$ times larger, exactly where the other vanished.
11. A bijection $\mathbb{R}^d \to \mathbb{R}^d$ preserves intrinsic dimension and topology; a Gaussian on $\mathbb{R}^d$ has full support. Fitting manifold-supported data forces the Jacobian towards singularity

(log-determinants heading to $\pm\infty$) to crush mass onto the manifold, steepening gradients and destabilizing training.

12. Z is drawn from a fixed Gaussian, so the push-forward of p_Z through the synthesis path must *match* the data density on any slice; W is an unnormalized learned space, only ever seen transformed by A/AdaIN, so it can organize itself to be convenient rather than density-matching.
13. Self-supervised when the input can be computed from the target (edges, greyscale, a downsampled copy), so the pairs build themselves; supervised when someone had to make the pairs (a hand-drawn segmentation map), which is why the lecture files it under "strictly semi-supervised". CycleGAN (Chapter 12) is the unpaired counterpart via cycle consistency.
14. (a) Flow > GAN $\approx$ diffusion. (b) GAN (one pass) < flow (one pass) < diffusion (iterative). (c) GAN $\approx$ diffusion > flow.
15. Add a third axis—*composition depth / hybridity*—or treat each leaf as a module and index models by which modules compose, preserving the density/sampling columns per module.
16. Mean 5, standard deviation $\sqrt{5} \approx 2.236$; normalized $(-1.342, -0.447, 0.447, 1.342)$; styled $(-5.025, -2.342, 0.342, 3.025)$. Scaling the input by 10 scales mean and standard deviation by 10, so the normalized map, and the output, are unchanged. The generator can therefore only control magnitudes by distorting the statistics themselves, which a localized spike does.
17. $\boldsymbol{w}' = (1, 2, 2)$; $\sqrt{1+4+4} = 3$; $\boldsymbol{w}'' = (1/3, 2/3, 2/3)$, and $1/9 + 4/9 + 4/9 = 1$.
18. The update matrix $\left(\begin{smallmatrix} 1 & -h/2 \\ h/2 & 1 \end{smallmatrix}\right)$ is a scaled rotation with determinant $1 + h^2/4$, so lengths grow by $\sqrt{1 + h^2/4}$. With R1 it becomes $\left(\begin{smallmatrix} 1 & -h/2 \\ h/2 & 1-h\gamma \end{smallmatrix}\right)$, whose eigenvalues are complex for small h with $|\lambda|^2 = \det = 1 - h\gamma + h^2/4 < 1$ once $h < 4\gamma$.
19. FID. IS rewards confident predictions and a broad *average* prediction; ten classes still give a moderately broad $p(y)$ and a high score, and IS never compares with real data. FID compares the feature statistics with real images, and missing 990 classes shifts the mean and badly shrinks the covariance.
20. Backwards from $r = 1$: $1 \to 4 \to 7 \to 16 \to 34$, so 34×34. Sizes: $256 \to 128 \to 64 \to 63 \to 62$, a 62×62 output map.
21. Mean $= 0.3 \cdot 0 + 0.3 \cdot 0.5 + 0.4 \cdot 1 = 0.55$, which never occurs. Median: $P(v \le 0.5) = 0.6 \ge 0.5$ and $P(v \ge 0.5) = 0.7 \ge 0.5$, so the median is 0.5, which does occur.
22. Write $V = \sum_c p(c) \int [p(\boldsymbol{x} \mid c) \log D + p_G(\boldsymbol{x} \mid c) \log(1 - D)]\, d\boldsymbol{x}$ and maximize the integrand pointwise as in Exercise 8; substituting gives one JS term per condition. Without c, the real images have density $p(\boldsymbol{x})$ and the fake ones $p_G(\boldsymbol{x}) = \sum_c p(c) p_G(\boldsymbol{x} \mid c)$, so $V = 2\,\mathrm{JS}(p(\boldsymbol{x}) \| p_G(\boldsymbol{x})) - 2\log 2$, which is minimal whenever the overall mixes match, as in the label-ignoring example.
23. $W_c \boldsymbol{c}$ picks out column k of W_c, so each class adds its own bias vector $\boldsymbol{b}_k$. A layer computing $\mathrm{ReLU}(W_h \boldsymbol{h} + \boldsymbol{b}_k)$ can shift which units are active per class, but the class

can interact with $\boldsymbol{h}$ only through that shift, so richer interactions must be built by later layers.

24. $f(\boldsymbol{x}, \text{cat}) = 0.5 \cdot 3 + 0.5 \cdot (-1) - 0.2 = 0.8$ and $f(\boldsymbol{x}, \text{dog}) = -0.5 \cdot 3 + 1 \cdot (-1) - 0.2 = -2.7$, so $D \approx 0.69$ and 0.06. The image is judged a plausible cat, and clearly not a dog; only the projection term differed.

Concept Map — how this chapter connects

Backward links. Chapter 4 introduced the VAE, whose stochastic encoder–decoder structure this chapter's flows replace with one bijective map (§6.3.1); the ELBO there is the tractable-bound entry in Table 6.15. Chapter 5's MAE is the lecture's canonical proof that generative pretraining yields transferable structure (§6.1). Chapter 2 supplied KDE's bandwidth trade-off and the Jacobian determinant. **Forward links.** Chapter 7 opens the implicit/indirect leaf with diffusion and continuous normalizing flows, closing the taxonomy loop. Chapter 8 discards density models in favor of pretext tasks—but keeps the same raw-data-only supervision. Chapter 12 resolves Pix2Pix's paired-data caveat with CycleGAN's unpaired, cycle-consistent formulation. Chapter 16 compares the generative, contrastive, and distillation families on the shared axes introduced here.

CHAPTER 7

Denoising, Diffusion, and Flow Matching

Rectified flow is a surprisingly simple approach to the transport mapping problem, which unifiedly solves generative modeling and domain transfer.

Liu, Gong, and Liu, Flow Straight and Fast *(Liu et al., 2023)*

In this chapter

7.1 The generative-model taxonomy, revisited 311

7.2 Denoising as a learning signal 313 • Corrupting is easy; coming back is learned • The denoising autoencoder: what it does • Why does this actually work? The score connection

7.3 Diffusion models as learned gradual denoising ... 319 • The principle • Where diffusion sits among VAEs, flows, and GANs • The reverse process: turning noise back into data • Why training is parallel and sampling is not • DDIM: fewer steps, the same marginals

7.4 DDPM under the hood: seven derivations the exercises rely on ..... 327 • Why the forward chain has a closed-form marginal • $\boldsymbol{\varepsilon}$-, $\boldsymbol{x}_0$-, and $\boldsymbol{v}$-parameterizations: the choice is numerics, not content • Tweedie's identity: the denoiser is a score estimator • The ELBO and the true likelihood are different curves • Deriving the DDPM objective, step by step • A DAE trained at one noise scale does not transfer to another • The timestep is a semantic-granularity dial

7.5 Rectified flow and flow matching, code first ..... 342 • Why we need all four: one problem, handed down • Setup: transport between two point clouds • Three definitions • Why regressing a random segment's velocity is correct • Sampling: Euler, backwards • What is "rectified" about it?

7.6 Continuous normalizing flows and the density question .. 357 • An ODE in disguise • The Hutchinson trick • Why not train the CNF by maximum likelihood?

7.7 Conditioning and classifier-free guidance 361 • Conditioning is one argument • Classifier-free guidance

7.8 Latent diffusion: composing everything 365

7.9 Autoregressive interlude: the tractable corner 367

7.10 Applications back in SSL territory 368 • DIFT: semantic correspondence falls out of diffusion training • Diffusion Policy: same machinery, deliberately *not* SSL

7.11 Closing the taxonomy loop 371

Chapter roadmap

Before you start. You need the ELBO and the reparameterization trick from Chapter 4 (Section 4.5, Section 4.6.2): diffusion's training loss is, underneath, one more instance of "sample noise, differentiate through the sampling step." You need the idea of a corruption-and-reconstruction target from Chapter 5's masked autoencoders (Chapter 5) — diffusion is that same idea with Gaussian noise instead of masking, applied at every corruption level at once. And you need the generative-model taxonomy and the normalizing-flow change-of-variables formula from Chapter 6 (Section 6.5, Section 6.3.1), because this chapter's ending is a one-sentence correction to that taxonomy. You do not need any prior exposure to stochastic differential equations or measure theory — everything here is built upward from a single squared-error loss.

The idea in one sentence. Corrupt a clean example by a known, controllable amount; train one network to undo a *small* amount of that corruption, at every corruption level; then chain the network's small guesses together, many steps at a time, to walk all the way from pure noise back to a plausible new sample.

An honesty note. This is the most mathematical chapter in the book — more derivations per page than any other. That is not padding: diffusion, score matching, and flow matching are three names for one underlying object, seen from three angles, and the connections *between* the angles are most of what makes the topic click. If this is a first pass, you can safely skip the closed-form-marginal induction (Section 7.4.1), the ELBO bookkeeping (Section 7.4.4), and the Hutchinson-trick algebra (Section 7.6) and still leave with a correct, usable picture of how diffusion and flow matching work. Come back to those three once you need to implement, debug, or examine one of these models yourself — a diagram with a "how to read it" line sits next to every one of them as a shortcut back to the intuition.

What we cover, and why each step is there.

- *Denoising as a learning signal* (Section 7.2): why undoing corruption is, by itself, enough supervision to force a network to learn real structure — and why that same mechanism turns out to be secretly computing a *score*, the direction of increasing data density.
- *Diffusion made visual* (Section 7.3): what actually happens to an image as noise is added and removed, and the single fact that resolves most confusion about diffusion — *training never simulates the chain.*
- *DDPM under the hood* (Section 7.4): the pieces of algebra that make the recipe precise — the closed-form noisy sample that makes training cheap, why three ways of parameterizing the network's output are secretly the same problem, and the identity that turns a denoiser into a score estimator for free.
- *Rectified flow / flow matching* (Section 7.5): the same idea, rebuilt from a straight line instead of a noise schedule, in about five lines of training code — and why straightness is exactly what lets sampling get away with far fewer steps.

- *Continuous normalizing flows* (Section 7.6): the precise sense in which a trained flow *is* a normalizing flow, closing the taxonomy loop opened in Chapter 6.
- *Guidance, latents, and two applications* (Sections 7.7–7.10): how to steer generation toward a condition without training a second network, how to make diffusion affordable at high resolution, and two places (DIFT, Diffusion Policy) where this exact denoising machinery does useful work outside of image generation.

Afterwards you will be able to write the training loop and the sampling loop for a diffusion or flow-matching model from memory and say what each line samples or computes; explain in one sentence why predicting the added noise and predicting the clean image are the same problem wearing different clothes; state precisely why DDPM needs hundreds of sampling steps while DDIM needs only tens, and why classifier-free guidance doubles sampling cost; and place diffusion correctly on the generative-model taxonomy, with a reason a classmate would accept.

Chapter 6 ended with a taxonomy: some generative models compute densities explicitly, others only let you sample, and denoising-based models sat awkwardly on the "implicit" side of that tree. This chapter shows that the awkwardness is an artifact of presentation. Taught the modern way, denoising diffusion is not a mysterious iterative sampler but a regression problem you can implement in five lines of PyTorch—and its sampling procedure is Euler integration of an ordinary differential equation (ODE) that *does* admit density evaluation. Diffusion, flow matching, and continuous normalizing flows turn out to be one connected idea, and the chapter closes by redrawing the taxonomy with that connection made explicit.

The lecture thread runs: denoising autoencoders (the bridge from Chapter 5's masked autoencoders) → diffusion as *learned gradual denoising* → rectified flow/flow matching as the primary, code-first formulation → the continuous normalizing flow that the flow is secretly defining → conditioning and classifier-free guidance → latent diffusion as the grand composition of everything in Part II so far → a short autoregressive interlude → two applications (DIFT, Diffusion Policy) that bring the discussion back to self-supervision.

7.1 The generative-model taxonomy, revisited

Where diffusion sits. Chapter 6 sorted generative models by one question: *can the model tell you the probability of an image, or can it only produce images?*

- **It can tell you the probability** (*explicit*): autoregressive models and flows exactly, VAEs approximately.
- **It can only produce images** (*implicit*): GANs, in one step — and diffusion models, in many small denoising steps.

Kernel density estimation was the simple warm-up: put a bump on every data point and add them up. It works in two dimensions and fails completely for images, because an image has

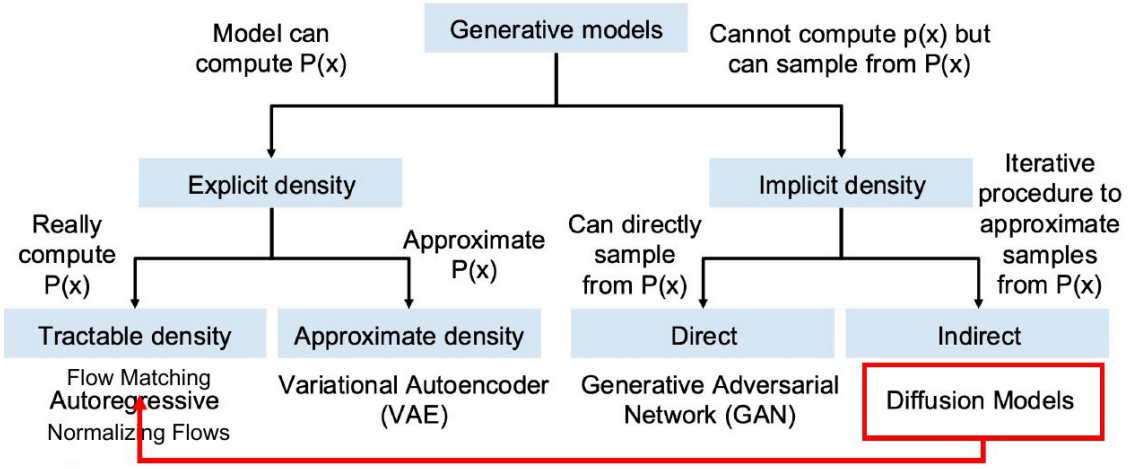


Figure 7.1. The taxonomy of generative models (Chapter 6), annotated on the lecture's closing slide (Lecture 05, slide 69): diffusion models were placed under implicit density; rectified flow / flow matching reveals them to be continuous normalizing flows—an explicit-density family (red arrow). Much of this chapter is the journey along that arrow.

hundreds of thousands of pixel values, and in that many dimensions every image is far from every other one. You would need astronomically many examples before the bumps overlapped at all. That failure is the reason deep generative models exist.

The normalizing-flow recall slide gives the pieces we will reuse in Section 7.6: compose invertible maps $\boldsymbol{x} = f_T \circ f_{T-1} \circ \cdots \circ f_1(\boldsymbol{z})$ with $\boldsymbol{z} \sim p_{\text{noise}}(\boldsymbol{z})$, and the change-of-variables formula yields a computable log-density:

$$\log p_{\boldsymbol{x}}(\boldsymbol{x}) \;=\; \log p_{\boldsymbol{z}}(\boldsymbol{z}_0) \;-\; \sum_{t=1}^{T} \log\left|\det \boldsymbol{J}_{f_t}(\boldsymbol{z}_{t-1})\right|, \tag{7.1}$$

where $\boldsymbol{z}_0 = \boldsymbol{z}$, $\boldsymbol{z}_t = f_t(\boldsymbol{z}_{t-1})$, $\boldsymbol{z}_T = \boldsymbol{x}$, and $\boldsymbol{J}_{f_t}$ is the Jacobian of step t. The minus sign says that a step which stretches space spreads the same probability mass more thinly, so the density drops. The point of this line: the stretch factors of a chain of steps *multiply*, so their logs *add* ($\log|\det(\boldsymbol{AB})| = \log|\det \boldsymbol{A}| + \log|\det \boldsymbol{B}|$). If every step has a cheap determinant, the whole chain does too. GANs, on the next slide, make the opposite choice: $\boldsymbol{x} = G(\boldsymbol{z})$ is easy to sample, but there is no formula for $p(\boldsymbol{x})$ at all — training matches the generator's output to the data by competition instead.

The GAN as counterfeiter and police officer. The lecture tells the GAN story as a parable. The generator G is a counterfeiter; the discriminator D is a police officer. Early fakes are easy to catch, and the feedback is rich: wrong paper, wrong ink. As the fakes improve, the officer has to notice finer and finer details.

At the end, the officer can only guess — 50/50. That is both the elegance and the weakness of the idea: learning only stops when the fakes are good, but at that point the feedback to

the counterfeiter has also dried up. The officer's best strategy against a given counterfeiter is $D^*(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})/(p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x}))$: "the fraction of the probability at $\boldsymbol{x}$ that comes from real data" (derived in Chapter 6).

Keep both trade-offs in mind. By the end of the chapter, diffusion will have migrated from the implicit side of the tree to a spot right beside the flows, and Eq. (7.1) will have a continuous-time counterpart. Figure 7.1 shows the destination in advance.

7.2 Denoising as a learning signal

7.2.1 Corrupting is easy; coming back is learned

The lecture starts with one animation (slides 7–9): a clean photo of a cat, an arrow labelled "easy" to a very noisy version of it, and an arrow back labelled "learn how to get back".

The two arrows are not equally hard. Adding noise is trivial. Removing it requires knowing what clean images look like — that fur has texture, that sky is smooth, that cat pixels form ears. And the training data supplies both ends for free: the noisy input and the clean target. So denoising is a complete learning signal with no labels at all.

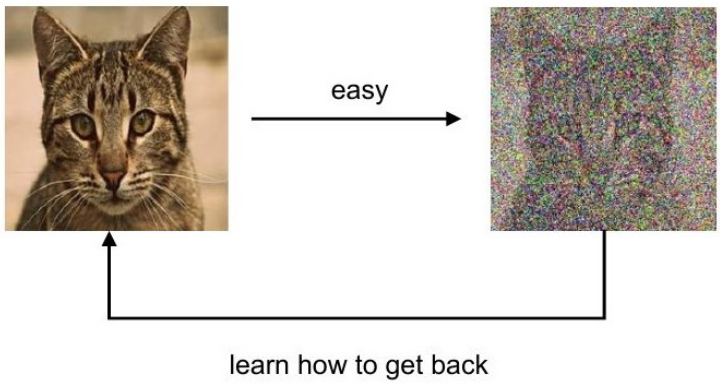


Figure 7.2. The idea the rest of this chapter elaborates, in one picture (Lecture 05, slides 7–9). Going right (clean → noisy) needs no learning at all: sample a random vector and add it. Going back up (noisy → clean) needs a model of what images look like, because infinitely many clean images could have produced the same noisy one — the arrow "learns" by being trained to guess a specific answer and being penalized for missing it.

How to read it: The two arrows are not the same kind of arrow. The top one is a fixed, parameter-free function (add noise). The bottom one is a learned function that must resolve an ambiguity the top one created.

Source: Lecture 05, slides 7–9.

Figure 7.2 is worth re-reading every time this chapter's equations get dense: whatever the current formula is computing, it is still just this picture — an easy corruption and a learned reversal.

7.2.2 The denoising autoencoder: what it does

What is it? A network with a bottleneck that is shown a corrupted image and scored on how well it reconstructs the *clean* version. It never sees the clean image as input — only as the target it is graded against.

Why was it introduced? A plain autoencoder (Chapter 4) can satisfy its own loss by learning the identity function: copy every pixel through the bottleneck, perfectly, without learning anything about images in general. If the bottleneck is wide enough, that shortcut is always available, and nothing forces the network to discover structure. Corrupting the input closes the shortcut: copying the corrupted pixels forward no longer scores well, because the target is the *uncorrupted* image, which the network never saw.

Key Concept — Denoising autoencoder

A *denoising autoencoder (DAE)* is trained to map a corrupted input $\tilde{\boldsymbol{x}}$ (e.g. $\boldsymbol{x}$ plus noise) back to the clean $\boldsymbol{x}$ (Vincent et al., 2008). Like the masked autoencoders of Chapter 5, it cannot copy its input: the encoder must compress abstract structure into the bottleneck $\boldsymbol{z}$, and the decoder retrieves $\boldsymbol{x}$ from the learned prior over images, not from $\tilde{\boldsymbol{x}}$.

Model at a glance: DAE — Vincent, Larochelle, Bengio & Manzagol, 2008

In one line	Corrupt the input with noise, keep the target clean, and force the bottleneck to hold whatever survives the corruption.
Problem it solves	A plain autoencoder can trivially satisfy its own reconstruction loss by copying, learning nothing. Denoising removes the copying shortcut.
Key intuition	The clean target is never available at the input, so the only way to score well is to know, in general, what clean images look like.
Inputs	A corrupted observation $\tilde{\boldsymbol{x}} \sim q(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ — Gaussian noise, salt-and-pepper noise, or masking all work.
Architecture	Encoder $f_{\boldsymbol{\theta}}$ to a bottleneck $\boldsymbol{z}$, decoder $g_{\boldsymbol{\theta}}$ back to image space. No constraint beyond the bottleneck; can be stacked layer by layer (below).
Loss	$\mathcal{L}_{\text{DAE}} = \mathbb{E}\big[\|g_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})) - \boldsymbol{x}\|^2\big]$ — plain squared error against the *clean* image.
Training	Ordinary SGD; one fixed corruption scale σ chosen in advance and held constant across the whole run.
What you keep	The bottleneck representation $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})$ as a feature for downstream tasks; or the whole network as a one-shot denoiser.
Strengths	No labels; simple squared-error training; the learned displacement $\hat{\boldsymbol{x}} - \tilde{\boldsymbol{x}}$ is provably a (local) score direction (Section 7.2.3).
Weaknesses	One network, one noise scale: shown a corruption larger than it was trained on, it under-corrects (Section 7.4.6). Cannot generate on its own — there is no principled way to sample a $\tilde{\boldsymbol{x}}$ to start from.
Leads to	Amortizing over *many* noise scales in one t-conditioned network turns a single DAE into a diffusion model (Section 7.3).

Step by step — training and running a denoising autoencoder

Step 1. Draw a clean example. Sample $\boldsymbol{x} \sim p_{\text{data}}$ from the training set, same as any supervised loss.

Step 2. Corrupt it. Sample $\tilde{\boldsymbol{x}} \sim q(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ — for Gaussian corruption, $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \sigma\boldsymbol{\varepsilon}$, $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, at one fixed σ decided ahead of time.

Step 3. Encode. Push $\tilde{\boldsymbol{x}}$ through $f_{\boldsymbol{\theta}}$ into the bottleneck $\boldsymbol{z}$. The bottleneck is narrower than the input, so something must be thrown away — the network's only choice is what.

Step 4. Decode. Push $\boldsymbol{z}$ through $g_{\boldsymbol{\theta}}$ to produce a reconstruction $\hat{\boldsymbol{x}}$.

Step 5. Compare to the clean target, not the input. Compute $\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|^2$ — critically, against $\boldsymbol{x}$, never against $\tilde{\boldsymbol{x}}$.

Step 6. Update. Backpropagate and take a gradient step on $\boldsymbol{\theta} = (\text{encoder}, \text{decoder})$ jointly.

Everyday analogy — a word blotted out by a coffee stain

A denoising autoencoder is like guessing a word blotted out by a coffee stain on a page. You do not have the ink, but the grammar and the meaning of the sentence around the stain constrain the missing word heavily — often to one obvious answer. Reconstructing it is not memory, it is using everything you know about how sentences are put together.

Where the analogy breaks. A blotted word usually has exactly one correct completion. A heavily noised image does not: many different clean images could plausibly have produced the same corrupted one. A network trained with squared error does not pick one of them — it outputs their *average*, which is why a DAE shown a large corruption tends to produce a plausible-looking but slightly blurred guess rather than a sharp, specific one. That averaging behavior resurfaces, with real consequences, in the latent-diffusion autoencoder of Section 7.8.

The geometric reading: a local projection back to the data manifold. A trained DAE implicitly defines a vector field on input space: at every point $\tilde{\boldsymbol{x}}$, the displacement $\hat{\boldsymbol{x}} - \tilde{\boldsymbol{x}}$ points from the corrupted observation back toward the nearest high-density region. For small corruptions the field behaves as a *local projection* onto the data manifold—largest exactly along the directions that leave the data support. Feed the same trained network an input corrupted at a much larger scale and the input lies outside the neighbourhood where that local field was learned, so the projection fails; this is precisely the failure mode the tutorial quiz probes, worked out in Section 7.4.6. The same vector-field view previews the score connection of Section 7.2.3: the arrows point uphill in log-density. Figure 7.3 shows both the plain DAE and the deeper, stacked variant that this local-projection reading justifies.

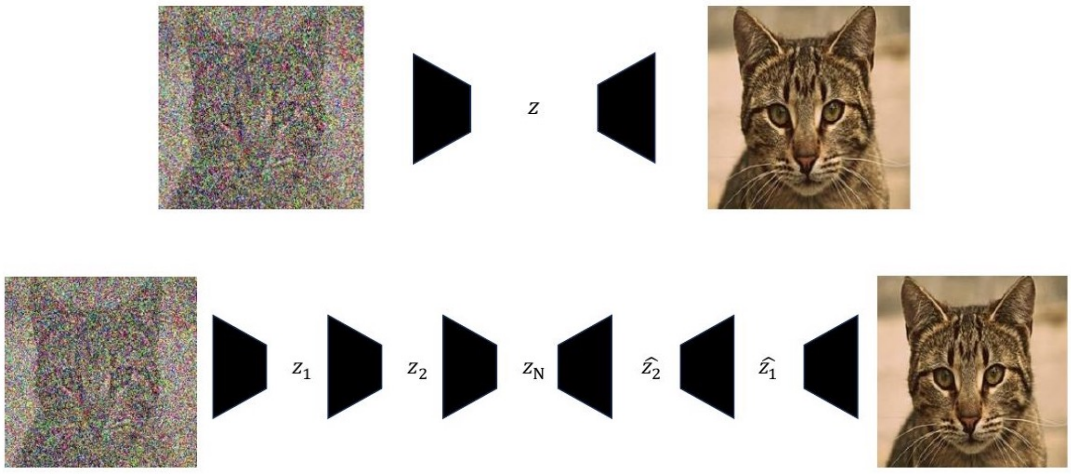


Figure 7.3. Top: a denoising autoencoder maps a noise-corrupted cat through a bottleneck $\boldsymbol{z}$ back to the clean cat (Lecture 05, slide 10). Bottom: the stacked variant, with encoder latents $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$, decoder latents $\hat{\boldsymbol{z}}_2, \hat{\boldsymbol{z}}_1$, and noise applied at each layer's input during layerwise pretraining (slide 11). Each layer is a shallow DAE whose features are then frozen before the next layer is trained on top of them.

How to read it: Count the bottlenecks. The top row has one: a single corrupt-encode-decode-clean pass. The bottom row has a chain *of them, $\boldsymbol{z}_1$ through $\boldsymbol{z}_N$ and back — each arrow is its own shallow DAE, trained and frozen before the next is added.*

Source: Lecture 05, slides 10–11.

In symbols: take a clean example $\boldsymbol{x}$, damage it to get $\tilde{\boldsymbol{x}}$ (Gaussian noise, salt-and-pepper noise, or masking), and train the network to recover the clean version:

$$\mathcal{L}_{\text{DAE}}(\boldsymbol{\theta}) \;=\; \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\, \mathbb{E}_{\tilde{\boldsymbol{x}}\sim q(\tilde{\boldsymbol{x}}|\boldsymbol{x})}\big[\, \| g_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})) - \boldsymbol{x}\|^2 \,\big],$$

where $f_{\boldsymbol{\theta}}$ is the encoder and $g_{\boldsymbol{\theta}}$ the decoder.

Two benefits follow (slide 10):

- **Robustness.** The features must ignore the damage. A plain autoencoder can get away with just copying its input; a denoising one cannot, because the input is not the answer.
- **More abstract features.** To repaint detail it never saw, the network has to capture what is *stable* about the input — its structure — rather than its exact pixels.

Stacking. The *stacked* denoising autoencoder of Vincent et al. (2010) (slide 11) builds a deep network one layer at a time:

1. Train one shallow denoising autoencoder on the input.
2. Freeze it.
3. Train the next layer as a denoising autoencoder on the frozen layer's outputs.
4. Repeat.

The slides add one practical note: adding noise to each stage's output is easier than masking hidden units. Before deep networks could be trained end to end, this was the standard way to get abstract features without labels.

What changed — from the plain autoencoder to the denoising autoencoder

Before (plain autoencoder, Chapter 4). Encoder and decoder are trained to reconstruct their own input. The loss is satisfied, in the limit, by memorizing the identity map, and nothing prevents the bottleneck from degenerating into "store the pixels more compactly" rather than "understand the image."

The change. Corrupt the input before the encoder sees it; keep scoring against the clean image. One line of the training loop moves: the target no longer equals the input.

What it cost. A choice that now has to be made and defended: how much corruption, of what kind. Too little and the identity shortcut reopens; too much and the target becomes unrecoverable from the input at all (Section 7.4.6 works out exactly this failure). That one new hyperparameter, and how to stop having to fix it in advance, is the thread the rest of the chapter pulls on.

Key takeaway

Denoising is a trick for removing a training shortcut, not a description of what the network is "for." By refusing to let the network see the clean image at its input, a squared-error loss on the bottleneck is forced to encode whatever general structure the corruption did not destroy. Everything that follows in this chapter is this same trick, applied at every corruption strength simultaneously instead of one fixed σ.

7.2.3 Why does this actually work? The score connection

Here is the bridge the book needs and the slides only gesture at. Consider a small Gaussian corruption, $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \sigma\boldsymbol{\varepsilon}$ with $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and small σ. Intuitively, the best denoiser moves $\tilde{\boldsymbol{x}}$ toward the nearest region where data actually lives—the data manifold.

In plain words — what a "score" is

Forget the gradient notation for a moment. At any point in image space, ask: "if I nudge this point slightly, which direction makes it look more like real data — more like something this dataset would actually produce?" The *score* is just that direction, written as a vector. It says nothing about *how likely* the current point is, only *which way is denser from here*. A denoiser that moves a noisy point toward its clean reconstruction is, whether or not it was ever told about scores, answering exactly that question.

More precisely, the best possible denoiser r satisfies

$$r(\tilde{\boldsymbol{x}}) - \tilde{\boldsymbol{x}} \;\propto\; \nabla_{\tilde{\boldsymbol{x}}} \log p(\tilde{\boldsymbol{x}}).$$

Read the left side as "the correction the denoiser applies" and the right side as "the direction in which the data gets more likely". They point the same way. This direction is called the *score*.

In plain words — the score, in one picture

Imagine the data distribution as a landscape of hills, tallest where images are most common. Drop a noisy image anywhere on it. The score is an arrow saying "uphill is this way". A trained denoiser has learned that arrow at every point — so denoising and climbing towards realistic images are the same act.

(That a network can learn this arrow without ever knowing the height of the landscape is the content of score matching (Hyvärinen, 2005).) Hold on to this picture: in Section 7.5 the velocity $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$ plays the same role, and Section 7.4.3 makes the link exact.

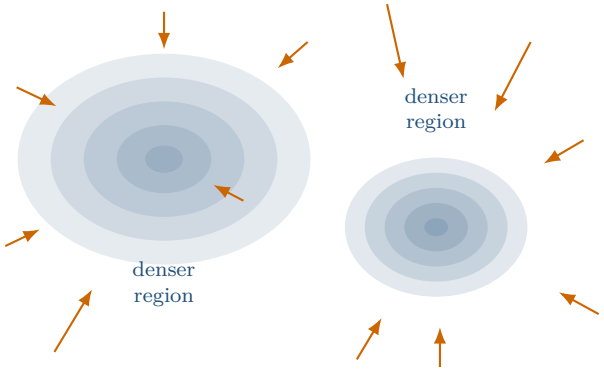


Figure 7.4. The score $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$ as a vector field. Each orange arrow marks a point in space and the direction that increases data density fastest from there — "which way is denser from here." Two density regions are shown (translucent blue); arrows on either side point toward whichever region is nearer, and the field is zero exactly at a mode, where there is no "uphill" left.

How to read it: Pick any arrow and imagine standing at its tail. The arrow is not telling you how likely your current position is — it is telling you which step would make you more typical of the data. A denoiser trained by squared error learns to take exactly that step, scaled by the noise variance (Section 7.4.3).

Key takeaway

A denoising autoencoder was never only a compressor. Its output minus its input is, up to a known scale factor, the score of the corrupted data density — the local "which way is denser" direction. Every later section that mentions a score (Tweedie's identity, the score-based SDE view, the velocity field of rectified flow) is naming the same object this section just derived from nothing but a squared-error loss.

7.3 Diffusion models as learned gradual denoising

7.3.1 The principle

One denoising step is useful; a *schedule* of many small denoising steps is generative (slides 12–17). The intuition the lecture pushes hardest is a physical one: adding small amounts of noise over many steps destroys structure the way waves flatten a sandcastle—each wave removes a few grains, irreversibly in accumulation but easy to describe locally. Rebuilding in one jump is hopeless; rebuilding grain by grain, reversing one wave at a time, is a learnable local correction. Diffusion models are the learned reversal of those waves. The recipe:

1. Pick a noise distribution $\boldsymbol{z} \sim p_{\text{noise}}$, usually the unit Gaussian $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$.
2. Consider data $\boldsymbol{x}$ corrupted under varying noise levels $t \in [0, 1]$, giving noisy data $\boldsymbol{x}_t$; $t = 0$ is the clean cat, $t = 1$ is pure noise (the vertical morphing strip on the slides).
3. Train a neural network $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ to remove a *little bit* of noise, conditioned on the level t.
4. At inference time, sample $\boldsymbol{x}_1 \sim p_{\text{noise}}$ and apply $f_{\boldsymbol{\theta}}$ many times in sequence to generate a noiseless sample $\boldsymbol{x}_0$.

Key Concept — Diffusion model

A *diffusion model* is a generative model that factorizes the hard problem "turn noise into data" into many easy problems "remove a little noise at level t," all handled by one shared, t-conditioned network trained with a supervised regression loss. Generation is the sequential composition of the learned denoising steps.

Figure 7.5 shows the picture behind the recipe above: one shared network, trained to predict a noise (or velocity) vector at every point along the corruption continuum.

DDPM in four sentences. The standard reference model is DDPM, the *denoising diffusion probabilistic model* of Ho et al. (2020). It fixes a *forward* process that adds a little Gaussian noise at each of many steps, until the image is pure noise. It then trains a network to undo one step at a time — and, after some algebra, the network's job reduces to *guessing which noise was added*, written $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$. So a DDPM is the denoising autoencoder of Section 7.2, trained at every noise level at once instead of just one.

Model at a glance: DDPM — Ho, Jain & Abbeel, 2020

In one line	Fix a many-step Gaussian noising chain analytically; train one time-conditioned network to undo one step of it; generate by running the learned reverse chain from pure noise.

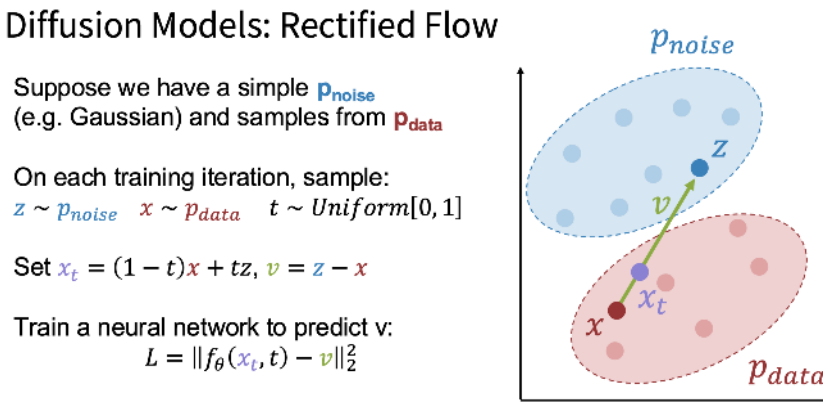


Figure 7.5. Diffusion models as learned denoising (Lecture 05, slides 12 and 34): pick a noise distribution $\boldsymbol{z} \sim p_{\text{noise}}$, interpolate $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$, and train one shared, t-conditioned network to predict the noise / velocity with a squared loss $\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{v}\|_2^2$—training across a continuum of noise levels, so that sampling becomes the gradual, iterated undoing of the corruption.

How to read it: The right-hand panel is the whole idea compressed: a point $\boldsymbol{x}_t$ sits between the data blob and the noise blob, and the network's only job is to say which way it came from.

Problem it solves	A single-scale denoising autoencoder (Section 7.2) only ever learns to remove *one* amount of noise, and has no way to generate on its own. DDPM amortizes one network over every noise level and gives it a principled training objective (a VAE-style ELBO over the whole chain) together with a concrete sampling procedure.
Key intuition	A single jump from noise to data is intractable to learn directly; a long sequence of small, locally easy jumps is not.
Inputs	A noisy sample $\boldsymbol{x}_t$ and its integer timestep $t \in \{1, \dots, T\}$, with T typically ≈ 1000.
Loss	After simplifying the ELBO, a noise-prediction mean squared error, $\mathbb{E}_{t,\boldsymbol{x}_0,\boldsymbol{\varepsilon}}\left[\|\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{\varepsilon}\|^2\right]$, with $\boldsymbol{x}_t$ built from $(\boldsymbol{x}_0, \boldsymbol{\varepsilon})$ by Eq. (7.4); derived in Section 7.4.5.
Training	One forward pass and one gradient step at a single, uniformly random t — the chain itself is never simulated (see the box below).
Sampling	Start from $\boldsymbol{x}_T \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and apply the learned reverse transition T times in sequence, one network call per step (Section 7.3.3).
What you keep	A network mapping (noisy input, timestep) to a noise estimate at every scale — reused unchanged by the representation-learning applications of Section 7.10.
Strengths	Stable regression training, no adversarial game; strong sample quality; a rigorous probabilistic derivation.
Weaknesses	Sampling needs hundreds to a thousand sequential network calls; no tractable exact likelihood in practice.
Leads to	DDIM, which keeps the same forward marginals but samples in tens of steps (Section 7.3.5); rectified flow / flow matching (Section 7.5), which replaces the Gaussian chain with a straight-line interpolant entirely.

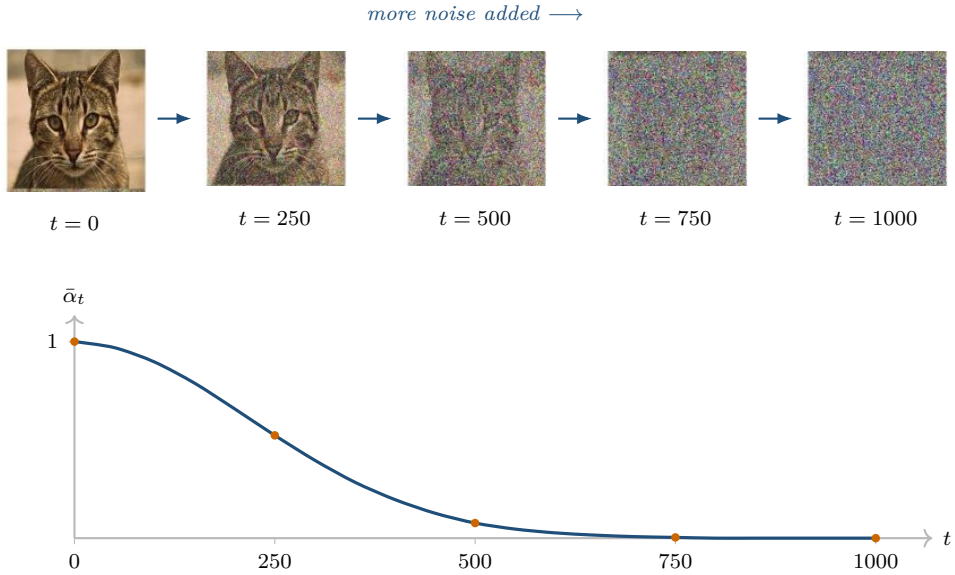


Figure 7.6. The forward process on one image, under the standard linear DDPM schedule (Ho et al., 2020) (β_1=10^{-4} to β_T=0.02, T=1000). Top: the same photo corrupted by Eq. (7.4) at five timesteps. Bottom: the corresponding value of $\bar{\alpha}_t$, the fraction of the original signal's *variance* still present at that t — marked at the same five points.

How to read it: Match a dot on the curve to the photo above it. By $t = 500$, $\bar{\alpha}_t \approx 0.08$: only 8% of the signal's variance survives, which is already most of the way to pure noise — the schedule spends its first few hundred steps destroying most of the image, then a long tail polishing off what little structure is left.

Step by step — one DDPM training step — and only one

Step 1. Sample an image. Draw $\boldsymbol{x}_0 \sim p_{\text{data}}$ from the training set, exactly as for any supervised loss.

Step 2. Sample a timestep. Draw $t \sim \text{Uniform}\{1, \dots, T\}$ — one random noise level, independent of the image just drawn.

Step 3. Sample noise. Draw $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, the same shape as $\boldsymbol{x}_0$.

Step 4. Build $\boldsymbol{x}_t$ in one shot. Compute $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}$ directly from Eq. (7.4) — one weighted sum, not a loop over steps $1, \dots, t$.

Step 5. Ask the network to name the noise. Compute $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$: the network's guess at which noise vector was mixed in.

Step 6. Take a gradient step on $\|\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{\varepsilon}\|^2$, then go back to step 1 with a fresh image, timestep, and noise draw.

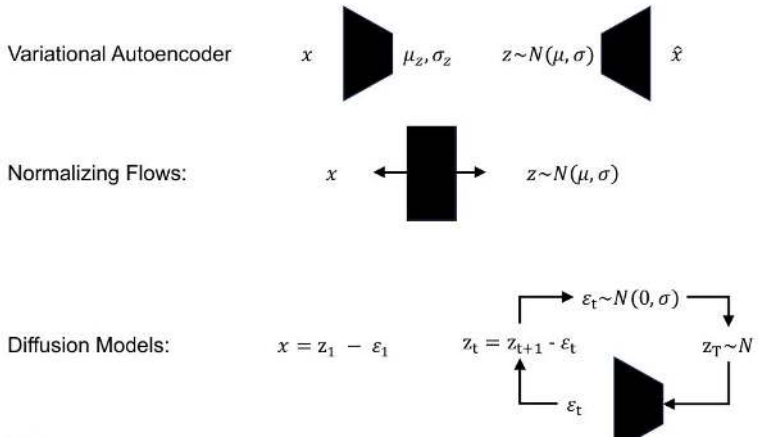


Figure 7.7. Three views of the same generative-model problem (Lecture 05, slide 18). VAE: encode to $(\boldsymbol{\mu_z}, \boldsymbol{\sigma_z})$, sample $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu_z}, \boldsymbol{\Sigma_z})$, decode. Flow: a single invertible block maps $\boldsymbol{x} \leftrightarrow \boldsymbol{z}$. Diffusion: an unrolled chain $\boldsymbol{z}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ with per-step noise subtraction, $\boldsymbol{z}_t = \boldsymbol{z}_{t+1} - \boldsymbol{\varepsilon}_t$, each $\boldsymbol{\varepsilon}_t$ predicted by the network.

Common Confusion

"So training walks the image through t noising steps before computing the loss?" No — and this is the single most common misunderstanding about diffusion models. Step 4 above is not a loop. The closed-form marginal of Eq. (7.4), proved in Section 7.4.1, lets training jump directly from $(\boldsymbol{x}_0, t, \boldsymbol{\varepsilon})$ to $\boldsymbol{x}_t$ in one weighted sum, at whatever t happened to be sampled — computing $\boldsymbol{x}_3$ costs exactly the same as computing $\boldsymbol{x}_{997}$, and a training batch mixes many different t values with no extra cost. The chain is simulated only at *sampling* time, when there is no $\boldsymbol{x}_0$ left to build a shortcut from and the network's own previous output must be fed back in as the next input, one step at a time (Section 7.3.3). Training is an ordinary single-step regression problem at a random noise level; only sampling is sequential.

The loss in step 6 is not a heuristic: it is derived from the data likelihood, step by step, in Section 7.4.5. The lecture itself moves on to rectified flow, which Section 7.5 develops from scratch. What matters here is the concept-level claim: diffusion training is denoising training distributed across a continuum of noise levels, and diffusion sampling is the gradual, iterated undoing of the corruption.

7.3.2 Where diffusion sits among VAEs, flows, and GANs

VAE, flow, diffusion: a packing analogy. Slide 18 (Figure 7.7) compares the three families. Imagine packing a fragile object into a box and unpacking it later:

- **VAE:** bubble-wrap it and squeeze it into a small standard box. Quick — but some detail is lost for good at the squeeze.
- **Flow:** fold it with reversible origami. Nothing is lost, but only certain folds are allowed.

- **Diffusion:** take it apart grain by grain into dust, and learn how to put the dust back together. Slow, but able to recover remarkable detail.

Where diffusion sits between the others:

compared with	alike	different
VAE	Gaussian at one end, learned path back	the "encoder" (adding noise) is fixed, not learned
flow	many steps between Gaussian and data	steps need not be invertible; no determinants
GAN	you mainly get samples, not probabilities	training is plain regression, not a game

The price: a GAN or VAE generates an image in one network pass; a diffusion model needs T of them.

Key takeaway

Diffusion is not a fourth, unrelated family — it borrows one piece from each of the other three and changes what it pays for. From the VAE it keeps a Gaussian endpoint, but fixes the "encoder" instead of learning it. From the normalizing flow it keeps a multi-step map to a Gaussian, but drops the invertibility constraint, buying much more expressive steps at the cost of an explicit density. From the GAN it keeps "sampling is all you directly get," but replaces the minimax game with a plain regression. The bill for all three concessions arrives at generation time, as T sequential network calls.

7.3.3 The reverse process: turning noise back into data

Training (Section 7.3) produces exactly one object: a network $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ that guesses the noise in a single noisy input, at whatever t it is handed. *Sampling* is a separate procedure, built on top of that trained object, and it is where diffusion's sequential cost actually originates — nothing about it is free, and nothing about it happened during training.

Step by step — the reverse process, one full pass

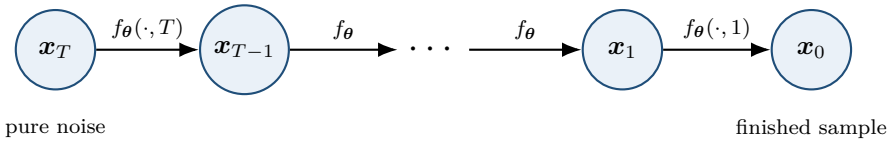
Step 1. Start from noise. Draw $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. No data was involved in producing this starting point — it is pure randomness.

Step 2. Ask the network what noise it sees. Compute $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_T, T)$.

Step 3. Remove a step's worth of it. Combine $\boldsymbol{x}_T$ and $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_T, T)$ through the reverse-transition formula (Eq. (7.17)) to obtain $\boldsymbol{x}_{T-1}$ — a small step back toward the data side, not a full jump to $\boldsymbol{x}_0$.

Step 4. Repeat, feeding each output back in as the next input: $\boldsymbol{x}_{T-1} \to \boldsymbol{x}_{T-2} \to \cdots \to \boldsymbol{x}_1 \to \boldsymbol{x}_0$, calling the *same* network with a smaller timestep each time.

Step 5. Stop at $\boldsymbol{x}_0$. After T calls, $\boldsymbol{x}_0$ is the finished sample; no further denoising is applied.



Every arrow calls the same *trained network at a different timestep — no new parameters appear between steps. Only the input changes, and each step's input is the previous step's output.*

Figure 7.8. The reverse (sampling) process: one trained network $f_{\boldsymbol{\theta}}$, applied T times in strict sequence, turning pure noise into a finished sample.

How to read it: Count the boxes, not the arrows, to see why this is slow: producing $\boldsymbol{x}_0$ needs T sequential *network calls, and call k cannot begin before call $k-1$ has finished and handed it an input.*

7.3.4 Why training is parallel and sampling is not

Figure 7.9 puts the two side by side, because mixing them up is the misunderstanding the box above warns about.

- **Training is a star.** Each example $(\boldsymbol{x}_t, t)$ is built in one jump and gives its own loss. None waits for any other, so a whole batch — with wildly different t values — runs in parallel.
- **Sampling is a chain.** Each step's input is the previous step's output. More GPUs can make each step faster, but step k still cannot start until step $k-1$ has finished.

Why DDPM needs about 1000 steps and DDIM only tens. DDPM's formula for undoing one step is an approximation that is only accurate for *small* steps — removing a little noise at a time. Take a few big steps instead and the approximation breaks, and the samples get visibly worse.

DDIM keeps the *same trained network* and changes only the sampling formula. Its formula comes from a deterministic process that passes through exactly the same intermediate distributions as DDPM, so it stays valid however far apart the steps are. It is the same reason rectified flow can take large steps (Section 7.5.6): a smooth, nearly straight path tolerates big strides.

7.3.5 DDIM: fewer steps, the same marginals

What is it? A different *sampling* rule for an *already-trained* DDPM network — no retraining, same weights, same loss. DDIM stands for *denoising diffusion implicit model* (Song et al., 2021a).

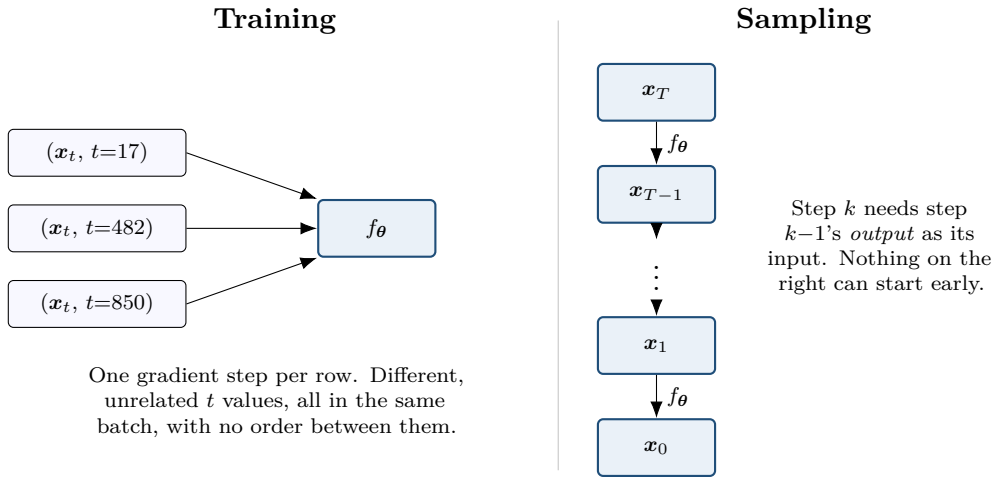


Figure 7.9. The training graph (left) is embarrassingly parallel across t; the sampling graph (right) is a strict sequential chain of length T. Both use the identical network $f_{\boldsymbol{\theta}}$ — only the *arrangement* of calls differs.

How to read it: This is the whole reason diffusion sampling is slow and training is not: training never has a chain to wait on, and sampling never has a batch to parallelize over.

Why was it introduced? DDPM's derivation ties the sampling formula to a specific Markov chain with T closely-spaced steps; naively evaluating it at only 50 of the original 1000 timesteps departs from the assumptions it was derived under, and quality drops sharply. Practitioners wanted DDPM's sample quality at a small fraction of its sampling cost.

Model at a glance: DDIM — Song, Meng & Ermon, 2021

In one line	Keep DDPM's trained network and its noisy-marginal distributions exactly as they are; replace only the *sampling* equation with one valid at any step size.
Problem it solves	DDPM sampling needs hundreds to a thousand sequential network calls; naively skipping steps in DDPM's own update rule degrades samples, because that rule assumes small steps.
Key intuition	Matching the marginals $q(\boldsymbol{x}_t \mid \boldsymbol{x}_0)$ does not pin down a unique reverse process — many reverse processes share the same marginals. Choose the member of that family that is deterministic and stays accurate at large step sizes.
Inputs	Identical to DDPM: a noisy $\boldsymbol{x}_t$ and its timestep t, fed to the *same* trained $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}$.
Loss	Unchanged from DDPM — DDIM is a sampling-time change only; a DDPM checkpoint can be sampled with DDIM with zero retraining.
Training	Not applicable; identical to DDPM.

Sampling	Deterministic update: use $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ to form the predicted clean image $\hat{\boldsymbol{x}}_0$ (Eq. (7.6)), then step to a *chosen* earlier timestep t' — not necessarily $t-1$ — by re-mixing $\hat{\boldsymbol{x}}_0$ and the predicted noise at t''s signal-to-noise ratio. Any subsequence of timesteps is valid, typically ∼50 evenly spaced ones.
What you keep	The exact DDPM weights; DDIM changes nothing about what was learned, only how it is called at inference.
Strengths	∼20× fewer network calls at comparable quality; deterministic given a fixed $\boldsymbol{x}_T$, which makes interpolation and image editing well defined (two nearby noise seeds decode to two similar images).
Weaknesses	Still tens of steps, not one; with very few steps it blurs or distorts samples like any coarse ODE solver. Because it is deterministic, all of its variety comes from the starting noise $\boldsymbol{x}_T$.
Leads to	Read as an ODE, DDIM's update is an Euler step of the *probability-flow ODE* discussed in Section 7.6 — the same object rectified flow (Section 7.5) builds directly, by choosing a path so straight that even fewer steps are needed.

What changed — from DDPM to DDIM

Before (DDPM). The sampling formula is derived under a small-step assumption baked into the Markov chain; it is only guaranteed accurate at (roughly) the same step spacing used during training's noise schedule.

The change. Re-derive a *different* reverse process — non-Markovian, and with a free parameter that can be set to zero variance — that produces the identical noisy marginals $q(\boldsymbol{x}_t \mid \boldsymbol{x}_0)$ DDPM already has, for any subsequence of timesteps.

What it cost. Nothing in training, and nothing in weights — the cost is conceptual: a DDIM sampler with very few steps is still an approximation to a continuous trajectory, and pushed too far it blurs or distorts samples just as a too-coarse Euler step does on any ODE (Section 7.5.6).

Key takeaway

DDIM is proof that "diffusion needs 1000 steps" was never a law, only a property of one particular sampling formula. The same trained network supports a whole family of samplers; DDPM and DDIM are two members of it, differing only in how large a step each is willing to trust.

7.4 DDPM under the hood: seven derivations the exercises rely on

This chapter taught diffusion the modern way, starting from rectified flow and only sketching DDPM. The lecture's quiz, however, asks questions that need the classical DDPM view, and one derivation is added here because everything else in the chapter uses its result:

1. How to get the noisy image at step t in one jump.
2. Why predicting the noise, the clean image, or the velocity are equivalent.
3. How denoising connects to the "score" (Tweedie's formula).
4. A common misunderstanding about the ELBO.
5. Where the "predict the noise" loss comes from: the DDPM objective, derived from the likelihood.
6. Why a denoiser trained at a single noise level fails.
7. Why the timestep t acts as a dial from coarse to fine.

Each is worked below with every step shown.

7.4.1 Why the forward chain has a closed-form marginal

The lecture defines diffusion's "learn how to get back" by composing many noising steps. The generic chain is

$$q(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \mid \boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}), \qquad q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) = \mathcal{N}\big(\boldsymbol{x}_t;\ \sqrt{1-\beta_t}\,\boldsymbol{x}_{t-1},\ \beta_t \boldsymbol{I}\big),$$

where $\beta_t \in (0,1)$ is the noise-variance *schedule* at step t. Writing $\alpha_t \equiv 1 - \beta_t$ and using the reparameterization trick (Section 4.6.2) ($\mathcal{N}(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I})$ can be sampled as $\boldsymbol{\mu} + \sigma\boldsymbol{\varepsilon}$, $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$), one step of the chain generates

$$\boldsymbol{x}_t = \sqrt{\alpha_t}\,\boldsymbol{x}_{t-1} + \sqrt{1-\alpha_t}\,\boldsymbol{\varepsilon}_t, \qquad \boldsymbol{\varepsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}). \tag{7.2}$$

Simulating Eq. (7.2) step by step is neither required nor cheap. Because a composition of Gaussians remains Gaussian, the marginal $q(\boldsymbol{x}_t \mid \boldsymbol{x}_0)$ is closed-form; deriving it is the standard induction of Ho et al. (2020). Start from Eq. (7.2),

$$\boldsymbol{x}_1 = \sqrt{\alpha_1}\,\boldsymbol{x}_0 + \sqrt{1-\alpha_1}\,\boldsymbol{\varepsilon}_1,$$

apply the step again,

$$\boldsymbol{x}_2 = \sqrt{\alpha_2}\,\boldsymbol{x}_1 + \sqrt{1-\alpha_2}\,\boldsymbol{\varepsilon}_2,$$

and substitute the expression for $\boldsymbol{x}_1$:

$$\begin{aligned}\boldsymbol{x}_2 &= \sqrt{\alpha_2}\big(\sqrt{\alpha_1}\,\boldsymbol{x}_0 + \sqrt{1-\alpha_1}\,\boldsymbol{\varepsilon}_1\big) + \sqrt{1-\alpha_2}\,\boldsymbol{\varepsilon}_2 \\ &= \sqrt{\alpha_1\alpha_2}\,\boldsymbol{x}_0 + \big(\underbrace{\sqrt{\alpha_2(1-\alpha_1)}\,\boldsymbol{\varepsilon}_1 + \sqrt{1-\alpha_2}\,\boldsymbol{\varepsilon}_2}_{\text{a sum of two independent Gaussians}}\big).\end{aligned}$$

Independent Gaussians add variances, so the bracketed term is one Gaussian with variance

$$\alpha_2(1-\alpha_1) + (1-\alpha_2) \;=\; \alpha_2 - \alpha_1\alpha_2 + 1 - \alpha_2 \;=\; 1 - \alpha_1\alpha_2.$$

Hence $\boldsymbol{x}_2 = \sqrt{\alpha_1\alpha_2}\,\boldsymbol{x}_0 + \sqrt{1-\alpha_1\alpha_2}\,\bar{\boldsymbol{\varepsilon}}_2$ with $\bar{\boldsymbol{\varepsilon}}_2 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, and continuing the recursion to step t produces the closed-form marginal

$$q(\boldsymbol{x}_t \mid \boldsymbol{x}_0) \;=\; \mathcal{N}\big(\boldsymbol{x}_t;\; \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0,\; (1-\bar{\alpha}_t)\boldsymbol{I}\big), \qquad \bar{\alpha}_t \equiv \prod_{s=1}^{t} \alpha_s. \tag{7.3}$$

Equivalently, in reparameterized form:

$$\boldsymbol{x}_t \;=\; \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 \;+\; \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}). \tag{7.4}$$

Equation (7.4) is the workhorse of the whole DDPM recipe: training samples a step t, draws one $\boldsymbol{\varepsilon}$, and builds $\boldsymbol{x}_t$ in $O(1)$ work instead of running t sequential chain steps — the exact "one weighted sum" step that Figure 7.6 and its accompanying training box already put to use.

Common Confusion

Same trick as the VAE — different reason. Equation (7.4) has exactly the VAE's form "mean + scale × standard noise" (Section 4.6.2), but it is used here for a different reason, and exams like to ask about it.

- **In a VAE** the trick is needed for *gradients*: the sampling distribution $q_{\boldsymbol{\phi}}$ has learnable parameters, and backprop must reach them through the sample.
- **In DDPM** the forward process q has *no* learnable parameters, so no gradient ever has to pass through it. The rewriting is used for *convenience*: it jumps straight from $\boldsymbol{x}_0$ to any $\boldsymbol{x}_t$ in one step, and it exposes the noise $\boldsymbol{\varepsilon}$ that the network is then trained to predict.

The variance-preservation check. Why those particular coefficients? Take data standardized so $\boldsymbol{x}_0 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ (which image preprocessing roughly enforces). Since $\boldsymbol{x}_0 \perp \boldsymbol{\varepsilon}$, variances add:

$$\mathrm{Var}(\boldsymbol{x}_t) \;=\; \big(\sqrt{\bar{\alpha}_t}\big)^2\,\mathrm{Var}(\boldsymbol{x}_0) \;+\; \big(\sqrt{1-\bar{\alpha}_t}\big)^2\,\mathrm{Var}(\boldsymbol{\varepsilon}) \;=\; \bar{\alpha}_t\,\boldsymbol{I} + (1-\bar{\alpha}_t)\,\boldsymbol{I} \;=\; \boldsymbol{I}.$$

The marginal variance stays exactly 1 at every step—hence a *variance-preserving (VP) schedule*: the total signal magnitude does not drift as noise is added; only the *fraction* of data and noise inside that fixed norm changes.

Information decay and the signal-to-noise ratio. Because every $\alpha_s \in (0,1)$, the cumulative product $\bar{\alpha}_t$ shrinks monotonically toward 0. The two ends of the schedule then read directly off Eq. (7.4):

- $t \approx 0$: $\bar{\alpha}_t \approx 1$, so $\boldsymbol{x}_t \approx \boldsymbol{x}_0$: the sample carries the *maximum* possible information about the clean image.
- $t \approx T$: $\bar{\alpha}_t \approx 0$, so $\boldsymbol{x}_t \approx \boldsymbol{\varepsilon}$: the sample has essentially been replaced by pure noise and carries the *least* possible information about $\boldsymbol{x}_0$.

The scalar statistic tracking that decay is the marginal's signal-to-noise ratio,

$$\mathrm{SNR}(t) \;=\; \frac{\bar{\alpha}_t}{1-\bar{\alpha}_t}, \tag{7.5}$$

which moves smoothly from "mostly data" ($\mathrm{SNR} \gg 1$) to "mostly noise" ($\mathrm{SNR} \ll 1$) as t grows. The flow-matching notation of Section 7.5 is a re-anchoring of the same quantity with time running the other way.

7.4.2 $\boldsymbol{\varepsilon}$-, $\boldsymbol{x}_0$-, and $\boldsymbol{v}$-parameterizations: the choice is numerics, not content

In plain words — why three "different" networks are secretly one

> $\boldsymbol{x}_t$ is built from exactly two ingredients — the clean image $\boldsymbol{x}_0$ and the noise $\boldsymbol{\varepsilon}$ mixed into it — combined with coefficients that are known numbers once t is fixed. So if a network can guess *either* ingredient, the *other* one follows by pure arithmetic, no extra learning required. "Predict the noise" and "predict the clean image" are not two different skills a network might have; they are the same skill, read off through two different rearrangements of one linear equation.

Equation (7.4) ties three variables at step t: the noisy observation $\boldsymbol{x}_t$, the clean endpoint $\boldsymbol{x}_0$, and the noise $\boldsymbol{\varepsilon}$ that hit it. A network $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ can be asked to predict any of the three: the noise $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$, the clean image $\hat{\boldsymbol{x}}_0 = f_{\boldsymbol{\theta}}^{(x_0)}(\boldsymbol{x}_t, t)$ directly, or a linear combination commonly called the *velocity* $\boldsymbol{v} = \sqrt{\bar{\alpha}_t}\,\boldsymbol{\varepsilon} - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{x}_0$, which matches the path-derivative form used by the flow-matching sections.

They carry exactly the same information. Because $\boldsymbol{x}_t$ is a fixed linear combination of $\boldsymbol{x}_0$ and $\boldsymbol{\varepsilon}$ with known coefficients, knowing any two of the three variables determines the third *exactly*, with no approximation. Rearranging Eq. (7.4) gives, for instance,

$$\hat{\boldsymbol{x}}_0 \;=\; \frac{\boldsymbol{x}_t - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)}{\sqrt{\bar{\alpha}_t}}. \tag{7.6}$$

Which parameterization is "correct" is therefore never the question—they are linear re-expressions of one another. What matters is *numerical conditioning*.

Where the error amplification comes from. Suppose an $\boldsymbol{\varepsilon}$-predicting network is off by a small vector $\boldsymbol{\delta}$ at step t: $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}} = \boldsymbol{\varepsilon} + \boldsymbol{\delta}$. Substituting into Eq. (7.6), the induced error in clean-image space is

$$\hat{\boldsymbol{x}}_0 - \boldsymbol{x}_0 \;=\; -\frac{\sqrt{1-\bar{\alpha}_t}}{\sqrt{\bar{\alpha}_t}}\,\boldsymbol{\delta},$$

whose magnitude carries the factor $1/\sqrt{\bar{\alpha}_t}$. At large t, as $\bar{\alpha}_t \to 0$, this factor explodes: a modest noise-prediction error becomes a huge clean-image error exactly where generation begins. The mirror-image statement holds at train time for the x_0-predicting objective $\|\boldsymbol{x}_0 - f_{\boldsymbol{\theta}}^{(x_0)}(\boldsymbol{x}_t, t)\|^2$: its gradient with respect to the underlying noise-space residual scales with $1/\sqrt{\bar{\alpha}_t}$, so late-time updates are noisy and unstable. That is the concrete reason the DDPM experiments of Ho et al. (2020), and most successors, predict $\boldsymbol{\varepsilon}$ (or $\boldsymbol{v}$, which averages the two ends) rather than $\boldsymbol{x}_0$.

Worked Example — the same numbers, two ways round

Pick a timestep where $\bar{\alpha}_t = 0.64$, so $\sqrt{\bar{\alpha}_t} = 0.8$ and $\sqrt{1-\bar{\alpha}_t} = 0.6$ (exact: $0.8^2 + 0.6^2 = 1$). Let the clean "image" be the single number $x_0 = 5$ and the noise $\varepsilon = 1$. Build the noisy sample: $x_t = 0.8 \times 5 + 0.6 \times 1 = 4.6$.

Both recoveries are exact. Given $(x_t, \varepsilon) = (4.6, 1)$, recover $x_0 = (x_t - 0.6\varepsilon)/0.8 = (4.6{-}0.6)/0.8 = 5$. Given $(x_t, x_0) = (4.6, 5)$, recover $\varepsilon = (x_t{-}0.8x_0)/0.6 = (4.6{-}4.0)/0.6 = 1$. Either direction lands exactly back where we started — "carries exactly the same information" is not just a slogan, it is this arithmetic.

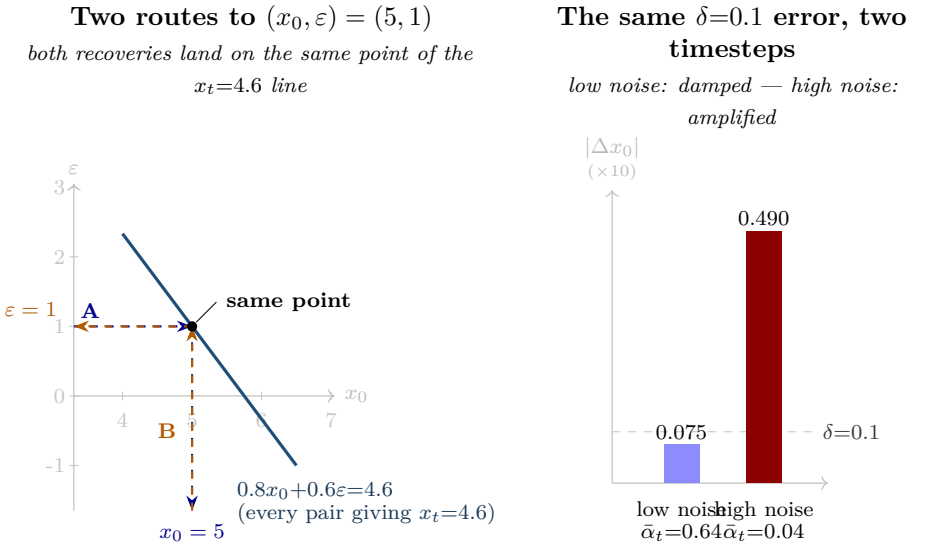
Now break the noise guess slightly and watch the two regimes. Suppose the network's noise prediction is off by $\delta = 0.1$: $\hat{\varepsilon} = 1.1$ instead of 1. *Low noise* ($\bar{\alpha}_t = 0.64$, mostly signal — near the *end* of generation):

$$\hat{x}_0 = \frac{4.6 - 0.6 \times 1.1}{0.8} = \frac{3.94}{0.8} = 4.925, \qquad \text{error} = -0.075 = -\tfrac{0.6}{0.8}\,\delta.$$

A 0.1 error in the predicted noise becomes a *smaller*, 0.075 error in the image — damped. *High noise* ($\bar{\alpha}_t = 0.04$, mostly noise — near the *start* of generation, so $\sqrt{\bar{\alpha}_t} = 0.2$, $\sqrt{1-\bar{\alpha}_t} \approx 0.980$), the identical $\delta = 0.1$ noise error gives

$$\text{error} \;\approx\; -\frac{0.980}{0.2}\,(0.1) \;\approx\; -0.490,$$

nearly five times the original noise error, and about 6.5× larger than the low-noise error above for the *exact same mistake*. This is Eq. (7.6)'s $1/\sqrt{\bar{\alpha}_t}$ factor made concrete: identical network error, nearly harmless at low noise, sharply amplified once $\bar{\alpha}_t$ is small — exactly where sampling starts. Every division above is exact arithmetic and checks in one line of `python3 -c`. Figure 7.10 draws both routes and both timesteps.



Given (x_t, ε), route A (blue) reads off x_0; given (x_t, x_0), route B (orange) reads off ε. Different starting knowledge, different arithmetic, identical destination.

The same network mistake $\delta = 0.1$ in ε costs 6.5× more image-space error at high noise (where sampling starts) than at low noise.

Figure 7.10. Left: the (x_0, ε) plane, with the diagonal line holding every pair consistent with the observed $x_t = 4.6$. Route A starts from ε and reads x_0 off the line; route B starts from x_0 and reads ε off the same line. They are different computations from different known quantities, and they land on the same black dot — "carries exactly the same information" made visible. Right: the induced image-space error from Eq. (7.6) for the identical noise-prediction mistake $\delta = 0.1$, at an early and a late timestep; bar heights are the exact values from the worked example, scaled ×10 for legibility.

How to read it: Trace route A and route B separately in the left panel — they start from different known quantities and travel different dashed paths — and confirm they stop on the same dot. That is the whole content of "two ways round": not similarity, identity. Then compare the two bars on the right: the same $\delta = 0.1$ mistake in ε-space is almost free at low noise and expensive at high noise, which is why the network is trained to predict ε and not x_0.

7.4.3 Tweedie's identity: the denoiser is a score estimator

The slide that causes the most confusion in this lecture is the one saying "denoising and the score are the same thing." The precise link is Tweedie's identity for Gaussian corruption: if $\boldsymbol{y} \mid \boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{y};\ \boldsymbol{x}_0,\ \sigma^2 \boldsymbol{I})$, then the posterior mean of the clean variable given the noisy observation is

$$\mathbb{E}[\boldsymbol{x}_0 \mid \boldsymbol{y}] \ = \ \boldsymbol{y} \ + \ \sigma^2 \nabla_{\boldsymbol{y}} \log p(\boldsymbol{y}),$$

with $p(\boldsymbol{y})$ the marginal density of the noisy observation.

Translate to the notation of this chapter. The noisy observation is $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\, \boldsymbol{x}_0 + \sigma_t \boldsymbol{\varepsilon}$ with $\sigma_t^2 = 1 - \bar{\alpha}_t$ (Eq. (7.3)), so the "clean variable" in Tweedie's identity is the *scaled* image $\sqrt{\bar{\alpha}_t}\, \boldsymbol{x}_0$. The optimal denoiser—the target that least-squares training converges to—is the conditional mean $D(\boldsymbol{x}_t) \equiv \mathbb{E}[\boldsymbol{x}_0 \mid \boldsymbol{x}_t]$, and the identity becomes

$$\sqrt{\bar{\alpha}_t}\, D(\boldsymbol{x}_t) \ = \ \boldsymbol{x}_t \ + \ \sigma_t^2 \nabla_{\boldsymbol{x}_t} \log p_t(\boldsymbol{x}_t), \qquad \Longleftrightarrow \qquad \nabla_{\boldsymbol{x}_t} \log p_t(\boldsymbol{x}_t) \ = \ \frac{\sqrt{\bar{\alpha}_t}\, D(\boldsymbol{x}_t) - \boldsymbol{x}_t}{\sigma_t^2}. \tag{7.7}$$

In noise-prediction terms this is simply $\nabla_{\boldsymbol{x}_t} \log p_t(\boldsymbol{x}_t) = -\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)/\sigma_t$: the predicted noise, flipped and rescaled, *is* the score.

Reading Tweedie's formula. The right-hand side is the arrow from the noisy point to the denoiser's cleaned-up guess (scaled by $\sqrt{\bar{\alpha}_t}$, the amount of signal the forward process kept), divided by σ_t^2. The left-hand side is the *score*: the direction in which the data density rises fastest — the arrows of Figure 7.4. The formula says these are the same arrow.

So training a Gaussian denoiser is not merely *similar* to learning the score — it *is* learning the score, exactly. That single identity is why the score-based SDE view (Song et al., 2021b) and DDPM (Ho et al., 2020) turned out to be two descriptions of one model.

Learning note from study sessions — Where the score "comes from"

The sticking point in the study session was that the score seemed to appear out of nowhere after a page of denoising. The resolution is that nothing new enters: the denoiser had to compute the posterior mean $\mathbb{E}[\boldsymbol{x}_0 \mid \boldsymbol{x}_t]$ all along (that is what the MSE optimum *is*), and Tweedie's identity only re-expresses that posterior mean as "noisy point plus σ_t^2 times the score" (then undoes the $\sqrt{\bar{\alpha}_t}$ shrinkage). The score was inside the denoiser from the start.

What is it, as its own model? The score-SDE view takes the number of discrete noise levels to the continuum limit: a forward stochastic differential equation smoothly turns data into noise over $t \in [0, T]$, and a reverse SDE — driven by exactly the score Eq. (7.7) defines — turns noise back into data.

Model at a glance: Score-SDE Song et al., 2021

In one line	Replace DDPM's T discrete noise levels with a continuous-time stochastic process; train a network to estimate the score at every real-valued t, not just at T fixed rungs.
Problem it solves	DDPM's discrete chain is one particular discretization choice among many. Making the noising process continuous unifies every fixed-schedule diffusion variant as one underlying object.
Key intuition	Tweedie's identity already showed a trained denoiser computes the score at every noise level it is asked about; letting t range continuously is a change of bookkeeping, not a new idea.
Inputs	A noisy state $\boldsymbol{x}_t$ and continuous time t, fed to a score network $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \approx \nabla_{\boldsymbol{x}_t} \log p_t(\boldsymbol{x}_t)$.
Loss	Denoising score matching at every t — the continuous-time twin of the DDPM noise-prediction MSE, related to it by Eq. (7.7).
Sampling	Simulate the reverse *SDE* (stochastic, many small steps) or its associated *probability-flow ODE* (deterministic, same marginals — the continuous-time generalization of DDIM).
What you keep	One formalism containing DDPM, DDIM, and (via the probability-flow ODE) the continuous-normalizing-flow view of Section 7.6, as special cases or limits of a single process.
Strengths	Principled and schedule-agnostic; the deterministic ODE branch admits exact density evaluation in principle (Section 7.6).
Weaknesses	Stochastic-calculus machinery is heavier to state than DDPM's discrete chain, for no practical gain if a model is only ever sampled on one fixed schedule.
Leads to	Rectified flow (Section 7.5) selects one particularly simple member of this same ODE family — a straight line — and trains it directly, without ever writing down an SDE.

What changed — from DDPM's discrete chain to the score-SDE view

Before (DDPM). T discrete noise levels, a Markov chain, and an ELBO derived specifically for that chain and that schedule.

The change. Let the number of levels go to infinity: the discrete chain becomes a continuous-time stochastic differential equation, and "predict the noise" becomes "estimate the score," defined at every real t, not just at T fixed rungs.

What it cost. Heavier machinery (stochastic calculus) to state precisely, bought back by a formalism in which DDPM, DDIM, and the probability-flow ODE are one continuous object viewed at different settings, rather than three unrelated recipes.

Key takeaway

Tweedie's identity is the whole score-based SDE literature compressed into one line: a denoiser's output already *is* a score, up to a known rescaling by the noise variance. Everything the score-SDE formulation adds on top — continuous time, a reverse SDE, a probability-flow ODE — is machinery built *around* that one fact, not a new fact about denoising.

7.4.4 The ELBO and the true likelihood are different curves

Chapter 4 derives the VAE's evidence lower bound; this subsection runs it backwards as the quiz demands. Write the ELBO identity rather than the inequality:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \;=\; \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}\mid\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}\mid\boldsymbol{z})] \;-\; D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}\mid\boldsymbol{x})\,\|\,p(\boldsymbol{z}))}_{\text{ELBO}} \;+\; D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}\mid\boldsymbol{x})\,\|\,p_{\boldsymbol{\theta}}(\boldsymbol{z}\mid\boldsymbol{x})).$$

The final term is the posterior-approximation gap: non-negative, and zero exactly when the encoder's approximate posterior equals the true posterior.

The trap question. A team retrains *only the encoder* of a VAE, and the ELBO on their data goes up. Does the model now give the data a higher *true* likelihood? **No.**

Why: the true likelihood $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ depends only on the prior and the decoder, $\boldsymbol{\theta}$. The encoder $\boldsymbol{\phi}$ does not appear in it at all. With $\boldsymbol{\theta}$ unchanged, the true likelihood is a fixed line.

The ELBO sits *below* that line, and the gap between them is how badly the encoder's guess matches the true posterior. A better encoder shrinks the gap, so the ELBO rises *towards* the line — but the line itself does not move.

Key takeaway

Improving the encoder only tightens the bound. Only improving the decoder can raise the true likelihood.

7.4.5 Deriving the DDPM objective, step by step

The training box of Section 7.3 asks the network to minimize $\|\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{\varepsilon}\|^2$. That looks like a heuristic ("guess the noise"), but it is not. It comes out of maximizing the likelihood of the data, through a chain of steps that are each short. This subsection walks the whole chain, following Ho et al. (2020).

Step by step — the route from likelihood to "predict the noise"

1. Write the model: a fixed forward chain q and a learned reverse chain $p_{\boldsymbol{\theta}}$.
2. Bound the log-likelihood from below (the ELBO, exactly as for a VAE).
3. Rearrange the bound into one KL divergence per step.

4. Compute the "true" reverse step, which is Gaussian once $\boldsymbol{x}_0$ is known.
5. Compare two Gaussians: the KL reduces to a squared distance between means.
6. Rewrite that distance in terms of the noise $\boldsymbol{\varepsilon}$.
7. Drop the per-step weights: the result is the training loss.

Step 1: the two chains

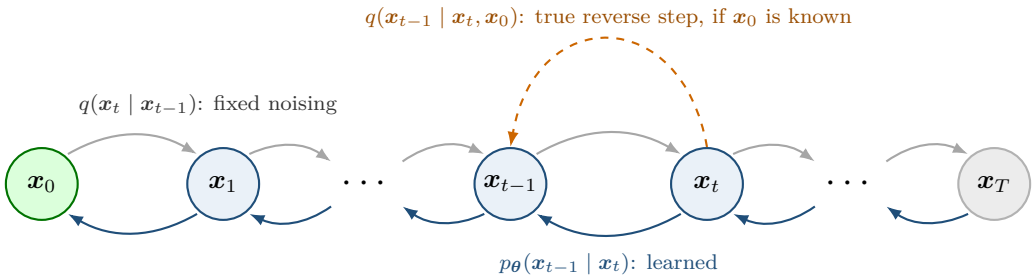
The symbols are the ones already used in Section 7.4.1: a schedule β_t, $\alpha_t = 1 - \beta_t$, and $\bar{\alpha}_t = \prod_{s \le t} \alpha_s$.

Forward (fixed, no parameters). $q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{\alpha_t}\,\boldsymbol{x}_{t-1}, \beta_t \boldsymbol{I})$, with the one-jump marginal $q(\boldsymbol{x}_t \mid \boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I})$ (Eq. (7.3)).

Reverse (learned). Start from $p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and take Gaussian steps back,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T}) = p(\boldsymbol{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t), \qquad p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t), \sigma_t^2 \boldsymbol{I}). \tag{7.8}$$

The network only has to output the mean $\boldsymbol{\mu}_{\boldsymbol{\theta}}$. The variance σ_t^2 is a fixed number per step, not learned: DDPM tried both $\sigma_t^2 = \beta_t$ and $\sigma_t^2 = \tilde{\beta}_t$ (defined in Step 4) and reports similar results (Ho et al., 2020).



The loss compares each blue step with the dashed orange one: one KL divergence per step.

Figure 7.11. The two chains of a diffusion model. The forward chain (grey, top) is fixed arithmetic. The reverse chain (blue, bottom) is what the network learns. Training never runs either chain end to end; it matches one blue step at a time to the true reverse step (orange), which is computable because during training the clean image $\boldsymbol{x}_0$ is known.

How to read it: The orange arrow is the teacher and the blue arrow below it the student, at a single step t. Step 3 below shows why the whole likelihood splits into exactly these per-step comparisons.

Step 2: the ELBO, exactly as for a VAE

The model's probability of a clean image averages over every path that could have produced it: $p_{\boldsymbol{\theta}}(\boldsymbol{x}_0) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})\, d\boldsymbol{x}_{1:T}$. That integral is intractable. Multiply and divide by the forward

chain $q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)$, which turns the integral into an average over forward paths, then use $\log \mathbb{E}[\cdot] \geq \mathbb{E}[\log \cdot]$ (Jensen's inequality):

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0) = \log \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)}\right] \;\geq\; \mathbb{E}_q\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)}\right]. \tag{7.9}$$

This is the VAE's evidence lower bound of Section 4.5, with the noisy images $\boldsymbol{x}_{1:T}$ playing the role of the latent variable and the forward chain playing the encoder. The one difference is that this encoder is *fixed*, not learned. We minimize $L = -\text{ELBO}$.

Step 3: one KL divergence per step

Substitute both chains into L. Every factor is a product, so the log is a sum:

$$L = \mathbb{E}_q\left[-\log p(\boldsymbol{x}_T) - \sum_{t=1}^{T} \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)}{q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1})}\right].$$

Each term compares a *backward* step of the model with a *forward* step of the noise, which is awkward: they go in opposite directions. The key trick flips the forward step around with Bayes' rule. Because the forward chain is Markov, $q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) = q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_0)$, and for $t \geq 2$

$$q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_0) = \frac{q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0)\; q(\boldsymbol{x}_t \mid \boldsymbol{x}_0)}{q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_0)}.$$

Substituting, the extra fractions form a telescoping product,

$$\prod_{t=2}^{T} \frac{q(\boldsymbol{x}_t \mid \boldsymbol{x}_0)}{q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_0)} = \frac{q(\boldsymbol{x}_2 \mid \boldsymbol{x}_0)}{q(\boldsymbol{x}_1 \mid \boldsymbol{x}_0)} \cdot \frac{q(\boldsymbol{x}_3 \mid \boldsymbol{x}_0)}{q(\boldsymbol{x}_2 \mid \boldsymbol{x}_0)} \cdots \frac{q(\boldsymbol{x}_T \mid \boldsymbol{x}_0)}{q(\boldsymbol{x}_{T-1} \mid \boldsymbol{x}_0)} = \frac{q(\boldsymbol{x}_T \mid \boldsymbol{x}_0)}{q(\boldsymbol{x}_1 \mid \boldsymbol{x}_0)},$$

and the leftover $q(\boldsymbol{x}_1 \mid \boldsymbol{x}_0)$ cancels against the $t = 1$ term. What remains is

$$L = \mathbb{E}_q\left[\log \frac{q(\boldsymbol{x}_T \mid \boldsymbol{x}_0)}{p(\boldsymbol{x}_T)} + \sum_{t=2}^{T} \log \frac{q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0)}{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)} - \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0 \mid \boldsymbol{x}_1)\right].$$

An expected log-ratio of two distributions is a KL divergence, so

$$\begin{aligned} L = \mathbb{E}_q\Big[& \underbrace{D_{\mathrm{KL}}(q(\boldsymbol{x}_T \mid \boldsymbol{x}_0) \,\|\, p(\boldsymbol{x}_T))}_{L_T} \underbrace{- \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0 \mid \boldsymbol{x}_1)}_{L_0} \\ & + \sum_{t=2}^{T} \underbrace{D_{\mathrm{KL}}(q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t))}_{L_{t-1}} \Big]. \end{aligned} \tag{7.10}$$

In plain words — what each term asks for

L_T asks whether the forward chain really ends in pure noise. It has no learnable parameters (the schedule is fixed), so it is a constant and training ignores it.

Each L_{t-1} asks: *given the noisy image $\boldsymbol{x}_t$, does the model's step back match the true step back?* "True step back" needs a qualifier. The reverse of the noising, $q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)$ on its own, would require knowing the whole data distribution. But during training we also know which clean image $\boldsymbol{x}_0$ we started from, and *given* $\boldsymbol{x}_0$ the step back is a simple Gaussian (Step 4). Conditioning on $\boldsymbol{x}_0$ is what makes every term computable.

L_0 is the final step to a clean image: how likely the true image is under the model's last move.

Step 4: the true step back is a Gaussian

By Bayes' rule, $q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0) \propto q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1})\, q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_0)$: a product of two Gaussians in $\boldsymbol{x}_{t-1}$, hence a Gaussian. To find its mean and variance, look at one coordinate and collect the exponent, writing x for the unknown x_{t-1}:

$$
\begin{aligned}
&-\frac{1}{2}\left[\frac{(x_t - \sqrt{\alpha_t}\,x)^2}{\beta_t} + \frac{(x - \sqrt{\bar\alpha_{t-1}}\,x_0)^2}{1-\bar\alpha_{t-1}}\right] \\
&\qquad = -\frac{1}{2}\left[\Big(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar\alpha_{t-1}}\Big)x^2 - 2\Big(\frac{\sqrt{\alpha_t}}{\beta_t}x_t + \frac{\sqrt{\bar\alpha_{t-1}}}{1-\bar\alpha_{t-1}}x_0\Big)x\right] + \text{const.}
\end{aligned}
$$

A Gaussian $\mathcal{N}(\mu, s^2)$ has exponent $-\frac{1}{2}\big[x^2/s^2 - 2\mu x/s^2\big] + \text{const}$, so $1/s^2$ is the x^2 coefficient and μ is the linear coefficient times s^2.

Variance. Over a common denominator, and using $\alpha_t(1-\bar\alpha_{t-1}) + \beta_t = \alpha_t - \bar\alpha_t + 1 - \alpha_t = 1 - \bar\alpha_t$,

$$
\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar\alpha_{t-1}} = \frac{1-\bar\alpha_t}{\beta_t(1-\bar\alpha_{t-1})} \quad\Longrightarrow\quad \tilde\beta_t = \frac{1-\bar\alpha_{t-1}}{1-\bar\alpha_t}\,\beta_t.
$$

Mean. Multiply the linear coefficient by $\tilde\beta_t$:

$$
q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_{t-1};\, \tilde{\boldsymbol{\mu}}_t,\, \tilde\beta_t \boldsymbol{I}), \qquad \tilde{\boldsymbol{\mu}}_t = \frac{\sqrt{\alpha_t}\,(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}\,\boldsymbol{x}_t + \frac{\sqrt{\bar\alpha_{t-1}}\,\beta_t}{1-\bar\alpha_t}\,\boldsymbol{x}_0. \tag{7.11}
$$

The true step back lands at a weighted average of where we are ($\boldsymbol{x}_t$) and where we started ($\boldsymbol{x}_0$), with a known, small spread $\tilde\beta_t$.

Step 5: two Gaussians, so the KL is a squared distance

For two Gaussians in d dimensions with isotropic covariances,

$$
D_{\mathrm{KL}}(\mathcal{N}(\boldsymbol{\mu}_1, s_1^2\boldsymbol{I}) \,\|\, \mathcal{N}(\boldsymbol{\mu}_2, s_2^2\boldsymbol{I})) = \frac{d}{2}\Big(\frac{s_1^2}{s_2^2} - 1 - \log\frac{s_1^2}{s_2^2}\Big) + \frac{\|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|^2}{2s_2^2}.
$$

Here $s_1^2 = \tilde{\beta}_t$ and $s_2^2 = \sigma_t^2$ are both fixed numbers, so the first part is a constant C. Only the means are learned:

$$L_{t-1} = \mathbb{E}_q\left[\frac{1}{2\sigma_t^2}\left\|\tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \boldsymbol{x}_0) - \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right\|^2\right] + C. \tag{7.12}$$

So the network should output the true posterior mean. That is already a regression loss, but on an awkward target.

Step 6: rewrite the target in terms of the noise

During training, $\boldsymbol{x}_t$ is built as $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}$ (Eq. (7.4)), so $\boldsymbol{x}_0 = \big(\boldsymbol{x}_t - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}\big)/\sqrt{\bar{\alpha}_t}$. Put this into Eq. (7.11) and collect terms.

- The $\boldsymbol{x}_0$ coefficient becomes, using $\bar{\alpha}_t = \alpha_t\bar{\alpha}_{t-1}$,

$$\frac{\sqrt{\bar{\alpha}_{t-1}}\,\beta_t}{(1-\bar{\alpha}_t)\sqrt{\bar{\alpha}_t}} = \frac{\beta_t}{(1-\bar{\alpha}_t)\sqrt{\alpha_t}}.$$

- The total coefficient on $\boldsymbol{x}_t$ is then

$$\frac{\sqrt{\alpha_t}\,(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} + \frac{\beta_t}{(1-\bar{\alpha}_t)\sqrt{\alpha_t}} = \frac{\alpha_t(1-\bar{\alpha}_{t-1}) + \beta_t}{(1-\bar{\alpha}_t)\sqrt{\alpha_t}} = \frac{1}{\sqrt{\alpha_t}},$$

  by the same identity as in Step 4.
- The coefficient on $\boldsymbol{\varepsilon}$ is $-\dfrac{\beta_t\sqrt{1-\bar{\alpha}_t}}{(1-\bar{\alpha}_t)\sqrt{\alpha_t}} = -\dfrac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar{\alpha}_t}}$.

Together:

$$\tilde{\boldsymbol{\mu}}_t = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\boldsymbol{\varepsilon}\right). \tag{7.13}$$

Look at what is unknown on the right. $\boldsymbol{x}_t$ is the network's input; α_t and β_t are schedule constants. The only thing the network cannot see is $\boldsymbol{\varepsilon}$. So give the model's mean exactly the same form, with a network guessing the noise:

$$\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right). \tag{7.14}$$

Subtracting, the $\boldsymbol{x}_t$ terms cancel and $\tilde{\boldsymbol{\mu}}_t - \boldsymbol{\mu}_{\boldsymbol{\theta}} = \frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar{\alpha}_t}}(\boldsymbol{\varepsilon}_{\boldsymbol{\theta}} - \boldsymbol{\varepsilon})$. Substituting into Eq. (7.12):

$$L_{t-1} = \lambda_t\,\mathbb{E}_{\boldsymbol{x}_0,\boldsymbol{\varepsilon}}\Big[\big\|\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_{\boldsymbol{\theta}}\big(\sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon},\ t\big)\big\|^2\Big] + C, \qquad \lambda_t = \frac{\beta_t^2}{2\sigma_t^2\,\alpha_t\,(1-\bar{\alpha}_t)}. \tag{7.15}$$

This is "predict the noise", derived. Each step of the likelihood bound is a noise-prediction error at that step's noise level, times a known weight λ_t.

Step 7: drop the weights

Summing Eq. (7.15) over t gives the true bound. DDPM instead trains on the unweighted version, with t drawn uniformly:

$$L_{\text{simple}}(\boldsymbol{\theta}) = \mathbb{E}_{t,\boldsymbol{x}_0,\boldsymbol{\varepsilon}}\Big[\big\|\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon},\ t)\big\|^2\Big], \qquad t \sim \text{Uniform}\{1,\dots,T\}. \tag{7.16}$$

This is exactly the loss in the training box. Three things were dropped or approximated along the way: L_T (a constant), the weights λ_t, and L_0, which DDPM folds into the $t = 1$ case with an approximation of its discretized decoder (Ho et al., 2020).

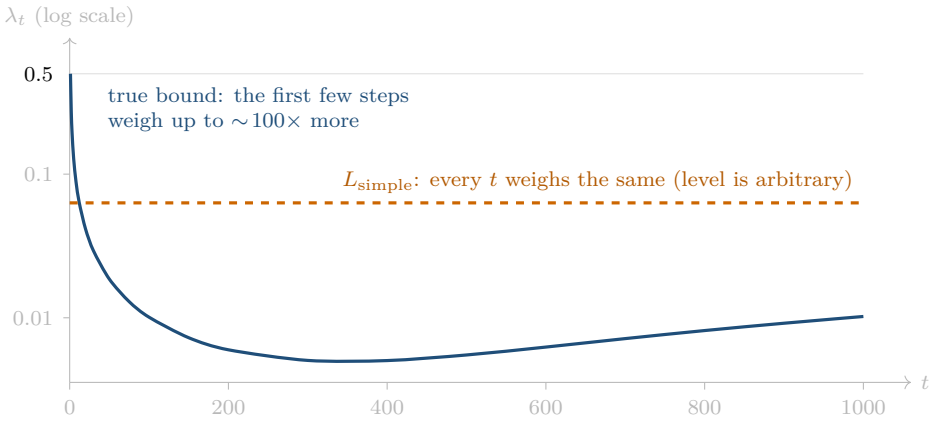


Figure 7.12. The per-step weight λ_t of Eq. (7.15) for DDPM's linear schedule ($\beta_1 = 10^{-4}$ to $\beta_T = 0.02$, $T = 1000$, $\sigma_t^2 = \beta_t$), computed for this book. The true bound gives the nearly clean steps at small t weights around 0.5, against about 0.005 in the middle of the chain. L_{simple} replaces the curve with a flat line, which relatively *down-weights* the easy, nearly clean steps.

How to read it: The vertical axis is logarithmic: each gridline is a factor of 10. *Only the* shape *of the blue curve matters when comparing to the flat line, since multiplying a loss by a constant does not change its minimizer.*

Why dropping the weights helps. The paper's explanation is that the small-t terms train the network to remove tiny amounts of noise, which is easy, and the true bound spends most of its weight there (Figure 7.12). Flattening the weights shifts effort to the harder denoising problems at larger t, and the experiments bear it out (Ho et al., 2020):

Two readings. On the true bound, predicting the mean and predicting the noise are about equally good (FID 13.22 against 13.51), as they should be: the two are the same quantity, rearranged. The large gain comes only from predicting the noise *and* dropping the weights (FID 3.17). The price is honest: L_{simple} is no longer the likelihood bound, and the paper reports that training on the true bound gives better log-likelihoods, while L_{simple} gives the best samples.

Table 7.1. Parameterization and objective ablation on unconditional CIFAR-10, from Table 2 of Ho et al. (2020). Lower FID is better. "Fixed Σ" means σ_t^2 is a constant per step, as above; missing entries were unstable to train.

Network predicts	Objective	IS	FID
mean $\tilde{\boldsymbol{\mu}}$	true bound, learned diagonal Σ	7.28 ± 0.10	23.69
mean $\tilde{\boldsymbol{\mu}}$	true bound, fixed Σ	8.06 ± 0.09	13.22
mean $\tilde{\boldsymbol{\mu}}$	unweighted $\|\tilde{\boldsymbol{\mu}} - \boldsymbol{\mu}_{\boldsymbol{\theta}}\|^2$	–	–
noise $\boldsymbol{\varepsilon}$	true bound, fixed Σ	7.67 ± 0.13	13.51
noise $\boldsymbol{\varepsilon}$	L_{simple} (unweighted)	$\mathbf{9.46 \pm 0.11}$	**3.17**

The sampling rule falls out for free

Sampling means drawing from $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)$, a Gaussian with mean Eq. (7.14) and variance σ_t^2:

$$\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \right) + \sigma_t \, \boldsymbol{z}, \qquad \boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}), \tag{7.17}$$

with $\boldsymbol{z} = \mathbf{0}$ at the final step. This is the "reverse-transition formula" of Section 7.3.3. In words: remove a scaled copy of the predicted noise, rescale slightly, and add back a little fresh noise.

Worked Example — one step, with numbers

Work in one dimension at a step where $\beta_t = 0.1$ (so $\alpha_t = 0.9$), $\bar{\alpha}_{t-1} = 0.5$ and therefore $\bar{\alpha}_t = 0.45$. Take the clean value $x_0 = 1$ and noise $\varepsilon = 0.5$.

Forward. $x_t = \sqrt{0.45} \cdot 1 + \sqrt{0.55} \cdot 0.5 \approx 0.6708 + 0.3708 = 1.0416$.

True step back, from Eq. (7.11). The coefficients are $\sqrt{0.9}\,(0.5)/0.55 \approx 0.8624$ on x_t and $\sqrt{0.5}\,(0.1)/0.55 \approx 0.1286$ on x_0, so $\tilde{\mu}_t \approx 0.8624 \times 1.0416 + 0.1286 \approx 1.0269$, with variance $\tilde{\beta}_t = (0.5/0.55)(0.1) \approx 0.0909$.

The same mean from the noise, Eq. (7.13). $\frac{1}{\sqrt{0.9}}(1.0416 - \frac{0.1}{\sqrt{0.55}} \cdot 0.5) \approx 1.0541 \times (1.0416 - 0.0674) \approx 1.0269$. It matches: knowing the noise is as good as knowing x_0.

A network that is slightly wrong. If $\varepsilon_{\boldsymbol{\theta}} = 0.4$, its mean is $1.0541 \times (1.0416 - 0.1348 \times 0.4) \approx 1.0411$, off by 0.0142. Eq. (7.15) predicts exactly this: the mean error is $\beta_t/(\sqrt{\alpha_t}\sqrt{1-\bar{\alpha}_t}) \approx 0.1421$ times the noise error 0.1. With $\sigma_t^2 = \tilde{\beta}_t$, the weight is $\lambda_t = 0.01/(2 \times 0.0909 \times 0.9 \times 0.55) \approx 0.111$, and this step's loss is $0.111 \times 0.1^2 \approx 0.0011$ above its minimum.

Key takeaway — the derivation in one paragraph

Maximize the likelihood; since it is intractable, maximize the ELBO, a VAE bound whose encoder is the fixed noising chain. A telescoping Bayes trick turns the bound into one KL divergence per step, each between the model's step back and the true step back given $\boldsymbol{x}_0$. Both are Gaussians with fixed variances, so each KL is a squared distance between

means. Writing the true mean in terms of the noise shows that the only unknown is ε, so the network predicts ε, and each KL becomes a weighted noise-prediction error. Dropping the weights gives L_{simple}, the loss everyone trains with; the same Gaussian gives the sampling rule.

7.4.6 A DAE trained at one noise scale does not transfer to another

Quiz questions Q05 and Q14 extend Section 7.2 in a direction the lecture slides only signal. A denoising autoencoder trained at one small fixed scale, say $\sigma = 0.1$, learns a *local* projection back onto the data manifold: it assumes the corrupted point lies one short step away and learns to move it that distance, approximately orthogonally to the manifold.

Show the same network an input corrupted at $\sigma = 0.6$ and it fails: the point now sits far off the training manifold, out of the distribution the denoiser ever saw. It still applies the short learned step, so it under-cleans and visible corruption remains. The single-scale DAE has no representation of *how far* its input is from clean data, hence no way to lengthen its step when it should.

The diffusion remedy is *amortization over time*: train one network $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ taking the noise level t as a regular input (through a time embedding), so one parameter set covers every scale. Neighboring t values are similar denoising problems, so parameter sharing improves data efficiency, and the network interpolates smoothly across scales instead of needing ∼1000 separate models. Amortization turns the discrete ladder of denoisers into a single continuous family—which the flow-matching sections then write as one time-conditioned velocity field.

7.4.7 The timestep is a semantic-granularity dial

Quiz question Q09 pre-answers a design choice Section 7.10.1 will need: when reading diffusion U-Net features for downstream tasks, *which* timestep t should the input be noised to? The quiz's answer: t acts as a granularity dial.

- At *large* t (in a 1000-step DDPM schedule, $t \approx 500$) the input is mostly noise; only coarse structure is recoverable. The network's features encode global layout and part-level semantics—the "what" and "where"—which generalize across instances (matching, say, the bill of one duck to the bill of another).
- At *small* t ($t \approx 10$) the input still contains fine texture; the features track local detail and near-exact pixel content. Those transfer poorly across instances of a category but are the right instrument for within-instance tracking.

Choosing t is choosing the level of meaning. Which timestep you extract features at is a real design decision, not a detail. Think of t as a dial: low t gives fine, texture-level features; high t gives coarse, category-level ones. The lecture's examples — matching a point

on a duck's bill to the same point on other duck species, or a real guitar to a painted one — only work with the dial turned towards the abstract end.

DIFT's chosen middle setting (Section 7.10.1) sits where the two needs balance: abstract enough to recognise "this is the bill", precise enough to say *where* on the image it is.

Key takeaway

> Every result in this section is the same closed-form marginal (Eq. (7.4)) looked at from a different angle: it makes training $O(1)$ per step, it makes the three parameterizations interchangeable but not equally well-conditioned, it turns a trained denoiser into a score estimator via Tweedie's identity, and it hands the timestep a semantic meaning that later sections (DIFT) put to direct use. One equation, many questions.

7.5 Rectified flow and flow matching, code first

The lecture's central move (slides 19–34) is to teach diffusion's modern formulation—*rectified flow* (Liu et al., 2023) / *flow matching* (Lipman et al., 2023)—almost entirely as evolving PyTorch pseudocode. We follow it, with more geometry spelled out. But first we answer a question the slides leave implicit: *why* does this idea exist at all? Four names appear in this part of the course — normalizing flows, continuous normalizing flows, flow matching, rectified flow — and it is easy to learn them as four unrelated tricks. They are not. Each one exists because the one before it left one specific problem unsolved. Once you see that chain, the five-line training loop below stops looking arbitrary.

7.5.1 Why we need all four: one problem, handed down

All four share one goal: an *invertible* map from Gaussian noise to images. Invertible, because then we get two things at once — samples (run the map forwards) and exact likelihoods (run it backwards and count how much it stretched space). Figure 7.13 is the map of this subsection.

Step 1 — normalizing flows (Chapter 6): exact, but in a straitjacket. A normalizing flow builds the map from a stack of invertible layers and gets $\log p(\boldsymbol{x})$ exactly from the change-of-variables formula (Eq. (7.1)). The price is that *every* layer must be invertible *and* have a Jacobian determinant that is cheap to compute. That is why flows use coupling layers that change only half the numbers at a time, 1×1 convolutions stored in LU form, and the same dimension everywhere. The architecture is chosen to suit the determinant, not the data. Flows needed very deep stacks and still produced visibly worse images than GANs. *Left open: can we get invertibility without designing every layer around it?*

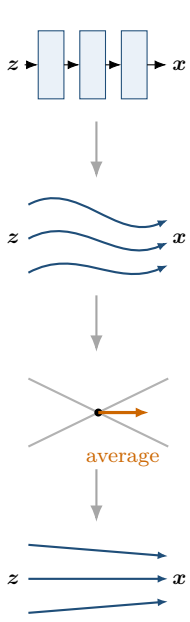
Step 2 — continuous normalizing flows: any network, but slow to train. Replace the few big invertible jumps by infinitely many tiny ones. A point moves through space with velocity given by an ordinary, unconstrained network, $\frac{dz}{dt} = f_{\boldsymbol{\theta}}(\boldsymbol{z}, t)$, and the map is "wherever the point is when the clock stops" (Chen et al., 2018). Two gifts come free:

- *Invertibility for any network.* Two trajectories of an ODE can never cross: if they were ever at the same point at the same time, they would have the same velocity from then on (and, running the clock backwards, the same past), so they would be one trajectory. Distinct starting points therefore stay distinct, and running the clock backwards inverts the map. No coupling layers needed.
- *Determinants become traces.* Over a tiny time step dt the map is $\boldsymbol{z} \mapsto \boldsymbol{z} + f\,dt$, whose log-determinant is $\log\det(\boldsymbol{I} + \boldsymbol{J}\,dt) \approx \operatorname{tr}(\boldsymbol{J})\,dt$. A trace is much cheaper than a determinant (Section 7.6).

Left open: training. Maximum likelihood needs $\log p(\boldsymbol{x})$ for every training image. For a CNF that means solving the ODE from $\boldsymbol{x}$ back to noise inside *every* gradient step — tens to hundreds of network calls, plus a trace estimate at each one, plus backpropagation through the whole solve. This is called *simulation-based* training, and it is slow; the CNFs of that era stayed on small images such as MNIST and CIFAR-10 (Grathwohl et al., 2019).

Step 3 — flow matching: train the velocity directly, no simulation. Suppose someone handed us a velocity field $\boldsymbol{u}(\boldsymbol{x}, t)$ already known to carry noise onto data. Then we would not need likelihoods at all: pick a random time and point, and *regress*, $\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{u}(\boldsymbol{x}_t, t)\|^2$. One network call per training example, no ODE solve. We do not know $\boldsymbol{u}$ for the whole distribution, but we *do* know it for a single noise–data pair: if we decide that this pair travels along a straight line, its velocity is simply $\boldsymbol{z} - \boldsymbol{x}$. Flow matching (Lipman et al., 2023) regresses on these per-pair velocities. Squared error averages all the pairs that pass through a point, and that average turns out to be exactly the field that moves the whole noise distribution onto the whole data distribution (Section 7.5.4). We end up with the same object as a CNF, trained like ordinary supervised regression. *Left open: which path should each pair follow?* Any path works, and the learned field inherits its shape. A curved field needs many small sampling steps.

Step 4 — rectified flow: choose straight paths, then straighten the result. Rectified flow (Liu et al., 2023) makes the simplest choice — the straight line at constant speed — and adds *reflow*: a second training round on the model's own (noise, sample) pairs that removes most of the remaining curvature (Section 7.5.6). The straighter the field, the fewer steps sampling needs, down to one in the limit.



1. Normalizing flow (Ch. 6)
a stack of hand-designed invertible layers; exact $\log p(\boldsymbol{x})$ as a *sum of log-determinants*.

left open: every layer must be invertible with a cheap determinant — the maths dictates the architecture.

2. Continuous normalizing flow
any network $f_{\boldsymbol{\theta}}$ as a velocity; trajectories cannot cross, so the map is invertible for free; log-det becomes an *integral of traces*.

left open: maximum-likelihood training must solve the ODE inside every gradient step — slow.

3. Flow matching
regress $f_{\boldsymbol{\theta}}$ on the known velocity of each noise–data *pair*; squared error averages the pairs into the right field. One network call per example.

left open: the path chosen per pair shapes the field; curved fields need many sampling steps.

4. Rectified flow
flow matching with *straight* per-pair paths, plus *reflow* to straighten the learned field further.

pay-off: a nearly straight field, so a few Euler steps (in the limit one) are enough.

Figure 7.13. Four models, one problem handed down. Each row fixes the problem the row above leaves open (red). Pictograms: a stack of invertible blocks; smooth ODE trajectories that never cross; two crossing per-pair lines whose velocities the network averages (orange); straight, non-crossing paths.

How to read it: Read the red column top to bottom: it is the reason each next row exists. Rows 2–4 all learn the same kind of object — *a velocity field defining an invertible ODE — and differ only in how it is trained and which paths it is trained on.*

Key Concept — What, how, which path

The four names answer different questions, which is why they fit together rather than compete.

- **CNF** is *what* is learned: a velocity field $f_{\boldsymbol{\theta}}(\boldsymbol{x}, t)$ whose ODE is an invertible noise-to-data map with a computable likelihood.
- **Flow matching** is *how* it is trained: regression on per-pair velocities, with no ODE solve during training.
- **Rectified flow** is *which paths* it is trained on: straight lines, plus reflow.

Diffusion fits the same frame: DDPM's noising process is one more choice of per-pair path (a curved one, set by the $\bar{\alpha}_t$ schedule), and DDIM's deterministic sampler is the ODE of the resulting field. That is the sense in which the closing slide moves diffusion next to the flows (Figure 7.1).

Everyday analogy — a staircase, a ramp, a survey, a straight ramp

A normalizing flow is a staircase built from prefabricated steps, each one certified safe on its own; you can only build shapes the certified steps allow. A CNF replaces it with a smooth ramp of any shape — but to learn where the ramp goes you must walk all of it, every time. Flow matching surveys the ramp instead of walking it: go to random spots, record which way people were heading there, and average. Rectified flow builds the ramp straight, so one stride takes you from top to bottom.

Where the analogy breaks. The "survey" does not measure one true slope at each spot. It averages many contradictory per-pair directions, and the proof that this average is the right field is the real content of flow matching (Section 7.5.4).

Common Confusion

"Flow matching and rectified flow are the same thing." With straight per-pair paths and Gaussian noise, the two papers' training losses coincide, which is why the lecture uses the names together. The differences are framing and extras: flow matching is the general recipe for *any* per-pair path; rectified flow is the straight-line choice together with the reflow procedure. **Watch the clock.** This chapter and the lecture put data at $t = 0$ and noise at $t = 1$; Lipman et al. (2023) use the opposite convention. Swapping $t \leftrightarrow 1 - t$ flips the sign of every velocity, so check the convention before copying a sampler.

7.5.2 Setup: transport between two point clouds

Suppose we have a simple p_{noise} (a unit Gaussian) and samples from p_{data}. Picture two blobs in the plane (Figure 7.14): blue points $\boldsymbol{z} \sim p_{\text{noise}}$ at the top, red points $\boldsymbol{x} \sim p_{\text{data}}$ at the bottom. The generative problem is a *transport mapping problem*: find a map that carries the blue distribution onto the red one. Rectified flow learns it as a *velocity field*—a function $\boldsymbol{v}(\boldsymbol{x}_t, t)$ giving, at every intermediate state $\boldsymbol{x}_t$ and time t, the direction and speed at which mass should move.

7.5.3 Three definitions

On each training iteration, sample

$$\boldsymbol{z} \sim p_{\text{noise}}, \qquad \boldsymbol{x} \sim p_{\text{data}}, \qquad t \sim \text{Uniform}[0, 1],$$

all independently. Define the interpolation and its velocity:

$$\boldsymbol{x}_t = (1 - t)\,\boldsymbol{x} + t\,\boldsymbol{z}, \tag{7.18}$$

$$\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}. \tag{7.19}$$

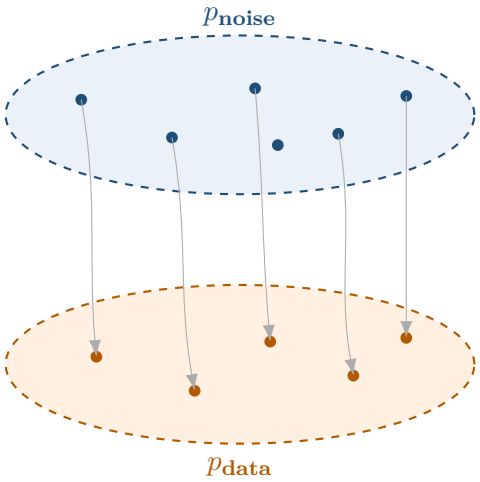


Figure 7.14. The problem rectified flow solves: find *any* map that carries samples of p_{noise} (top) onto samples of p_{data} (bottom). The thin grey arrows show one arbitrary pairing — not a claim about which noise point "belongs" to which data point; Section 7.5.6 returns to exactly that question.

How to read it: Nothing here is learned yet — this is the problem statement, before any equation. Compare Figure 7.5, which shows the same two blobs after a specific point $\boldsymbol{x}_t$ and its velocity $\boldsymbol{v}$ have been picked out and the network has something to regress against.

Equation (7.18) puts $\boldsymbol{x}_t$ a fraction t of the way along the straight line from the data point $\boldsymbol{x}$ to the noise point $\boldsymbol{z}$. Its speed is found by differentiating:

$$\frac{d}{dt}\boldsymbol{x}_t \;=\; \frac{d}{dt}\big[(1-t)\,\boldsymbol{x} + t\,\boldsymbol{z}\big] \;=\; \boldsymbol{z} - \boldsymbol{x}.$$

The answer does not depend on t: the point moves at *constant* velocity $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$. Note the direction: $\boldsymbol{v}$ points from data *towards* noise, because $t = 0$ is data and $t = 1$ is noise. Generation will therefore walk *against* $\boldsymbol{v}$.

Everyday analogy — a mountain road versus a desert highway

Earlier diffusion models follow curved paths from noise to data — like a winding mountain road, which you must drive in many small, careful steps. Rectified flow lays a straight highway instead. On a straight road at constant speed you can take a few very large steps without leaving it.

Where the analogy breaks. Each individual training line is straight, but the field the network learns is only approximately so (see "reflow" below).

So we train the network to predict that velocity:

$$\mathcal{L}_{\text{RF}}(\boldsymbol{\theta}) \;=\; \mathbb{E}_{t,\boldsymbol{x},\boldsymbol{z}}\big[\,\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{v}\|_2^2\,\big]. \tag{7.20}$$

That is all. "The core training loop is just a few lines of code" (slide 23):

```python
for x in dataset:
    z = torch.randn_like(x)
    t = random.uniform(0, 1)
    xt = (1 - t) * x + t * z
    v = model(xt, t)
    loss = (z - x - v).square().sum()
```

Listing 7.1. Rectified-flow training step (Lecture 05, slide 23).

Model at a glance: Rectified Flow — Liu, Gong & Liu, 2023

In one line	Connect a noise sample and a data sample by a *straight line*; train a network to predict that line's constant velocity; generate by integrating the learned velocity field backwards.
Problem it solves	DDPM's Gaussian chain needs hundreds to a thousand sampling steps because its underlying path is curved. A straight path can, in principle, be integrated exactly in a single step.
Key intuition	Regressing the velocity of a randomly paired noise/data segment, by squared error, recovers the *average* velocity field at every point — and integrating an average field still transports the full noise distribution onto the full data distribution.
Inputs	An interpolated point $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$ and its time $t \in [0,1]$ — no discrete timestep index, no noise schedule $\bar{\alpha}_t$.
Loss	$\mathbb{E}_{t,\boldsymbol{x},\boldsymbol{z}}\left[\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t) - (\boldsymbol{z}-\boldsymbol{x})\|^2\right]$ — Eq. (7.20), five lines of code (Listing 7.1).
Training	One forward pass, one gradient step, at a single uniformly random $t \in [0,1]$ — the continuous-time twin of the DDPM training step.
Sampling	Euler-integrate $\dot{\boldsymbol{x}}_t = f_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$ backwards from t=1 to t=0 in T steps (Listing 7.2), $T \approx 50$ typically — far fewer than DDPM because the learned path is close to straight.
What you keep	One velocity network, usable as a deterministic ODE for editing and interpolation — the same trick DDIM plays, without ever deriving a Markov chain first.
Strengths	Simplest training loop in the chapter; straight paths tolerate large Euler steps; unifies cleanly with continuous normalizing flows (Section 7.6).
Weaknesses	One training pass gives only *approximately* straight paths (Section 7.5.6); exact one-step sampling needs an extra "reflow" training round.
Leads to	Read as an ODE, this is a continuous normalizing flow (Section 7.6); conditioned on $\boldsymbol{y}$, it is guided generation (Section 7.7); composed with an autoencoder, it is latent diffusion (Section 7.8).

7.5.4 Why regressing a random segment's velocity is correct

An attentive reader objects: $\boldsymbol{x}_t$ was built from *one random* pairing of $\boldsymbol{x}$ and $\boldsymbol{z}$; many different pairs pass through the same point $\boldsymbol{x}_t$ at the same time t, each with a different velocity $\boldsymbol{z}-\boldsymbol{x}$. Which does the network learn? Minimizing the squared error in Eq. (7.20) means $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$

converges to the *conditional mean* of the target given the input—and it is worth seeing this conclusion rather than asserting it. Fix the conditioning $(\boldsymbol{x}_t, t)$ and treat the network's output as a constant u to be optimized against the random target $V = \boldsymbol{z} - \boldsymbol{x} \mid (\boldsymbol{x}_t, t)$. The expected squared loss is

$$\begin{aligned}\mathbb{E}[\|u - V\|^2] &= \mathbb{E}[\|u - \mathbb{E}[V] + \mathbb{E}[V] - V\|^2] \\ &= \mathbb{E}[\|u - \mathbb{E}[V]\|^2] \;+\; 2\,(u - \mathbb{E}[V])^\top \underbrace{\mathbb{E}[\mathbb{E}[V] - V]}_{=\,0} + \mathbb{E}[\|\mathbb{E}[V] - V\|^2] \\ &= \|u - \mathbb{E}[V]\|^2 + \mathrm{Var}(V),\end{aligned}$$

where the cross-term vanishes by the definition of the mean and the last line splits the loss into a term the optimizer controls and an irreducible conditional variance. The minimizer is therefore exactly $u = \mathbb{E}[V] = \mathbb{E}[\boldsymbol{z} - \boldsymbol{x} \mid \boldsymbol{x}_t, t]$, no matter how irregular the conditional distribution of V is:

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \;\longrightarrow\; \mathbb{E}[\,\boldsymbol{z} - \boldsymbol{x} \mid \boldsymbol{x}_t,\, t\,].$$

In words: the network learns the *average* direction of all the training lines passing through each point. The flow-matching theorem (Lipman et al., 2023) and the rectified-flow construction (Liu et al., 2023) prove that following this averaged field from noise at $t = 1$ ends at the data distribution at $t = 0$. Individual training lines cross and collide; the averaged field does not, and no single training example ever travelled along it. (Pairing each $\boldsymbol{x}$ with a well-chosen $\boldsymbol{z}$, instead of at random as in Listing 7.1, makes the lines cross less and the field straighter — a refinement beyond the slides.)

In plain words — what that derivation just proved

Every training pair $(\boldsymbol{x}, \boldsymbol{z})$ gives the network one straight-line direction: "from this noise point, head towards that data point". Many different pairs pass through the same spot $\boldsymbol{x}_t$, each pointing a different way, so the targets the network sees at that spot contradict each other.

Squared-error regression resolves contradictions in one specific way: it learns their *average*. The derivation shows that this average is not a compromise or an approximation — it is exactly the velocity field that moves the whole noise distribution onto the whole data distribution.

Everyday analogy — a busy crossroads

Stand at one street corner and ask every driver passing through where they are heading. Individually, they point every which way. Average their directions and you get the *traffic flow* at that corner — a single arrow. Nobody drove exactly along that arrow, yet if you draw it at every corner of the city, following those arrows moves the city's traffic exactly the way the real drivers did, in aggregate.

Flow matching trains on individual drivers (single noise–data pairs) and learns the traffic flow (the averaged field). Sampling then follows the traffic flow.

Where the analogy breaks. Real traffic averages can send you down a street nobody wanted; the flow-matching theorem is what guarantees that, for *probability mass*, following the averaged arrows lands you on the right distribution.

Worked Example — A two-point dataset, traced by hand

Let p_{data} be a single point mass at $x = 4$ on the real line, and let z come from a *two-point* noise distribution: $z = -2$ or $z = +2$ with equal probability. Then $\boldsymbol{x}_t = (1-t)x + tz$ takes only two values at any time: the path toward $z = +2$ passes through $x_t^{(+)} = 4 - 2t$, the path toward $z = -2$ through $x_t^{(-)} = 4 - 6t$, with constant velocities $v^{(+)} = -2$ and $v^{(-)} = -6$ (negative: from data toward noise as t grows; the sampler walks backwards). Figure 7.16 draws every step of this trace, forward and in reverse.

Trace $t = \frac{1}{2}$. The paths sit at $x_{1/2}^{(+)} = 3$ and $x_{1/2}^{(-)} = 1$, still distinct, so the optimal predictor is exact: $f_\theta(3, \frac{1}{2}) = -2$, $f_\theta(1, \frac{1}{2}) = -6$. No averaging yet. Now trace $t = 1$: both paths end at their noise points $z = \pm 2$, but these are *different* inputs, so still no collision. The two paths meet only where $4 - 2t = 4 - 6t$, i.e. at $t = 0$ (the shared data point). At $t = 0$, the conditional mean target is $\mathbb{E}[z - x \mid x_0 = 4, t = 0] = \frac{1}{2}(-2) + \frac{1}{2}(-6) = -4$. So the learned field at the data point itself is the average of the two training velocities—exactly the conditional-mean mechanism at work.

A sanity check with the dataset-wide mean: with a continuous standard Gaussian p_{noise} and the same point-mass data at $x = 4$, the paths are $x_t = 4 + t(z - 4)$; at $t = \frac{1}{2}$, the interpolated states are $x_{1/2} = 2 + z/2$, spread over 2 ± 0.5 (one Gaussian width), and along any of them the constant velocity is $v = z - 4$. Note that at $t = 0$ the conditioning set $(x_t = 4, t = 0)$ now has measure zero under the continuous noise—the formal conditional mean is $\mathbb{E}[z - 4 \mid x_0 = 4, t = 0] = \mathbb{E}[z] - 4 = -4$, but no training batch ever lands exactly there, so a parametric network learns the local average over the thin tube around the ray instead. Integrating an Euler sampler backwards from $z \sim \mathcal{N}(0, 1)$ follows these rays home to $x_0 \approx 4$—the sampler converts Gaussian width into a delta at the data point, as it must.

7.5.5 Sampling: Euler, backwards

Generation reverses the clock (slides 24–33). Choose a number of steps T (often $T = 50$), sample $\boldsymbol{x} \sim p_{\text{noise}}$, and walk from $t = 1$ to $t = 0$ in T equal steps. At each step, ask the network for the velocity at the current point and time, then move a small distance *against* it:

$$\hat{\boldsymbol{v}} \;=\; f_{\boldsymbol{\theta}}(\boldsymbol{x}, t), \qquad \boldsymbol{x} \;\leftarrow\; \boldsymbol{x} - \frac{\hat{\boldsymbol{v}}}{T}, \qquad t = 1,\ 1 - \tfrac{1}{T},\ \ldots,\ \tfrac{1}{T}. \tag{7.21}$$

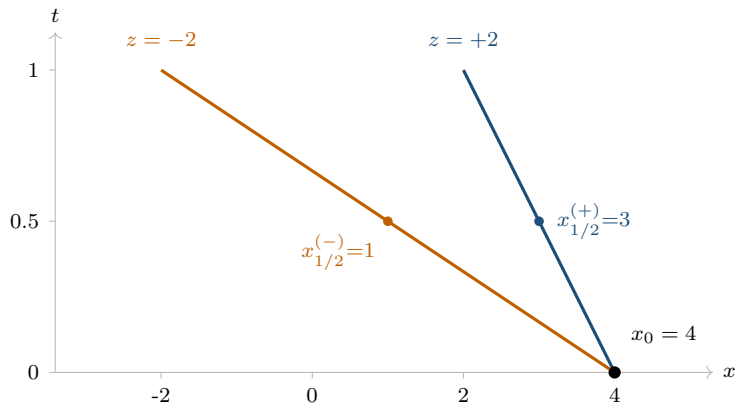


Figure 7.15. The worked example above, drawn as position x (horizontal) against time t (vertical). Two straight rays leave the single data point $x_0 = 4$ at $t = 0$ and arrive at the two noise points $z = \pm 2$ at $t = 1$; their constant slopes are the velocities $v^{(+)} = -2$ and $v^{(-)} = -6$.

How to read it: The two rays touch only *at $t = 0$ — everywhere else, including at $t = \frac{1}{2}$ where they sit at $x = 3$ and $x = 1$, they are at different points, so the network's target there is exact, not an average. Averaging is needed only exactly where rays cross, and here that is a single point.*

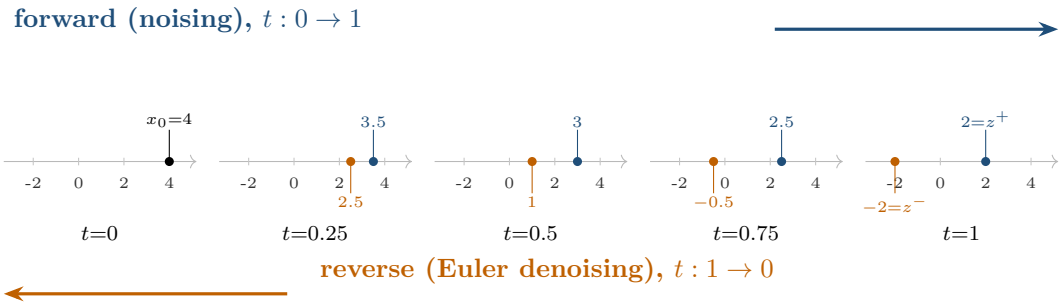


Figure 7.16. The two-point dataset of the worked example, as a strip of snapshots. Blue dots trace $x_t^{(+)} = 4 - 2t$ (the path toward z=+2), orange dots trace $x_t^{(-)} = 4 - 6t$ (the path toward z=−2); every position is the exact value computed in the text. Read the strip left-to-right for the forward (noising) process — one point splits into two — and right-to-left for the reverse (Euler) sampler: two points converge back onto the single black dot at $t = 0$. It is the same five snapshots either way, because the process is a deterministic ODE: nothing is regenerated when time is reversed, the arrow of travel simply flips.

How to read it: Read right-to-left. At $t = 1$ the sampler is holding two separate noise points, $z^+ = 2$ and $z^- = -2$; each step shrinks the gap (t=0.75 → 2.5, -0.5; t=0.5 → 3, 1; t=0.25 → 3.5, 2.5) until both land on $x_0 = 4$ at $t = 0$. Denoising is exactly this convergence, run one Euler step at a time.

That is T network calls, at $t = 1, 1 - \frac{1}{T}, \ldots, \frac{1}{T}$; after the last one the clock reads $t = 0$ and $\boldsymbol{x}$ is the sample. Note what the network returns: $\hat{\boldsymbol{v}}$ is a *velocity* (a direction and speed), not an image. The image is $\boldsymbol{x}$, and it changes only through the update.

Where the formula comes from. Training taught the network the velocity field of the ODE $\frac{d\boldsymbol{x}_t}{dt} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$. Over a short time Δt, position changes by velocity times time: $\boldsymbol{x}_{t+\Delta t} \approx \boldsymbol{x}_t + \Delta t\, f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ (Euler's method, the first term of a Taylor expansion). Sampling runs the clock *backwards*, so $\Delta t = -\frac{1}{T}$:

$$\boldsymbol{x}_{t-1/T} \;\approx\; \boldsymbol{x}_t - \tfrac{1}{T}\, f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t),$$

which is Eq. (7.21). The slide animation shows iterates $\boldsymbol{x}_1 \to \boldsymbol{x}_{2/3} \to \boldsymbol{x}_{1/3} \to \boldsymbol{x}_0$ leaving the noise blob, each pushed by the velocity evaluated at its current position, until $\boldsymbol{x}_0$ lands inside p_{data}.

Why the minus sign. The training velocity is $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$ (Eq. (7.19)): it points from data *towards noise*, because the clock was set up with data at $t = 0$ and noise at $t = 1$. Sampling goes the other way, so it subtracts. If the time convention were flipped (noise at $t = 0$), the velocity would flip too and the update would add; some papers write it that way.

```
sample = torch.randn(x_shape)
for t in torch.linspace(1, 0, num_steps):
    v = model(sample, t)
    sample = sample - v / num_steps
return sample
```

Listing 7.2. Rectified-flow sampling (Lecture 05, slide 33).

Training (Listing 7.1) and sampling (Listing 7.2) side by side are the lecture's summary slide 34; each fits in five lines. The entire theory of the next two sections exists to explain why those five lines work.

Common Confusion

The lecture's loop and Eq. (7.21) differ slightly. `torch.linspace(1, 0, num_steps)` gives `num_steps` times spaced 1/(num_steps − 1) apart, *including* $t = 0$, while each update moves by 1/num_steps. So the times at which the network is queried drift slightly from where $\boldsymbol{x}$ actually is on the clock, and the last call is made at $t = 0$, where there is nothing left to do. With 50 steps the effect is small, but a loop that matches the equation exactly is

```
sample = torch.randn(x_shape)
for i in range(num_steps):
    t = 1 - i / num_steps          # 1, 1-1/T, ..., 1/T
    v = model(sample, t)
    sample = sample - v / num_steps
return sample
```

What the network returns at sampling time

During training the target was one pair's velocity $\boldsymbol{z} - \boldsymbol{x}$. At sampling time there is no pair: we only have the current point $\boldsymbol{x}_t$. So what does $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ give? The previous subsection proved it: the *average* velocity of all the training lines through that point, $\mathbb{E}[\boldsymbol{z} - \boldsymbol{x} \mid \boldsymbol{x}_t]$. This has a useful closed form. On a training line, $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$, so $\boldsymbol{z} = (\boldsymbol{x}_t - (1-t)\boldsymbol{x})/t$ and $\boldsymbol{z} - \boldsymbol{x} = (\boldsymbol{x}_t - \boldsymbol{x})/t$. Averaging over which data point the line came from,

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \;\approx\; \mathbb{E}[\boldsymbol{z} - \boldsymbol{x} \mid \boldsymbol{x}_t] \;=\; \frac{\boldsymbol{x}_t - \mathbb{E}[\boldsymbol{x} \mid \boldsymbol{x}_t]}{t}. \tag{7.22}$$

Read it as: *the network's velocity points from its current best guess of the clean image,* $\mathbb{E}[\boldsymbol{x} \mid \boldsymbol{x}_t]$*, towards where we are now, scaled by* $1/t$*.* Rearranged, the network's current guess of the final image is

$$\hat{\boldsymbol{x}}_0 \;=\; \boldsymbol{x}_t - t\, f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t). \tag{7.23}$$

This is the same "clean-image guess" that DDPM reads off its noise prediction (Eq. (7.6)), in rectified-flow form.

In plain words — what one Euler step actually does

Put Eq. (7.23) next to the update. A single step of size t (all the remaining time at once) would move $\boldsymbol{x}_t$ to $\boldsymbol{x}_t - t f_{\boldsymbol{\theta}} = \hat{\boldsymbol{x}}_0$: *exactly* the network's current guess of the clean image. A step of size $1/T$ moves only a fraction $1/(tT)$ of the way towards that guess.

So sampling is: look at where you are, guess the finished image, take a small step towards the guess, and *guess again from the new point.* Early guesses are vague (from pure noise, the best guess is roughly the average of all images). Each step commits a little, the next guess is made from a point that already carries that commitment, and the guesses sharpen.

Two examples you can compute exactly

In both examples the ideal velocity of Eq. (7.22) can be written down, so we can see what a perfectly trained network would do.

Worked Example — one data point: every step count is exact

Let the data be a single point, $x = 2$, and start from the noise $z = 1.5$. Every training line ends at $x = 2$, so $\mathbb{E}[x \mid x_t] = 2$ and the ideal velocity is $v(x_t, t) = (x_t - 2)/t$.

Two steps ($T = 2$). At $t = 1$: $v = (1.5 - 2)/1 = -0.5$, so $x \leftarrow 1.5 - (-0.5)/2 = 1.75$. At $t = \frac{1}{2}$: $v = (1.75 - 2)/0.5 = -0.5$, so $x \leftarrow 1.75 + 0.25 = 2.0$.

The velocity was the same at both steps: the path is a straight line walked at constant speed, so Euler's method is exact, and even $T = 1$ lands exactly on 2. This is the ideal rectified flow aims for.

Worked Example — two data points: why one step is not enough

Now let the data be -1 or $+1$, equally likely, with Gaussian noise. Given x_t, Bayes' rule weighs the two possible origins: x_t is $\mathcal{N}((1-t)x,\, t^2)$ for each x, and the ratio of the two likelihoods gives

$$\mathbb{E}[x \mid x_t] = \tanh\Big(\frac{(1-t)\,x_t}{t^2}\Big), \qquad v(x_t, t) = \frac{x_t - \tanh\big((1-t)x_t/t^2\big)}{t}.$$

Start from $z = 1.4$.

One step ($T = 1$). At $t = 1$ the noise has erased everything, so $\mathbb{E}[x \mid x_1] = \tanh(0) = 0$: the best guess is the *average* of the data. Then $v = 1.4$ and $x \leftarrow 1.4 - 1.4 = 0$. The sample lands on 0, which is not a data point at all. Every starting noise does the same: one step returns the average, the "blurry" answer.

Two steps ($T = 2$). At $t = 1$: $x \leftarrow 1.4 - 1.4/2 = 0.7$. At $t = \frac{1}{2}$: now $\mathbb{E}[x \mid 0.7] = \tanh(0.7 \cdot 0.5/0.25) = \tanh(1.4) \approx 0.885$, so $v = (0.7 - 0.885)/0.5 \approx -0.371$ and $x \leftarrow 0.7 + 0.185 = 0.885$. Closer, but still not on $+1$. The first step committed halfway to "the average", and the second guess was made from there.

Many steps. With $T = 4$ this start reaches 1.000, and with $T \geq 10$ every start we tried lands exactly on -1 or $+1$ (Figure 7.17).

The lesson. The individual training lines are straight, but the *averaged* field the network learns can curve wherever several data points are plausible origins, as in Figure 7.17. Euler steps follow a curve by cutting its corners, and big steps cut too much. That is why generation takes many steps, and why making the learned paths straighter (next subsection) is what allows fewer.

7.5.6 What is "rectified" about it?

Many different flows transport p_{noise} to p_{data}; the paper's position, quoted on slide 35, is that straight paths are preferred on two grounds (Liu et al., 2023). *Theoretically*, a straight line is the shortest path between endpoints, so straight trajectories minimize transport effort.

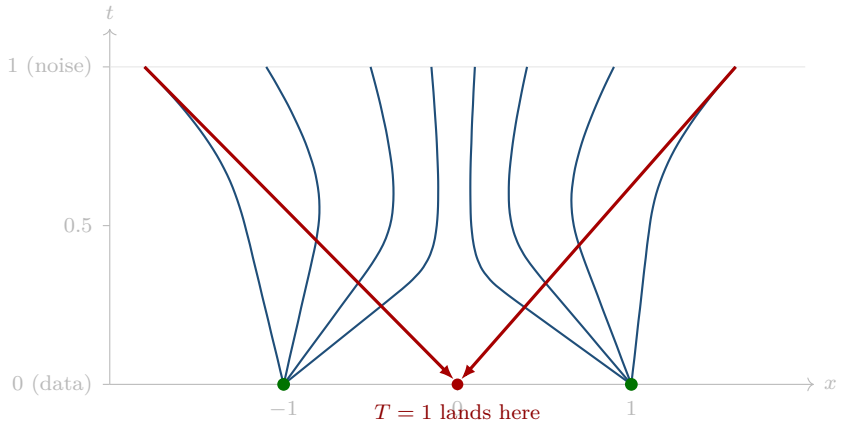


Figure 7.17. Euler sampling with the exact ideal velocity, for data at -1 and $+1$ (green), computed for this book. Blue: $T = 50$ steps from eight starting noises; every path ends on a data point. The paths are *curved*: they first drift towards the middle, because early on the best guess is the average, then split towards whichever point they are nearer to. Red: with $T = 1$, a single straight jump takes every start to 0, the average, which is not a data point.

How to read it: Time runs downwards, from noise at the top to data at the bottom. Where a blue path bends, a large Euler step would cut the corner, and cutting it lands between the two data points. The more a learned flow bends, the more steps it needs.

Computationally, if paths are exactly straight, the ODE can be simulated *exactly without time discretization*—one step suffices, and $T = 1$ Euler sampling is exact. Hence straight-path flows *bridge the gap between one-step and continuous-time models*: the same model behaves like a GAN-style one-shot generator at $T = 1$ and a classical continuous normalizing flow at large T.

Why the first model is not perfectly straight — and "reflow". In training, each data point $\boldsymbol{x}$ is paired with a *random* noise point $\boldsymbol{z}$, and the straight lines between pairs *cross*. At a crossing the network sees several targets pointing different ways, and (because it is trained with squared error) learns their average. Following an averaged direction bends the path — towards one line's destination, then another's. In high dimensions crossings are everywhere, so the trained field is only *roughly* straight.

Liu et al.'s fix, *reflow* (Liu et al., 2023), removes the crossings:

1. Use the first model to turn each noise point $\boldsymbol{z}$ into a generated image $\boldsymbol{x}_{\text{gen}}$. This gives new pairs $(\boldsymbol{x}_{\text{gen}}, \boldsymbol{z})$.
2. Train a fresh model on those pairs.

The new pairs come from the model's own flow, whose trajectories never cross each other, so the straight lines between the new pairs cross far less than random pairings do. The retrained field is almost exactly straight, and sampling in very few steps — even one — becomes nearly exact. Figure 7.18 shows the whole story on the two-point data of Figure 7.17, from the same eight starting noises.

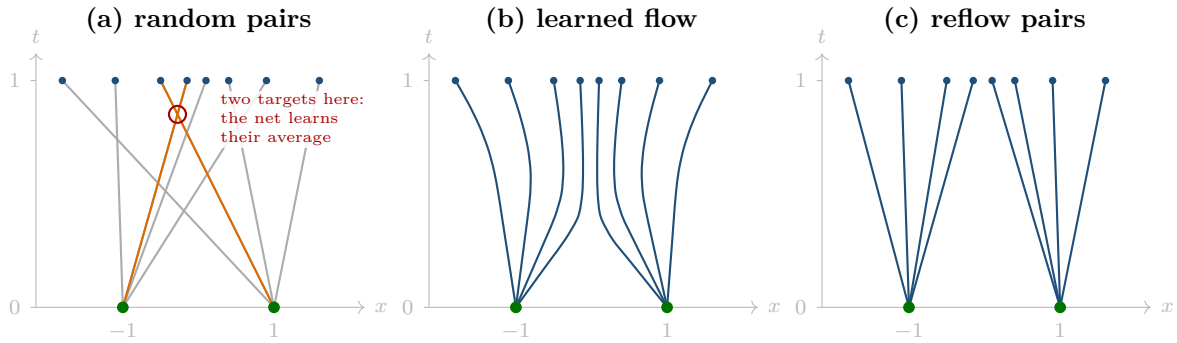


Figure 7.18. Rectified flow and reflow on the two-point data (±1, green) from eight noise samples (blue dots at $t = 1$), computed for this book with the exact ideal velocity. (a) First-round training pairs each noise point with a *random* data point; the straight training lines cross (one crossing circled: the two orange lines ask for different velocities at the same spot). (b) The field the network learns averages those targets. Its trajectories never cross, but they bend: they drift to the middle, then split. (c) Reflow pairs each noise point with the data point that flow (b) actually delivered it to, and trains on straight lines between those pairs. These lines do not cross, so no averaging is needed and the new field is straight. (On real data reflow removes most crossings, not always all.)

How to read it: Compare the endpoints of (b) and (c): identical. Reflow does not change which *noise goes to* which *data point; it changes only the* route, *from a bent one to a straight one. Only (c) can be sampled in one Euler step.*

Worked Example — where the bend comes from, and how reflow removes it

Use the data of Figure 7.18 (−1 or +1) and follow one noise point, $z = 0.1$.

First round (random pairs). During training, noise near 0.1 was paired with −1 half the time and with +1 the other half. So at $t = 1$ the network has seen two contradictory targets at this spot and learns their average: its best guess of the destination is $\mathbb{E}[x \mid x_1 = 0.1] = 0$, the midpoint, and the velocity is $v = (0.1 - 0)/1 = 0.1$. The sampler therefore starts moving *towards* 0, not towards either data point. Only later, as t shrinks and the point is visibly on the positive side, does the guess tip to +1 and the path turn. That turn is the bend in panel (b). One big Euler step would use only the first velocity and land at $0.1 - 0.1 = 0$, which is not a data point. With 50 small steps the path follows the bend and ends at +1.

Reflow (pairs from the first model). Panel (b) delivered $z = 0.1$ to +1, so reflow trains on the pair $(z, x) = (0.1, +1)$. The same holds for every positive z, and every negative z is paired with −1. Now every training line through this spot goes to +1. There is nothing to average, and the velocity at $t = 1$ is already the full straight-line one, $v = z - x = 0.1 - 1 = -0.9$. One Euler step gives $0.1 - (-0.9) = 1.0$: exactly on the data point, with no bend at all.

Why are there no conflicts any more? A reflow line from a positive z sits at $x_t = (1 - t) + tz > 1 - t$, and one from a negative z sits below $-(1 - t)$. The two families are

separated by a gap at every $t < 1$, so no two lines ever pass through the same point, and the averaged field equals the straight lines themselves.

In plain words — the one-sentence answer

The first model's paths bend because, at the start, the model does not know *which* data point a given noise sample is headed for (random pairing taught it both answers), so it hedges towards the middle and turns once it can tell. Reflow gives each noise sample one fixed destination — the one the first model already chose — so the new model knows the destination from the first step and can head straight there.

Everyday analogy — a taxi that does not know the address

A taxi told only "one of two addresses, east or west" sensibly drives towards the middle of town, and turns once it learns which. That is the first-round flow. Reflow is the passenger giving the address at the kerb: the same trip, the same destination, but a straight route.

Where the analogy breaks. The flow never "learns" the address mid-trip from outside; the destination becomes clear only because the moving point drifts to one side, where one data point is more likely.

In practice one training pass of Eq. (7.20) yields approximately straight paths (approximately constant velocity along each trajectory), and the "reflow" procedure of Liu et al. (2023)—retrain on pairs generated by the current model—straightens them further at the cost of extra training rounds. Figure 7.19 makes the payoff of that straightening visible directly.

Simplified intuition vs. more precise view

Simplified story. Rectified flow makes paths straight, so you can sample in very few steps, even one. **More precise view.** One pass of flow-matching training with independent couplings does *not* give exactly straight paths: the optimal field straightens the marginal trajectories, but residual curvature remains, so few-step Euler sampling still accumulates error. True one-step behavior emerges only after reflow (or distillation), and "straight" is a property to iterate toward, not a property you get for free. Straightness should be thought of as a spectrum that controls how large a step size $1/T$ you can afford.

What changed — from DDPM's Gaussian chain to rectified flow

Before (DDPM). A forward process defined by adding Gaussian noise in many small steps; a network trained to undo one step, conditioned on a discrete timestep from a fixed schedule.

The change. Replace the noising chain with one straight-line interpolant $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$, defined at every real $t \in [0, 1]$; train the network to predict that line's velocity instead of a schedule-specific noise vector.

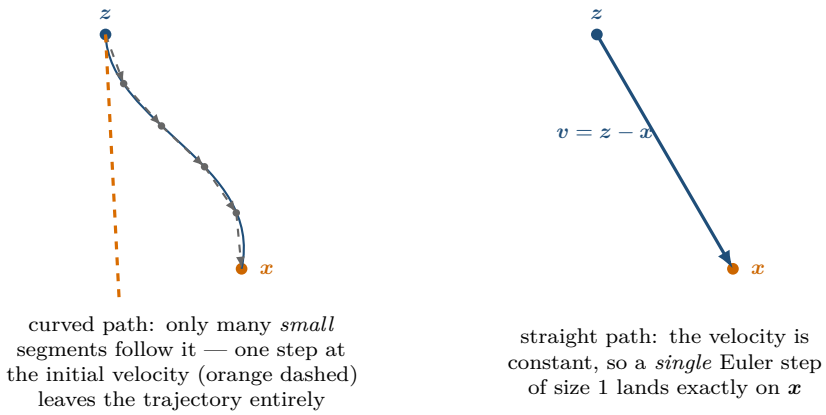


Figure 7.19. Straight vs. curved transport paths (graphic after Fjelde et al., shown as Lecture 05, slide 35, and redrawn here). A curved flow (left) can only be simulated by many small Euler chords; a single large step follows the initial velocity and departs the trajectory. A rectified path (right) has constant velocity $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$ and is simulated *exactly* in one step.

How to read it: Count segments, not distance. Both panels move the same endpoint $\boldsymbol{z}$ to the same data point $\boldsymbol{x}$ — what differs is how many discretization steps you must trust. That count is the practical difference between DDPM's hundreds of sampling steps and rectified flow's tens (or one).

Source: Graphic after Fjelde et al., via Lecture 05, slide 35.

What it cost. Nothing in training simplicity — if anything, the loop is shorter. What it bought: a path that is already close to straight by construction, so Euler sampling needs a fraction of DDPM's steps, and (as Section 7.6 shows next) the same trained object is, without any further argument, a continuous normalizing flow.

Key takeaway

Rectified flow does not add machinery to diffusion — it removes it. There is no noise schedule $\bar{\alpha}_t$, no distinction between $\boldsymbol{\varepsilon}$-, $\boldsymbol{x}_0$-, and $\boldsymbol{v}$-parameterizations, no ELBO. There is one interpolation, one velocity, one squared error, and one ODE to integrate. Everything this chapter proved about DDPM by way of careful algebra, rectified flow gets by choosing a simpler object to begin with.

7.6 Continuous normalizing flows and the density question

Step 2 of Section 7.5.1 promised that the velocity field we just trained is a continuous normalizing flow, and that this buys something the sampler alone does not: an exact likelihood. Slides 36–37 supply the machinery. This section cashes that promise and then explains why we still train with flow matching rather than with the likelihood.

7.6.1 An ODE in disguise

Sampling runs the rectified-flow clock backwards, from $t = 1$ (noise) to $t = 0$ (data). To match the usual flow notation, where the map goes *from* noise *to* data as its clock runs forward, give generation its own clock $s = 1 - t$. Along a trajectory $\boldsymbol{z}_s$, reversing time flips the sign of the velocity, so the generative ODE is

$$\frac{d\boldsymbol{z}_s}{ds} = \boldsymbol{g}(\boldsymbol{z}_s, s) \;:=\; -f_{\boldsymbol{\theta}}(\boldsymbol{z}_s,\, 1-s), \qquad \boldsymbol{z}_0 \sim p_{\text{noise}}, \qquad \boldsymbol{x} = \boldsymbol{z}_1 = \boldsymbol{z}_0 + \int_0^1 \boldsymbol{g}(\boldsymbol{z}_s, s)\, ds.$$

(The Euler sampler of Eq. (7.21) is exactly this integral, cut into T steps of size $1/T$.) Solving an ODE from a starting point is an *invertible* map between noise and data — run it backwards to invert, and trajectories never cross (Section 7.5.1). That is exactly what a normalizing flow is, except the discrete stack of invertible layers has become continuous-time dynamics: the *continuous normalizing flow (CNF)* of neural-ODE fame (Chen et al., 2018).

Can we then evaluate $p(\boldsymbol{x})$, settling the taxonomy? Think of the probability as a fluid carried by the flow. Where the flow spreads out (positive divergence), the same mass covers more volume and the density drops; where it squeezes together, the density rises. So the log-density of a moving point changes at exactly minus the local rate of expansion — the divergence of the velocity field, which is the trace of its Jacobian. Integrating along the trajectory gives the *instantaneous change of variables* (Chen et al., 2018):

$$\log p(\boldsymbol{x}) \;=\; \log p_{\text{noise}}(\boldsymbol{z}_0) \;-\; \int_0^1 \operatorname{tr}\!\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{z}_s}\right] ds, \tag{7.24}$$

where $\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{z}_s}$ is the Jacobian of the velocity field at the running state and tr its trace. Compare Eq. (7.1): the discrete sum of log-determinants has become an integral of traces, because one tiny Euler step is the map $\boldsymbol{z} \mapsto \boldsymbol{z} + \boldsymbol{g}\, ds$, and $\log\det(\boldsymbol{I} + \boldsymbol{J}\, ds) \approx \operatorname{tr}(\boldsymbol{J})\, ds$ for infinitesimal ds.

Worked Example — a flow that shrinks everything by half

Take the one-dimensional field $g(z, s) = -z \ln 2$. Its solution is $z_s = z_0\, 2^{-s}$, so after one unit of time every point has moved halfway to the origin: $x = z_0/2$. A distribution squeezed into half the width must be twice as tall, so $p(x) = 2\, p_{\text{noise}}(2x)$.

Now check Eq. (7.24). The "Jacobian" is $\partial g/\partial z = -\ln 2$, constant, so the integral is $-\ln 2$ and $\log p(x) = \log p_{\text{noise}}(z_0) + \ln 2$. That is a factor of 2 in density: the squeeze is counted exactly, without ever writing down the map. For a network $\boldsymbol{g}$ the trace changes along the path and must be integrated numerically, but the bookkeeping is the same.

Key Concept — Instantaneous change of variables

For a sample moving under $\dot{\boldsymbol{z}}_t = f(\boldsymbol{z}_t, t)$, its log-density evolves as $\frac{d}{dt}\log p_t(\boldsymbol{z}_t) = -\operatorname{tr}[\partial f/\partial \boldsymbol{z}_t]$. Evaluating $\log p(\boldsymbol{x})$ requires one ODE integration of the state *backward*

to noise plus one scalar per time step—but the trace of a $d \times d$ Jacobian costs d vector–Jacobian products per step, which for images ($d \sim 10^5$–10^6) is prohibitive.

7.6.2 The Hutchinson trick

The problem. Equation (7.24) needs the *trace* of a Jacobian — the sum of its diagonal entries. For an image that Jacobian is a 200,000 × 200,000 matrix. We cannot afford to build it, let alone read off its diagonal.

The trick, in one sentence. Multiply the matrix by a random vector of ±1 entries on both sides; on average, the answer is the trace.

Worked Example — the trace trick on a 2×2 matrix

Take $\boldsymbol{A} = \left[\begin{smallmatrix} 2 & 1 \\ 3 & 4 \end{smallmatrix}\right]$, whose trace is $2+4=6$. Try all four random sign vectors $\boldsymbol{\varepsilon}$ and compute $\boldsymbol{\varepsilon}^\top \boldsymbol{A}\boldsymbol{\varepsilon}$:

$$(1,1)\colon\ 10 \qquad (1,-1)\colon\ 2 \qquad (-1,1)\colon\ 2 \qquad (-1,-1)\colon\ 10.$$

No single answer equals 6, but their average is $(10+2+2+10)/4 = 6$ exactly. The off-diagonal entries 1 and 3 appear with a + sign in half the cases and a − sign in the other half, so they cancel; the diagonal entries always appear with a + sign, so they survive.

That cancellation is the whole proof. FFJORD (Grathwohl et al., 2019) used this estimator (Hutchinson, 1990), stated in general as follows: for any matrix $\boldsymbol{A}$ and $\boldsymbol{\varepsilon}$ with independent, zero-mean, unit-variance entries (e.g. Rademacher ±1 or standard Gaussian),

$$\operatorname{tr} \boldsymbol{A} \;=\; \mathbb{E}_{\boldsymbol{\varepsilon}}\left[\, \boldsymbol{\varepsilon}^\top \boldsymbol{A}\, \boldsymbol{\varepsilon} \,\right],$$

because $\mathbb{E}[\varepsilon_i \varepsilon_j]$ is 1 when $i = j$ and 0 otherwise. Written out entry by entry,

$$\mathbb{E}\left[\sum_{i,j} \varepsilon_i A_{ij} \varepsilon_j\right] = \sum_{i,j} A_{ij}\, \mathbb{E}[\varepsilon_i \varepsilon_j] = \sum_i A_{ii} \;=\; \operatorname{tr} \boldsymbol{A}.$$

Why it is cheap. We never build $\boldsymbol{A}$. We only need $\boldsymbol{\varepsilon}^\top \boldsymbol{A}$, and automatic differentiation computes a "vector times Jacobian" product for about the cost of one ordinary gradient: form the number $g = f(\boldsymbol{z}_t, t)^\top \boldsymbol{\varepsilon}$, backpropagate once, and dot the result with $\boldsymbol{\varepsilon}$. In practice FFJORD uses a *single* random $\boldsymbol{\varepsilon}$ per step rather than averaging many. That single estimate is noisy — but training is already noisy from minibatching, so one more source of noise averages out the same way.

So: yes, $p(\boldsymbol{x})$ can be evaluated for a rectified flow, and that is why the closing taxonomy moves flow matching to the explicit-density side (Figure 7.1).

7.6.3 Why not train the CNF by maximum likelihood?

With an exact $\log p(\boldsymbol{x})$ in hand, the obvious training rule is the one every flow in Chapter 6 used: maximize it. That is how CNFs were trained before flow matching (Chen et al., 2018; Grathwohl et al., 2019). It works, but compare what one training example costs:

	Maximum likelihood (FFJORD-style)	**Flow matching / rectified flow**
What is computed	$\log p(\boldsymbol{x})$ via Eq. (7.24): integrate the ODE from $\boldsymbol{x}$ back to noise	the per-pair velocity $\boldsymbol{z} - \boldsymbol{x}$ at one random t
Network calls	one per solver step (tens to hundreds), each with a Hutchinson trace	**one**
Backpropagation	through the whole ODE solve (or an adjoint ODE of the same length)	through one forward pass
Solver error	enters the loss	none during training
Straightness of paths	not controlled — any field with high likelihood is fine, even a very curved one	encouraged directly by the choice of straight per-pair paths

With a solver that takes a hundred steps, a likelihood-trained CNF spends on the order of a hundred network passes per example where flow matching spends one. Worse, nothing in the likelihood rewards straight trajectories, so the trained model may also be slow to *sample*.

Flow matching's answer is to split the two jobs. *Train* by regression, which is cheap and shapes the paths. *Evaluate* with Eq. (7.24) only when a likelihood is actually needed. It is the same trained network either way, because rows 2–4 of Figure 7.13 all learn the same object.

In plain words — the whole section in four sentences

A normalizing flow gives exact likelihoods but forces every layer to be invertible. Letting a network define a velocity instead (a CNF) makes any network invertible and turns determinants into traces, but training by likelihood means simulating the ODE at every step. Flow matching skips the simulation by regressing on velocities we can write down for single noise–data pairs. Rectified flow picks straight lines for those pairs, so the learned field is nearly straight and sampling needs only a few steps.

Failure Modes

Sampling schedule mismatch. The marginal (7.4) trains the network on every $\bar{\alpha}_t$ the schedule touches, but each sampling trajectory visits a specific timetable of t values. A model trained on a given schedule and sampled with a much coarser one inherits an exposure bias at every skipped step, and one trained on a wrong-pinned schedule —for instance, $\bar{\alpha}_T$ bounded away from 0 so that $\boldsymbol{x}_T \not\approx \boldsymbol{\varepsilon}$—hands the sampler inputs that it was never trained to denoise.

Sampling cost. Every sample needs T network evaluations ($T \approx 50$ typifies rectified flow; DDPM-family models historically used hundreds to a thousand steps). GANs and VAEs need one. Fewer than optimal steps bends curved trajectories and blurs outputs; making few-step sampling exact (reflow, distillation) costs extra training, not extra theory.
Exposure bias. Training feeds the network $\boldsymbol{x}_t$ built by the exact interpolation of Eq. (7.18); sampling feeds it its own previous Euler output $\boldsymbol{x}_{t-1} - f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}, t-1)/T$, which is slightly off-interpolation. The distributional shift between train-time and sample-time inputs compounds over T steps, and is worst exactly where discretization is already worst (few steps, curved regions).
Poor density estimates in practice. Equation (7.24) is exact only with the true field, exact integration, and infinite Hutchinson samples. With a learned $f_{\boldsymbol{\theta}}$, Euler integration, and one Rademacher draw, reported CNF and diffusion likelihoods carry visible Monte-Carlo noise and truncation bias, and diffusion log-likelihoods have often lagged the best autoregressive models on images. Treat "diffusion is explicit-density" as a theoretical fact about the ODE, not a practical reason to score data with it.

7.7 Conditioning and classifier-free guidance

7.7.1 Conditioning is one argument

Conditioning: "generate a *cat*". So far the model generated from the whole data distribution. Asking for a particular kind of output is almost embarrassingly easy (slides 38–39): give the model the label $\boldsymbol{y}$ as an extra input, both in training and in sampling. The slides use triangles and squares as the two classes. In code the changes are tiny:

- training: `for x in dataset` becomes `for (x, y) in dataset`, and `model(xt, t)` becomes `model(xt, y, t)`;
- sampling: choose `y = user_input()`.

The maths does not change: the network now learns the average velocity *given* $\boldsymbol{y}$, $\mathbb{E}[\boldsymbol{z} - \boldsymbol{x} \mid \boldsymbol{x}_t, t, \boldsymbol{y}]$, and following it lands on the distribution of that class. Text-to-image systems do exactly this, with $\boldsymbol{y}$ an encoding of the text prompt.

7.7.2 Classifier-free guidance

Key Concept — Classifier-free guidance (CFG)

Classifier-free guidance (Ho and Salimans, 2022) strengthens conditioning at sampling time without any external classifier. Train one model to be both conditional and unconditional by randomly replacing $\boldsymbol{y}$ with a null token $\boldsymbol{y}_\varnothing$ (the slides drop $\boldsymbol{y}$ with probability 0.5; the original paper found 0.1–0.2 works well; Listing 7.3). At sampling, form $v^{\text{cfg}} =$

$(1+w)\,v^{\boldsymbol{y}} - w\,v^{\varnothing}$ from the two predictions and step along it, where $w \geq 0$ is the guidance weight.

Model at a glance: Classifier-Free Guidance (CFG) Ho & Salimans, 2022

In one line	Train one model to answer both "generate $\boldsymbol{x}$" and "generate $\boldsymbol{x}$ given $\boldsymbol{y}$"; at sampling time, extrapolate past the conditional answer, away from the unconditional one.
Problem it solves	Plain conditional training underuses $\boldsymbol{y}$: the easiest high-probability sample under $p(\boldsymbol{x} \mid \boldsymbol{y})$ often only weakly reflects the condition, because conditional distributions are broad.
Key intuition	The difference between the conditional and unconditional predictions is the direction attributable to the condition alone; scaling that difference up is a free knob on how strongly the condition is obeyed.
Inputs	A noisy $\boldsymbol{x}_t$, timestep t, condition $\boldsymbol{y}$ (real or the null token $\boldsymbol{y}_\varnothing$), and a user-chosen guidance weight $w \geq 0$.
Architecture	No new network. One model handles both branches; the null token is just another value of $\boldsymbol{y}$.
Loss	Identical to the base model's loss, with $\boldsymbol{y}$ replaced by $\boldsymbol{y}_\varnothing$ on a random fraction of examples (50% on the slides, 10–20% in the original paper) (Listing 7.3).
Sampling	Two forward passes per step, $v^{\text{cfg}} = (1+w)v^{\boldsymbol{y}} - w\,v^{\varnothing}$ (Eq. (7.25), Listing 7.4).
What you keep	A single conditional generator whose adherence to $\boldsymbol{y}$ is a sampling-time dial, not something fixed once at training time.
Strengths	No separate classifier to train or keep noise-robust; a continuous, interpretable knob from w=0 (unguided) upward.
Weaknesses	Doubles sampling compute; large w trades diversity and natural image statistics for on-condition fidelity, with visible saturation artefacts.
Leads to	The default steering mechanism in essentially every production text-to-image and text-to-video diffusion system that follows.

```
for (x, y) in dataset:
    if random.random() < 0.5:
        y = y_null          # same net learns unconditional task
    z = torch.randn_like(x); t = random.uniform(0, 1)
    xt = (1 - t) * x + t * z
    v = model(xt, y, t)
    loss = (z - x - v).square().sum()
```

Listing 7.3. CFG training: randomly drop the condition (Lecture 05, slide 40).

The slide animation (slides 41–44) motivates the formula geometrically (Figure 7.20). At a noisy $\boldsymbol{x}_t$, the unconditional velocity $v^{\varnothing} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{y}_\varnothing, t)$ points toward *all* of p_{data}; the conditional one $v^{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{y}, t)$ points toward the conditional mode $p(\boldsymbol{x} \mid \boldsymbol{y})$. Their difference

$v^{\boldsymbol{y}} - v^{\varnothing}$ is the component of the motion attributable to the condition—the direction "more toward this class than toward data in general." Extrapolating along it,

$$v^{\text{cfg}} = v^{\boldsymbol{y}} + w\,(v^{\boldsymbol{y}} - v^{\varnothing}) = (1+w)\,v^{\boldsymbol{y}} - w\,v^{\varnothing}, \tag{7.25}$$

yields a velocity that overshoots $v^{\boldsymbol{y}}$ in the stylized direction the user asked for; step along it (Listing 7.4). At $w = 0$ this reduces to plain conditional sampling.

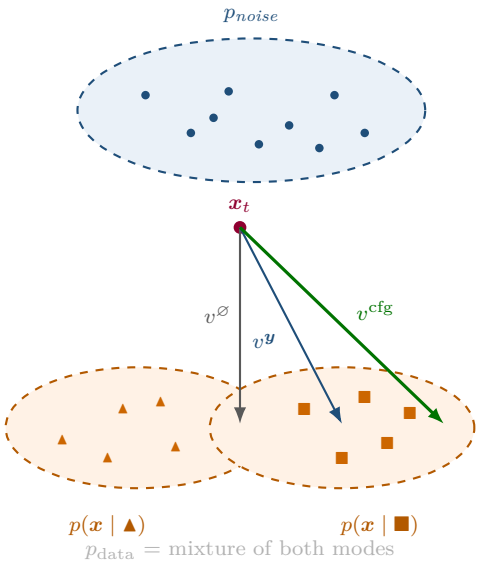


Figure 7.20. The geometry of classifier-free guidance (Lecture 05, slides 41–44). The data distribution is a mixture of two modes (▲ and ■ classes); the point $\boldsymbol{x}_t$ sits between the noise blob and the data. From there the unconditional velocity $v^{\varnothing}$ points at the centroid of *both* modes — the safest average answer; the conditional velocity $v^{\boldsymbol{y}}$ points at the requested mode; and the guided velocity $v^{\text{cfg}} = (1+w)v^{\boldsymbol{y}} - w\,v^{\varnothing}$ ($w = 1$ drawn) continues *past* the conditional target along the difference direction $v^{\boldsymbol{y}} - v^{\varnothing}$, which is exactly the component the condition contributed.

How to read it: All three arrows start at the same $\boldsymbol{x}_t$*; they are not different trajectories, only three candidate first steps. Unlike a sliding scale on* $\boldsymbol{y}$*'s influence inside the network, the extrapolation happens in the space of* outputs *of a network already trained — guidance is a sampler-side trick, nothing is retrained. Each sampling step redraws this triangle at the new point.*

Source: Lecture 05, slides 41–44 (geometry redrawn).

A study-session picture fixes the extrapolation direction: ask an artist for a pink elephant. The unconditional velocity is "draw an elephant"—the generic animal. The conditional velocity is "draw a pink elephant"—same content, plus the attribute. The difference $v^{\boldsymbol{y}} - v^{\varnothing}$ isolates the *pinkness direction*; CFG scales that difference up and adds it back on top of conditional generation, so at $w > 0$ the elephants get unmistakably pink, at very large w garishly oversaturated. Nothing in the formula conditions on a classifier: the two velocities share one network and one arithmetic step.

Why does a plain conditional model underuse $\boldsymbol{y}$ in the first place? Because conditional data distributions $p(\boldsymbol{x} \mid \boldsymbol{y})$ are broad: the easiest high-probability conditional sample often only weakly depends on $\boldsymbol{y}$. Guidance fixes the model's laziness not by reweighting training but by *extrapolating in the space of already-trained predictions*—a sampling-time intervention whose cost is one extra forward pass per step.

```
vy = model(sample, y, t)
v0 = model(sample, y_null, t)
v  = (1 + w) * vy - w * v0
sample = sample - v / num_steps
```

Listing 7.4. CFG sampling: two evaluations per step (Lecture 05, slide 45).

What distribution does Eq. (7.25) actually head for? Guided sampling tilts toward $p(\boldsymbol{x} \mid \boldsymbol{y})^{1+w}\, p(\boldsymbol{x})^{-w}$—a sharpening of the conditional distribution, upweighting regions whose conditional probability exceeds their unconditional probability (Ho and Salimans, 2022). This explains both the felt effect (at large w: strikingly on-condition outputs, less diversity, and characteristic contrast/saturation artefacts) and the historical context: Dhariwal and Nichol (2021) had shown that diffusion beats GANs on image synthesis once *classifier* guidance was applied; classifier-free guidance (Ho and Salimans, 2022) recovered the effect without training a noisy-input classifier. Two honesty notes carry over verbatim from the slides: CFG is "used everywhere in practice—very important for high-quality outputs," and Eq. (7.25) *doubles the cost of sampling*, since every step now requires the $v^{\boldsymbol{y}}$ and $v^{\varnothing}$ evaluations. Some of the cost can be recovered by batching the two evaluations into one forward pass, but the compute is real.

Common Confusion

Diffusion vs. flow matching vs. score-based SDEs. These are three presentations of one family, not three competing families. *DDPM-style diffusion* (Ho et al., 2020) discretizes the noising into a Markov chain and frames learning as ELBO-optimization, which simplifies to predicting the added noise. *Score-based SDE formulations* (Song et al., 2021b) take the chain's continuous-time limit: a forward stochastic differential equation, a reverse SDE driven by the score $\nabla_{\boldsymbol{x}} \log p_t(\boldsymbol{x})$, and an associated deterministic *probability-flow ODE* with the same marginals. *Flow matching / rectified flow* (Lipman et al., 2023; Liu et al., 2023) start from the ODE, choose the interpolant $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$ directly, and train by velocity regression—no SDE, no ELBO. All three routes learn, in the end, vector fields that point from noised states back toward the data (this is the DAE–score connection of Section 7.2 writ continuous), and they can be translated into one another with schedule and parameterization bookkeeping. They differ in what they make easy: SDEs give principled stochastic samplers, rectified flows give straight paths and cheap sampling, DDPM gives the simplest discrete derivation. Deterministic ODE

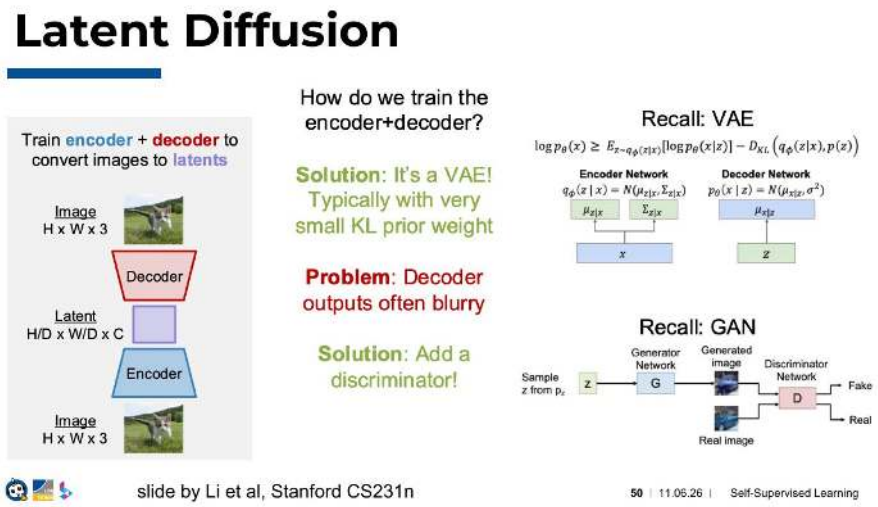


Figure 7.21. Latent diffusion training after Rombach et al. (2022), from Lecture 05: train the encoder+decoder with a VAE at *very small KL prior weight*, but VAE decoder outputs are often blurry—so add a discriminator. Panels: VAE recall $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}_{z|x}, \boldsymbol{\Sigma}_{z|x})$, $p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) = \mathcal{N}(\boldsymbol{\mu}_{x|z}, \sigma^2)$ and GAN recall (sample $\boldsymbol{z}$ from p_z through generator G; discriminator D separates generated from real).

samplers such as DDIM (Song et al., 2021a), and faster solvers built on the same idea, cut DDPM sampling to tens of steps.

7.8 Latent diffusion: composing everything

Pixel-space diffusion at high resolution is expensive: the velocity network must handle textures it could obtain "for free" from a good image decoder. The latent diffusion model (LDM) of Rombach et al. (2022)—the architecture behind Stable Diffusion—splits generation into two learned stages (slides 46–52, Figure 7.21).

Stage 1: learn a latent space. Train an encoder–decoder pair that compresses an image $H \times W \times 3$ into a latent $\frac{H}{D} \times \frac{W}{D} \times C$ (downsampling factor $D \approx 8$ in the standard configuration) and reconstructs the image from it.

Stage 2: diffuse in latent space. Freeze the encoder. Train a flow matching / diffusion model that removes noise *from latents*: noisy latent → denoising network → clean latent. At inference, sample a random latent, iteratively denoise it, and run the decoder once to obtain the image. All the diffusion machinery of this chapter applies verbatim after replacing "image" with "latent," at a small multiple of the compute.

How is the autoencoder trained? It is a VAE (Chapter 4 gives the derivation; slide 50 "recalls" the panel): maximize the ELBO

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \;\geq\; \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\big] \;-\; D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p(\boldsymbol{z})\big),$$

with $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}}, \boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x}})$ and a Gaussian decoder, *but with a very small weight on the KL prior term*—the slides state $\beta \approx 10^{-6}$ explicitly. Why any KL at all? The stage-2 diffusion model needs the aggregate distribution of latent codes $q(\boldsymbol{z}) = \mathbb{E}_{\boldsymbol{x}}[q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})]$ to be well matched to its own noise front $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$; a latent space with arbitrary gaps or a scale far from unit Gaussian would leave regions of stage-2's own input distribution with no training signal. The small-β compromise is the lecture's point: enough regularization to keep latents roughly standard and connected, far too little to impose the isotropy that would wash out fine detail. *Problem:* even with that compromise, VAE decoders output blurry reconstructions, because the Gaussian decoder loss is minimized by the conditional mean over remaining uncertainty—an average of the possible sharp images, i.e. a blur. *Solution: add a discriminator*—train the decoder additionally with an adversarial loss (Chapter 6's GAN machinery) so it must produce images that *look* real, not merely likely under a conditional Gaussian; the total stage-1 objective reads $\mathcal{L}_{\mathrm{rec}} + \beta\,\mathcal{L}_{D_{\mathrm{KL}}} + \lambda\,\mathcal{L}_{\mathrm{GAN}}$ (slide 51). The highlighted takeaway on slide 52 deserves to be set off:

Latent diffusion is a composition of VAEs + GANs + flows + diffusion.

Each parent has a precise job: the VAE provides the latent space, the GAN loss provides sharp decoding, the flow view provides the generative model in latent space, and the chapter's denoising machinery trains it. Nothing new is invented; Part II so far is reassembled.

Paper Analysis: Latent Diffusion Models (Rombach et al., 2022)

Problem. Pixel-space diffusion wastes capacity and compute modeling perceptually irrelevant detail at full resolution (512^2 and up). **Prior work.** Pixel DDPMs with classifier guidance (Ho et al., 2020; Dhariwal and Nichol, 2021); VQ-VAE-style discrete tokenizers plus autoregressive transformers (expensive in their own right).
Insight. Split perception from generation: a convolutional autoencoder removes high-frequency redundancy; diffusion need only model the semantic remainder in a space $\sim 48\times$ smaller.
Method. Two stages as above; a KL-regularized VAE with an added adversarial (GAN) loss on the decoder; a U-Net (Ronneberger et al., 2015) denoiser with cross-attention for conditioning $\boldsymbol{y}$ (text, layout, class).
Result. Competitive or state-of-the-art FID-quality image synthesis at a fraction of pixel-diffusion compute; the released model (Stable Diffusion) became the de facto open text-to-image system.
Limitations. Stage-1 reconstruction bounds achievable fidelity; the tiny-KL/adversarial autoencoder is delicate to balance; inherits every failure mode in this chapter's box, plus text–image alignment problems no component owns. **What to remember.** Diffusion on latents, not pixels; and recognize the system as the Part II generative zoo, composed.

7.9 Autoregressive interlude: the tractable corner

The "exact probability" family. Slides 53–58 cover the corner of the taxonomy where the model writes down an explicit formula $p(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{W})$ and fits it by maximum likelihood: choose the weights that make the training data as probable as possible,

$$\boldsymbol{W}^* = \arg\max_{\boldsymbol{W}} \prod_{i=1}^{N} p(\boldsymbol{x}^{(i)}) \;=\; \arg\max_{\boldsymbol{W}} \sum_{i=1}^{N} \log f(\boldsymbol{x}^{(i)}, \boldsymbol{W}).$$

No adversary and no bound — just an exact score, climbed by gradient ascent. Taking the log does two jobs. *Numerically*, multiplying thousands of small probabilities gives a number too small for a computer to store, while adding their logs does not. *For optimisation*, the log never changes which $\boldsymbol{W}$ is best, and it turns one product into a sum of per-example terms, which is what gradient descent handles well.

Autoregressive models, briefly. Any probability over a sequence can be split into a chain of "next item given everything so far" predictions:

$$p(\boldsymbol{x}) = p(x_1)\, p(x_2 \mid x_1)\, p(x_3 \mid x_1, x_2) \cdots \;=\; \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}),$$

and a network (an RNN or a Transformer) learns each "next item" step. This is exactly how language models work.

Training is fast. The whole sequence is known from the dataset, so all T predictions can be checked at once.

Generation is slow. Item t cannot be produced until item $t-1$ exists. PixelRNN (van den Oord et al., 2016) applies this to images by reading the pixels in scan order, like text, and predicting each colour value as one of 256 classes. A 256×256 colour image then needs about 200,000 predictions made *one after another* — which is why this family, despite excellent likelihoods, is rarely used to generate images.

Failure Modes

Autoregressive images are too expensive. A 1024×1024 image is a sequence of 3 million subpixels; training and (worse) sampling scale linearly in that number, with no parallelism at generation time, since each subpixel conditions on all previous ones. The slide's foreshadowed escape (green box): model the image as a sequence of *tiles*, not subpixels—i.e. patch or VQ-token sequences. That is exactly the autoregressive side of the latent-diffusion philosophy (model a compressed sequence), and Chapter 5's patch embedding is the vision world's version of the same move.

Contrast drawn: autoregressive models give exact likelihoods and trivially stable training but serial, slow sampling; diffusion models trade exact densities for parallel, iterative sampling; latent models buy efficiency by changing the space first. Every corner of the taxonomy is a different point in this likelihood–speed–stability trade-off.

7.10 Applications back in SSL territory

7.10.1 DIFT: semantic correspondence falls out of diffusion training

Is anything useful inside a diffusion model? Diffusion models are trained without labels — just images and added noise. So the self-supervised question is natural: do their internal features know anything? Two observations say yes (slides 59–60). First, the DIFT paper (Tang et al., 2023) finds that matching points between two images ("this is the same beak in both photos") emerges inside diffusion models with no supervision at all. Second, there is a striking figure: ask the model for its best guess of the clean image at several noise levels, $t \in \{1, 51, 261, 501\}$.

- At **high noise**, only a rough silhouette survives, so the model must know *what* kind of thing is there ("a cat shape in the middle").
- At **medium noise**, it must know the *parts* (head, ears, tail — which requires knowing what cats look like).
- At **low noise**, only *texture* is left to restore (fur, whiskers).

So one network, trained at every noise level, is forced to learn every level of abstraction — from "what" down to "which hair". That is why its features carry meaning. (The high-noise guesses in the figure look very coarse partly for a technical reason: the model predicts noise, not the image, and converting back via Eq. (7.6) divides by the small number $\sqrt{\bar{\alpha}_t}$, which amplifies errors; Section 7.4.2 explains.)

Paper Analysis: DIFT: Emergent Correspondence from Image Diffusion (Tang et al., 2023)

Problem. Establishing semantic correspondence—which point on the duck's *wing* in image A is the wing point in image B—classically needs labeled keypoints or weak supervision.
Prior work. Handcrafted local descriptors; weakly-supervised correspondence networks; off-the-shelf self-supervised features (this book's Chapters 5 and 11) used as generic matchers.
Insight. A diffusion U-Net trained to denoise must internally represent object-part structure; that implicit knowledge can be *read off* as features.
Method. (1) Noisify the input to a chosen t and run the U-Net; (2) extract intermediate U-Net features and bilinearly upsample them to pixel-level resolution; (3) match pixels across images by cosine similarity of their DIFT features.
Results. DIFT outperforms weakly-supervised methods and is competitive with off-the-shelf features on semantic, geometric, and temporal correspondence (Tang et al., 2023); qualitatively, a source point on a bird transfers to the semantically matching point across instances, categories (penguin, hummingbird), and domains (paintings, graffiti)—and likewise dog breeds, polar bears versus paintings of bears, guitars versus cellos.
Limitations. Dependence on the choice of t and of U-Net block; the features are a by-product, not a trained endpoint, and are not yet a replacement for feature-backbone landmarks such as DINO (Chapter 11). **What to remember.** A self-supervised generative objective left semantic structure in the activations: representation learning for free, from generative training.

The recipe (slide 61) deserves emphasis because diffusion models are commonly U-Nets (Ronneberger et al., 2015), so intermediate features come with spatial maps at several resolutions: extract at one noise level, upsample bilinearly, match by cosine similarity. Downstream (slide 64) these correspondences feed keypoint and pose-estimation pipelines. From the SSL perspective, DIFT is Exhibit A that generative pretraining is also representation pretraining—the generative/discriminative wall the field once took for granted is soft. (Which block and which t to read is an empirical question the paper explores at length; mid-range t values, where abstraction is high but geometry is intact, tend to win.)

The whole method is three steps, and *none* of them is trained to find correspondences:

1. **Add a little noise** to the image and run it through the frozen diffusion U-Net (Ronneberger et al., 2015) — exactly the kind of input the network saw during training.

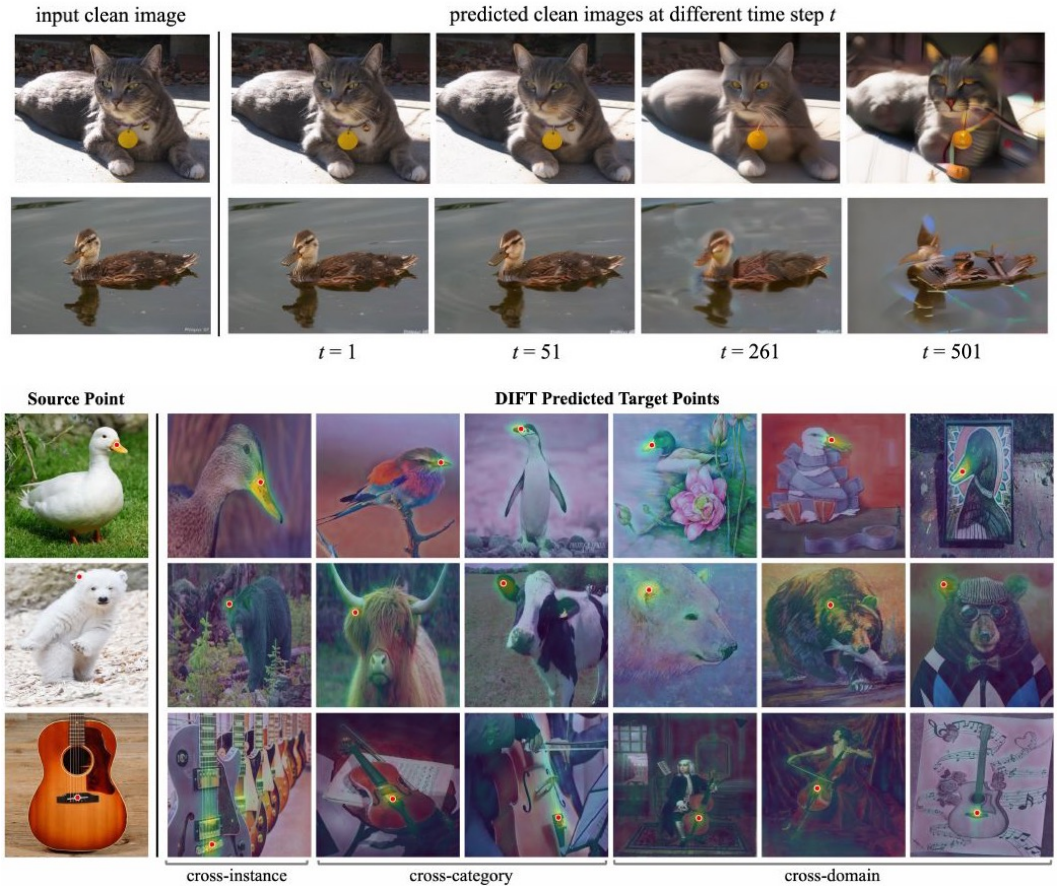


Figure 7.22. What diffusion training leaves inside the network (Lecture 05, slides 60 and 63). *Top:* the model's predicted clean image as the noise level grows, $t \in \{1, 51, 261, 501\}$. At $t = 1$ the predictions are nearly the input; by $t = 501$ a cat keeps only its pose and category (the fur pattern is gone), and a duck only its silhouette and "duck-ness" — texture dies first, semantics last, giving the coarse-to-fine abstraction hierarchy the features inherit. *Bottom:* DIFT correspondence maps (Tang et al., 2023). A source point is picked on the left column (on the duck's head, the bear cub's forehead, the guitar's body); the remaining columns show, as heat maps over target images, where the diffusion U-Net's features cosine-match that source point — transferring to other instances, to other categories with shared parts (penguin, highland cattle, cello), and across rendering domain to paintings and line drawings — with no correspondence supervision anywhere in training.

How to read it: Read the bottom grid columns left to right as "harder and harder questions": same category, different category, different domain. *The heat map staying localized on the semantically matching part across all three is the entire claim of the DIFT paper in one picture.*

Source: Lecture 05, slides 60 and 63.

2. **Grab an internal feature map** and resize it up to the image size, so every pixel gets its own feature vector.
3. **Match by nearest neighbour.** To find where pixel p of image A is in image B, pick the pixel of B whose feature vector is most similar (highest cosine similarity).

Step 1 is ordinary noise, step 2 is image resizing, step 3 is a lookup. So every bit of correspondence ability was already sitting inside the network, put there by nothing more than learning to remove noise.

7.10.2 Diffusion Policy: same machinery, deliberately *not* SSL

Diffusion Policy: denoising robot actions. The same machinery drives real robots (Chi et al., 2023). Diffusion Policy treats "what should the robot do next?" as a generation problem: the *condition* is what the robot sees (camera images, joint readings), and the *generated sample* is a short sequence of future actions.

Why not just predict one action with ordinary regression? Because human demonstrations are often *multimodal*. To push a T-shaped block, sometimes the demonstrator goes around the left, sometimes the right — both fine. A regressor averages the two and drives straight into the block. A generative model samples *one* of the valid options and commits to it.

The robot executes the first few actions of the sampled sequence, then generates again (the same "plan, act a little, re-plan" loop as in Chapter 13). Across 15 tasks from 4 benchmarks, the slides report an average improvement of 46.9% over the previous best (Chi et al., 2023). Figure 7.23 shows the T-pushing task and a two-arm sauce-preparation setup.

The lecture is careful, and this book repeats the caveat verbatim, to flag the frame with a warning: *this is supervised imitation learning, not SSL.* The state–action pairs come from human teleoperated demonstrations; no label is manufactured from raw data. Yet it belongs in this chapter twice over: it is the policy-learning application of exactly the velocity-regression machinery developed above, and it previews the "when *is* it self-supervision?" demarcation question that Chapter 13 takes up again for world models—and that Chapter 14 will turn system-side, where robots generate their own supervision.

7.11 Closing the taxonomy loop

Diffusion crosses the taxonomy. Slide 68 shows the taxonomy tree of Section 7.1 again. Slide 69 draws a red box around "Diffusion Models" and a red arrow moving it to the *explicit* side, with "Flow Matching" written in beside the flows.

That arrow is the chapter's main claim, and each link has now been shown:

1. Rectified-flow training is plain squared-error regression (Listing 7.1).
2. Sampling from it means following an ODE (Listing 7.2).
3. Any ODE that carries noise to data is a continuous normalizing flow (Eq. (7.24)).

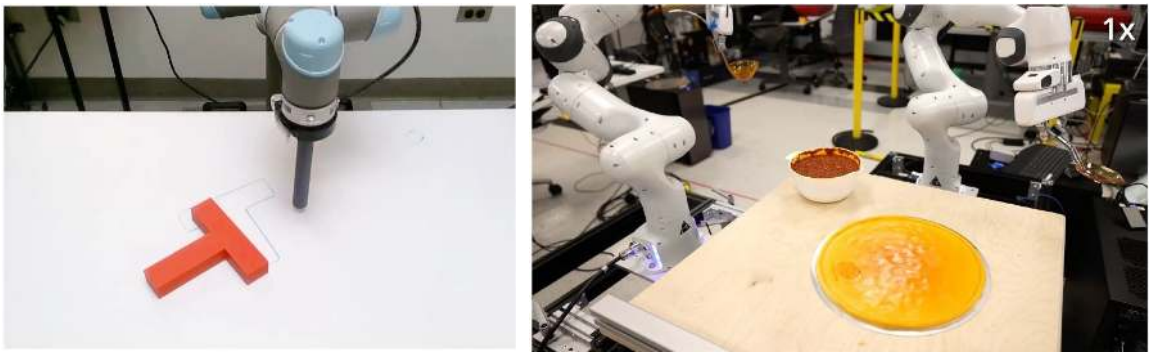


Figure 7.23. The two Diffusion Policy demos shown in the lecture (Lecture 05, slides 65–66; stills from Chi et al. (2023)). *Left:* the T-pushing task — a robot arm must push the orange T-shaped block into the outlined goal pose, and from many states there are two equally valid motions (push it left or right around the corner). This deliberate *multimodality* is why the policy is a generative model over action sequences rather than a regressor: averaging the two valid motions yields a third, invalid one, while a denoising sampler commits to one mode per trajectory chunk — the same conditional-mean averaging failure this chapter met for the squared-error denoiser (Section 7.2). *Right:* the two-arm sauce-preparation setup, where both arms execute coordinated action chunks sampled in receding-horizon fashion.

How to read it: The pictures are of robots, but the mathematics above them is unchanged: what is being "denoised" is not an image but a chunk of future actions, and what plays the role of the conditioning text or class is the camera's current observation. Noticing that is the whole point of closing the chapter here.

Source: Lecture 05, slides 65–66.

4. Continuous flows can compute exact probabilities, made affordable by the Hutchinson trick.

So diffusion was only ever filed under "can't compute probabilities" because the way it was first presented hid the ODE inside it.

Check Your Understanding

1. **Level 1.** In Eq. (7.18), what is $\boldsymbol{x}_t$ at $t = 0$ and $t = 1$? Why does training sample t uniformly rather than always using $t = \frac{1}{2}$?
2. **Level 1.** Write the five lines of rectified-flow training from memory and annotate what each line samples or computes.
3. **Level 1.** State the diffusion-vs-GAN-vs-VAE trade-offs on (i) training stability and (ii) sampling cost.
4. **Level 2.** Many training segments cross a single point $\boldsymbol{x}_t$ with different velocities. Show that MSE regression recovers the conditional mean velocity at $\boldsymbol{x}_t$, and explain (one sentence) why the conditional mean is the right field to integrate.
5. **Level 2.** Compute $\operatorname{tr} \boldsymbol{A} = \mathbb{E}_{\boldsymbol{\varepsilon}}[\boldsymbol{\varepsilon}^\top \boldsymbol{A} \boldsymbol{\varepsilon}]$ explicitly for a diagonal 2×2 matrix $\boldsymbol{A} = \operatorname{diag}(a_1, a_2)$ and Rademacher $\boldsymbol{\varepsilon}$. Why is this estimator cheaper than the exact trace inside a neural ODE?

6. **Level 2.** In classifier-free guidance, what happens at $w = 0$, and what distribution does large w tilt toward? Why does sampling cost double?
7. **Level 2.** Why does latent diffusion keep the VAE's KL weight very small, and what failure of plain VAE decoding does the discriminator fix?
8. **Level 3.** Derive Euler sampling (Eq. (7.21)) from $\dot{\boldsymbol{x}}_t = f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$, and explain why straight paths make few-step Euler exact while curved paths do not.
9. **Level 3.** Give the transport-map argument for placing flow matching on the taxonomy's explicit-density side, naming the two formulas that make the placement precise. Under what practical conditions is the claimed density evaluation misleading?
10. **Level 3.** In the two-point worked example, the noise was changed from two points to a Gaussian. Redo the $t = 0$ conditional-mean velocity for $x = 4$ and $z \sim \mathcal{N}(0, 1)$.
11. **Level 4.** Autoregressive image models have exact likelihoods; diffusion models do not in practice. Yet Stable Diffusion is a wholly diffusion-based system that displaced autoregressive image generators. Which costs dominate, and what role does the latent tokenizer play?
12. **Level 4.** DIFT finds correspondences by cosine similarity of U-Net features at noise level t. Predict and justify what happens to correspondence quality at very small and very large t.
13. **Level 4.** Design a training protocol that turns rectified flow into an exactly one-step generator. What re-pairings of data and noise are required? (Hint: Section 7.5.6.)
14. **Level 5.** Is Diffusion Policy self-supervised? Argue both sides precisely, connecting to the demarcation criteria of Chapter 1, and state what raw robot data a genuinely self-supervised variant would exploit.
15. **Level 1.** (Quiz Q01.) Starting from $\boldsymbol{x}_t = \sqrt{\alpha_t}\,\boldsymbol{x}_{t-1} + \sqrt{1-\alpha_t}\,\boldsymbol{\varepsilon}_t$, derive the closed-form marginal $q(\boldsymbol{x}_t \mid \boldsymbol{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0,\, (1-\bar{\alpha}_t)\boldsymbol{I})$ step by step, stating where the assumption "independent Gaussians add variances" is used.
16. **Level 2.** (Quiz Q15.) For standardized data, verify that the schedule in Eq. (7.4) is variance-preserving, and compute the SNR of the $\boldsymbol{x}_t$ marginal as a function of $\bar{\alpha}_t$. For $\bar{\alpha}_t \in \{0.9, 0.5, 0.01\}$, state whether $\boldsymbol{x}_t$ is closer to data or to noise.
17. **Level 3.** (Quizzes Q02/Q03.) Derive the conversion between $\boldsymbol{\varepsilon}$-prediction and $\boldsymbol{x}_0$-prediction (Eq. (7.6)) and show that an $\boldsymbol{\varepsilon}$-space error of norm $\|\boldsymbol{\delta}\|$ is amplified by $1/\sqrt{\bar{\alpha}_t}$ in $\boldsymbol{x}_0$-space. Explain in one paragraph why this singles out the start of generation as the unstable region.
18. **Level 3.** (Quiz Q13.) Starting from Tweedie's identity $\mathbb{E}[\boldsymbol{x}_0 \mid \boldsymbol{y}] = \boldsymbol{y} + \sigma^2 \nabla_{\boldsymbol{y}} \log p(\boldsymbol{y})$ for Gaussian corruption at variance σ^2, derive $\nabla_{\boldsymbol{x}_t} \log p_t(\boldsymbol{x}_t) = (\sqrt{\bar{\alpha}_t}\, D(\boldsymbol{x}_t) - \boldsymbol{x}_t)/\sigma_t^2$ for $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sigma_t \boldsymbol{\varepsilon}$, and state in one sentence what this implies about denoiser training.
19. **Level 4.** (Quiz Q11.) A colleague retrains only the encoder of a VAE and the training-set ELBO increases. Using the identity $\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \text{ELBO} + D_{\text{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid$

$\boldsymbol{x})\|p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x}))$, decide whether the true log-likelihood has changed, and justify your answer in two sentences.

20. **Level 4.** (Quizzes Q05/Q14/Q09.) A DAE trained at $\sigma = 0.1$ is evaluated at $\sigma = 0.6$ and fails. Diagnose the failure geometrically, state the time-amortized fix used by diffusion models, and specify which timestep you would read U-Net features at for (i) matching cat ears across breeds and (ii) tracking a moving whisker.

Solution sketches.

1. t=0 gives the data point, t=1 the noise point. Uniform t makes the network a correct denoiser at *every* noise level, which the sampler needs at every Euler step; one fixed t trains a single-step corrector only.
2. See Listing 7.1: sample $\boldsymbol{x}$; sample $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$; sample $t \sim \text{Uniform}[0, 1]$; interpolate; regress $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$ by squared error.
3. Diffusion: stable regression training, slow iterative sampling. GAN: unstable minimax training, one-shot sampling. VAE: stable reconstruction-based training, one-shot sampling but blurrier samples (approximate density).
4. MSE risk is minimized pointwise by the conditional expectation $\mathbb{E}[\boldsymbol{v} \mid \boldsymbol{x}_t, t]$; the conditional mean is the ODE's marginal velocity field, and integrating it transports noise to data while individual training segments need not be followed by any particle.
5. With Rademacher entries, $\mathbb{E}[(\varepsilon_1 a_1 \varepsilon_1) + (\varepsilon_2 a_2 \varepsilon_2)] = a_1 + a_2$ since $\varepsilon_i^2 = 1$ and $\mathbb{E}[\varepsilon_1 \varepsilon_2] = 0$, the cross terms vanish. A vector–Jacobian product is reverse-mode autodiff—one backward pass rather than d of them.
6. $w = 0$: conditional sampling. Large w tilts toward $p(\boldsymbol{x} \mid \boldsymbol{y})^{1+w} p(\boldsymbol{x})^{-w}$: sharper, more on-class, less diverse, with contrast artefacts. Two forward passes per step double cost.
7. A full-weight KL over-regularizes latents toward the prior, sacrificing detail the decoder and diffusion stage need. The GAN discriminator removes the Gaussian-decoder blur by penalizing any perceptible deviation from real images.
8. Euler: $\boldsymbol{x}(t + \Delta t) \approx \boldsymbol{x}(t) + f_{\boldsymbol{\theta}}(\boldsymbol{x}, t)\, \Delta t$ with $\Delta t = -1/T$. Constant-velocity (straight) ODEs are integrated exactly by any Euler step; curvature makes the local-linear step drift off trajectory, error compounding over the remaining steps.
9. Sampling integrates an invertible ODE $\boldsymbol{x} = \boldsymbol{z}_0 + \int f\, dt$; Eq. (7.24) evaluates its density, with the Hutchinson trick making the trace affordable. Misleading when the field is learned (not exact), discretization is coarse, and one $\boldsymbol{\varepsilon}$ draw is used—exact in theory, noisy in practice.
10. $\mathbb{E}[z - x \mid x_0 = 4, t = 0] = \mathbb{E}[z] - 4 = 0 - 4 = -4$; the Gaussian case averages over the same conditional-mean mechanism.
11. Serial sampling cost dominates: 3M sequential subpixel decisions versus ∼50 parallel-per-pixel network evaluations. Latent tokenizers (a VAE trained with an extra GAN loss) shorten sequences and let either family work in a compact space; likelihood exactness mattered less than sample quality per FLOP.

12. Small t: features track texture and noise, poor part matching. Very large t: geometry is destroyed with the semantics. Mid-range t keeps part structure while discarding instance detail—best correspondences.
13. Reflow: train once, then generate new $(\boldsymbol{x}, \boldsymbol{z})$ pairs by sampling with the current model and retrain flow matching on those pairs; generated noise–data couplings share the model's path, straightening trajectories so a T=1 Euler step becomes nearly exact.
14. Not SSL as presented: action labels are human teleoperation, a supervised external signal; the diffusion machinery is irrelevant to the supervision question. A self-supervised variant would generate its own training signal from unlabeled robot streams—e.g. predict future sensory states, use proprioception to supervise vision—in the spirit of Chapter 14's systems.
15. Substitute recursively; at each step the noise terms merge into one Gaussian whose variance is computed from $\alpha_2(1-\alpha_1) + (1-\alpha_2) = 1 - \alpha_1\alpha_2$; the induction yields Eq. (7.3).
16. $\mathrm{Var}(\boldsymbol{x}_t) = \bar{\alpha}_t + (1-\bar{\alpha}_t) = 1$ by unit variance of both terms; $\mathrm{SNR}(t) = \bar{\alpha}_t/(1-\bar{\alpha}_t)$ evaluates to 9, 1, 1/99 for the three $\bar{\alpha}_t$ values—mostly data, balanced, essentially noise.
17. Rearrange $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\varepsilon}$ for $\boldsymbol{x}_0$; the induced error is $-\sqrt{1-\bar{\alpha}_t}/\sqrt{\bar{\alpha}_t}\,\boldsymbol{\delta}$, whose norm ratio blows up as $\bar{\alpha}_t \to 0$. Large t is where sampling begins, so the initial iterates inherit the amplified error.
18. Substitute $\boldsymbol{y} \to \boldsymbol{x}_t$ and $\sigma^2 \to 1-\bar{\alpha}_t$; the clean variable is $\sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0$, whose posterior mean is $\sqrt{\bar{\alpha}_t}\,D(\boldsymbol{x}_t)$. Rearrange and divide by σ_t^2: the denoising displacement is the score, so training a denoiser is score estimation (Hyvärinen, 2005).
19. No change: $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ depends only on $\boldsymbol{\theta}$, which was not updated; the encoder moved into a smaller KL gap between its approximate posterior and the fixed true posterior, lifting the bound toward a constant line.
20. The small-σ DAE learned only a local projection near the manifold and sees $\sigma = 0.6$ inputs off its training distribution, so it under-cleans. Amortizing over t in one network handles all scales. Read large $t \approx 500$ for cross-breed ear matching; small $t \approx 10$ for within-instance whisker tracking.

Concept Map — how this chapter connects

Backward links. Chapter 4's VAE supplies the ELBO panel recalled on slide 50 and the latent-diffusion stage 1; Chapter 6 supplies the normalizing flow change of variables (Eq. (7.1)), the GAN minimax game that sharpens the LDM decoder, and the taxonomy this chapter redraws. Chapter 5's masked autoencoders are the DAE's masking cousin, and Chapter 5's patch tokens are the vision version of the autoregressive "tiles" fix.

Forward links. DIFT previews Chapter 11's thesis that self-supervised backbones expose part structure usable by cosine similarity—DINO's emergent attention maps are the discriminative counterpart of DIFT's generative features. The autoregressive chain rule and its per-token cost anticipate the sequence models behind Chapters 11–13. Diffusion Policy's "is this SSL?" question is exactly Chapter 13's world-models debate

and Chapter 14's system-level supervision, where robots manufacture their own labels. Part III (Chapter 16) revisits guidance, sampling cost, and the likelihood–quality trade-off when comparing method families head to head.

CHAPTER 8

Pretext Tasks and Augmentations

> "To generate samples in X, models need to learn some underlying structure of X."
>
> Lecture 06, opening slide

In this chapter

8.1 From Generating Data to Representing It 378
8.2 Augmentations: The Raw Material . 380 • A vision taxonomy, with design rationale • Time series: jittering and windowing • Text: from character noise to backtranslation
8.3 Augmentations as Inductive Bias 384
8.4 The Organizing Question: Invariance versus Covariance 386
8.5 Reconstruction-Based Pretext: Inpainting ..... 389 • The reconstruct-versus-predict split • Context Encoders
8.6 Prediction-Based Pretext Tasks 392 • Relative Patch Location • Jigsaw Puzzles • Colorization and Split-Brain • Video Colorization: The Emergent-Capability Moment • Temporal Order
8.7 Worked Example: RotNet 398 • The hypothesis: visual commonsense • Method • What the network actually learns • Results: the first honest transfer table
8.8 Judging What We Learned: An Evaluation Segue 403
8.9 Closing the Arc: From Predicting the Augmentation to Ignoring It 406 • The compromise: keep jigsaw, throw away jigsaw • The loss • Why this closes the arc

Chapter roadmap

Before you start. You need three things. (i) What an *encoder* is: a network $f_{\boldsymbol{\theta}}$ that turns an input into a vector (Chapter 4). (ii) What *cross-entropy* classification is: the loss you use when a network must pick one of k answers (Chapter 2). (iii) Roughly what masking did for the masked autoencoder of Chapter 5 — hide part of the input, predict it back. Everything else is built here.

The idea in one sentence. If you change a picture in a way you chose yourself, you know exactly what changed — so you can make up a free question about the change and train a network on it, and the network has to understand the picture to answer.

What we cover.

- *Augmentations* (Section 8.2): the small library of image edits everything else is built from. We need the vocabulary before we can build tasks out of it.
- *Invariance versus covariance* (Section 8.4): the one question that splits this chapter in two. Should the features forget the edit, or remember it?
- *Undo the edit* (Section 8.5): inpainting and Context Encoders. Delete a region, paint it back.
- *Name the edit* (Section 8.6): patch location, jigsaw, colourisation, video colourisation, temporal order. Each one is a guessing game whose answer you already know.
- *RotNet* (Section 8.7): the simplest game of all — which way is up? — worked end to end, because it is the one that won.
- *How to grade a representation* (Section 8.8): kNN, linear probe, fine-tuning, and why they disagree.
- *PIRL* (Section 8.9): the method that keeps the edit but throws away the question — and walks us into Chapter 9.

Afterwards you will be able to (1) design a pretext task and name the cheap shortcut it invites, plus a fix; (2) say, for any augmentation, whether you are asking the features to ignore it or to track it, and what that costs; (3) read a transfer table and say which column measures feature *quality* and which measures feature *initialisation*; (4) explain why predicting the transformation was eventually abandoned in favour of ignoring it.

8.1 From Generating Data to Representing It

So far, Part II has toured generative models: VAEs (Chapter 4), masked and denoising autoencoders (Chapters 5 and 7), flows and GANs (Chapter 6), and diffusion and flow matching (Chapter 7). The lecture's recap arranges them with Chapter 6's taxonomy:

- models that compute $p(\boldsymbol{x})$ *exactly* — autoregressive models, flows, flow matching;
- models that compute it *approximately* — VAEs;
- models that can only *sample* — GANs directly, diffusion step by step.

What they share, and what matters from here on, is the epigraph's point: a model that can generate the data must have learned *how the data is structured*.

That internalized structure lives in the model's internals, and it can be read out as a *feature*. Two pieces of evidence from the generative chapters:

Diffusion features. A diffusion model predicts the clean image from a noisy input at timestep t. Looking at predictions made from large t—where the input is essentially noise—the model still produces images with the right *semantics* (the right object category, coarse layout) but wrong details. The network's intermediate activations therefore organize themselves as a

ladder of abstraction: high noise level corresponds to semantic, detail-free structure. This is exactly the property that makes diffusion features useful for tasks like correspondence (DIFT, covered in Chapter 7).

Latent interpolations. The VAE latent spaces of Chapter 4 support smooth interpolation: walking in a straight line between the codes of two faces produces a grid of intermediate faces that morph continuously in identity and orientation. A space in which straight lines are semantically meaningful is a useful space.

Generation is an expensive way to learn structure. To render a realistic background, a generator must decide where every leaf sits and how every hair falls. Most of its capacity goes on detail a recognition task will never use. Real images occupy a tiny, highly structured corner of all possible pixel grids, and it is *that structure* — not the rendering — that good features should capture. Getting features from a generative model means paying the full price of generation (likelihoods, decoders, sampling) for what is really a by-product.

A change of goal. If what we really want is a good mapping from inputs X to representations Z, why pay for full generative modelling — likelihoods, decoders, sampling — at all? This chapter and the next three learn representations *directly*:

1. build a training signal from the data itself;
2. train an encoder on it;
3. keep the encoder, and discard whatever produced the signal.

The organising principle (lecture slide 8) is short enough to memorise: *learn what to ignore and what to track, so the representation keeps only what matters.* MAE (Chapter 5) already fits: it captures the image's content while being indifferent to which patches were hidden. This chapter makes the principle explicit and explores its two extremes.

Key Concept — Representation

A *representation* (or feature) of an input $\boldsymbol{x} \in X$ is a vector $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in Z$, produced by an encoder $f_{\boldsymbol{\theta}}$, that is more convenient to process than $\boldsymbol{x}$ itself. "Convenient" usually means: lower-dimensional or better structured, and equipped with a meaningful geometry—points of Z that are close under a standard metric (Euclidean or cosine, Chapter 2) correspond to inputs that are semantically close. Two photos of the same kind of dog should land at nearby points of Z; a photo of a cat should land farther away, whether we measure closeness by $d(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|_2$ or by the cosine similarity $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^\top \boldsymbol{v} / (\|\boldsymbol{u}\| \|\boldsymbol{v}\|)$ introduced in the notation table.

In plain words

A generative model has to draw the whole picture. A representation only has to *describe* it. Drawing is harder, and most of the effort goes into detail — every leaf, every hair — that nobody downstream will ever ask about. So instead of training a painter and then peeking at its brain, we will train a describer directly. The rest of the chapter is about how to grade a describer when nobody has written down the right answers.

Key takeaway

Generative models learn structure as a *by-product* of learning to draw. That structure is what we actually want. This chapter buys it directly — make up a question whose answer you already know, train an encoder to answer it, then throw the question away and keep the encoder.

8.2 Augmentations: The Raw Material

Every method in this part of the book is built from one primitive ingredient: a family $\mathcal{T}$ of transformations $t : X \to X$ we can apply to data, called *augmentations*. An *augmentation* is just an edit you make to an input on purpose: crop it, flip it, wash out its colours, blank out a square. Because you chose the edit, you know it exactly — and that is the whole trick. The lecture's butterfly panel (Lecture 06, slides 10–11) catalogs the standard vision family; we add the design considerations the slides leave implicit.

New words used from here on

Augmentation family $\mathcal{T}$. The set of edits you are willing to make, together with the random range of each ("crop to between 20% and 100% of the area", "shift hue by up to ± 0.1"). Sampling $t \sim \mathcal{T}$ means picking one edit with random settings.

Nuisance variable. Something that changes in the picture but must not change the answer. Lighting is a nuisance for "is this a dog".

Invariant. The features do not move when the edit is applied.

Covariant. The features do move, in a way that still tells you which edit was applied.

Pretext task. A made-up training task whose labels come free from the data pipeline. Nobody cares about the task itself; we care about what the network had to learn in order to do it.

8.2.1 A vision taxonomy, with design rationale

For images, the canonical family $\mathcal{T}$ contains:

- **Geometric transforms.** Random-resized crop, horizontal flip, rotation by multiples of $90°$ (or small arbitrary angles), and mild affine warps. Rationale: object identity and

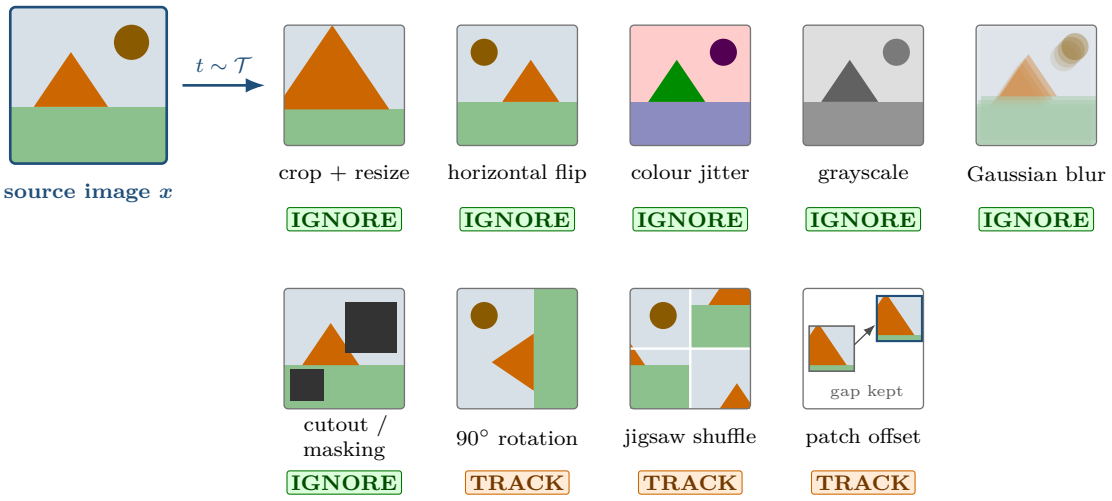


Figure 8.1. The augmentation zoo, and the one decision attached to every entry. One source image on the left; nine edits drawn from the standard vision family $\mathcal{T}$ on the right. Each edit carries a tag saying what we want the representation to do about it. The green edits are nuisances we want erased; the orange ones are the edits whose identity a pretext task will ask the network to recover. Nothing about the *picture* decides the tag — we do, and we can be wrong.

How to read it: Pick any one tile and ask "if I saw only this, could I still name the object?" If yes, it belongs under IGNORE. Then notice that the bottom-row three tagged TRACK are tagged that way only because Sections 8.6.1–8.7 choose to build guessing games out of them; the same three could be tagged IGNORE instead, and Section 8.9 does exactly that.

class are stable under changing the camera's position and the sensor's orientation. The random *resized* crop does double duty—it changes both scale and aspect ratio, forcing features to respond to content rather than to a canonical framing.

- **Photometric transforms.** Brightness/contrast jitter, color jitter (per-channel scaling), saturation changes, grayscale conversion (*color drop*), and Gaussian blur. Rationale: lighting, white balance, and camera response are nuisance variables for most semantic tasks.
- **Occlusion and masking.** Cutout (zeroing a random rectangle) and patch-wise masking (as in MAE). Rationale: objects are partially occluded in deployment; the model must infer content from context.
- **Noise.** Additive Gaussian noise, JPEG artifacts; occasionally edge or Sobel filtering.

The desiderata for choosing $\mathcal{T}$ are in tension. An augmentation should (i) *destroy information that is not needed* for the downstream tasks you care about and (ii) *preserve the*

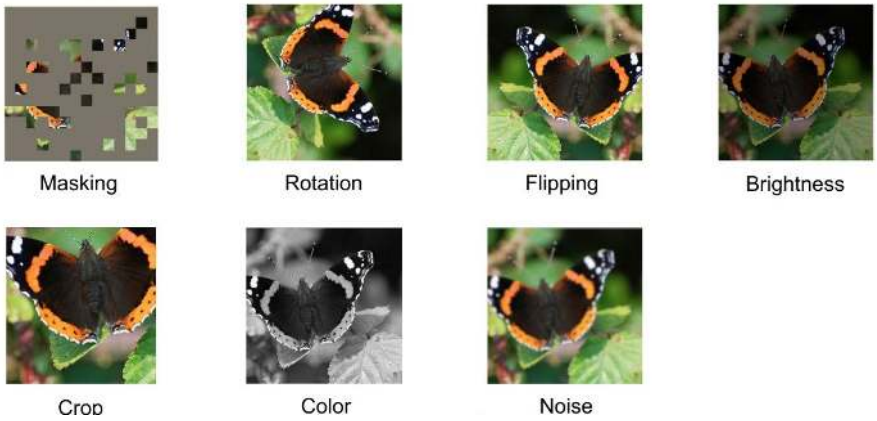


Figure 8.2. The same seven edits on a real photograph, from the lecture. A human names the butterfly in every tile, which is the empirical claim behind tagging these edits "IGNORE" in Figure 8.1. Note how much the pixel values move — grayscale and brightness change nearly every number in the array — while the content does not.

Source: Lecture 06, slide 11.

Table 8.1. Augmentation families by modality. Each family is a sampleable set of parameterized transformations; the "risk" column names the kind of label information the family can destroy when applied carelessly.

Modality	Transforms	Risk
Vision	crop/resize, flip, rotate, color jitter, grayscale, blur, noise, cutout, masking	crops out the object itself; photometric jitter destroys color-, brightness-, or texture-based distinctions
Time series	amplitude scaling, time shift, DC (offset) shift, window slicing, zero-masking, additive noise, band-stop filtering	removes the frequency band or amplitude range that carried the label; destroys absolute level when it is the signal
Text	character noise, word dropout/swap/deletion, synonyms, embedding neighbors, language-model paraphrase, interpolation, backtranslation	character noise unlikely under deployment; strong edits change meaning or sentiment

information that is. Since we do not know the downstream tasks in advance, augmentation design is applied domain knowledge: horizontal flips are safe for natural images but wrong for satellite imagery with orientation-sensitive labels; 90° rotations are fine for medical scans with no canonical "up" but destroy the very signal RotNet will exploit (Section 8.7).

8.2.2 Time series: jittering and windowing

Time series. For signals over time — heart traces, audio, sensor readings — good augmentations imitate what really goes wrong with measurements (Lecture 06, slide 12):

Augmentation	What it imitates
amplitude scaling	the sensor's gain changes
time shifting	the event starts at a different time
DC shifting	the baseline drifts
masking / cropping a window	a stretch of signal drops out
added Gaussian noise	electrical noise
band-stop filtering	interference blocks some frequencies

Two cautions. Cutting a long recording into windows is both an augmentation and a decision about what one "sample" is. And many time-series labels depend on exactly what these augmentations change: "blood pressure above 140" depends on amplitude, a heart-rhythm diagnosis on frequency. Scaling or filtering can erase the very thing the downstream task needs.

8.2.3 Text: from character noise to backtranslation

Text augmentations organize by granularity (the lecture cites the taxonomy of Bayer et al., 2021). At the *character level*: random noise (typos, swaps) and rule-based edits. At the *word level*: deletion (*word dropout*), permutation, synonym substitution, replacement by embedding-space neighbors, replacement by language-model predictions. At the *phrase level*: structural reordering and interpolation between sentences. At the *document level*: machine translation to another language and back (*backtranslation*), which preserves meaning while rewriting surface form, and fully generative rewriting. The deeper the granularity, the more semantic the perturbation — and the higher the risk that the edit changes the meaning (negation flips, sentiment flips). As a rule of thumb, augmentations that humans would describe as "reading the same thing written differently" are safe; augmentations that look like typos are only safe if the deployment data contains typos.

Key takeaway

An augmentation family is not a utility function — it is a *claim* about the domain. "Flipping is safe" means "no task I care about can tell left from right." Every entry of $\mathcal{T}$ is a bet, and the bet is settled only downstream, long after pretraining is over.

Historical Note

Augmentation-as-supervision is a rediscovery, not an invention. The handcrafted feature descriptors of classical computer vision — SIFT chief among them — were *engineered* to be invariant: histogram-of-gradient representations binned over spatial cells to gain translation tolerance, estimated dominant orientations to gain rotation invariance, illumination

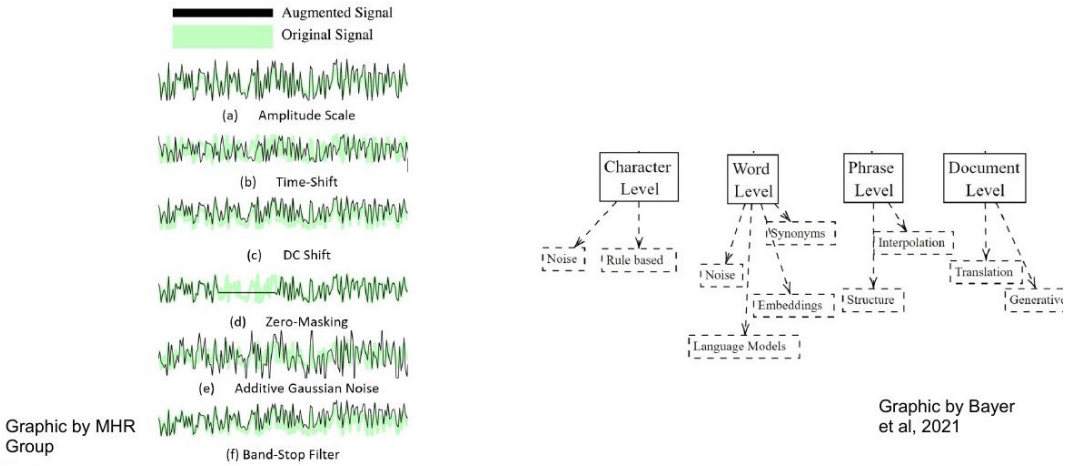


Figure 8.3. Augmentations outside vision. *Left*: six standard edits for a one-dimensional signal — amplitude scaling, time shift, DC (baseline) shift, zero-masking, added Gaussian noise, band-stop filtering. Each one imitates a way a real sensor gets corrupted. *Right*: the text taxonomy by granularity, from character-level noise up to whole-document rewriting. The pattern is the same in both: an augmentation family is a written-down list of the ways the world is allowed to vary without the answer changing.

Source: Lecture 06, slide 12; time-series panel after the MHR Group graphic, text panel after Bayer et al., 2021.

normalization for photometric robustness. The research program of the 1990s and 2000s was to identify the nuisance transformations of the domain and then divide them out by hand. Modern representation learning replaces the hand design with a training objective: the transformations are still chosen by the practitioner (the augmentation family is still domain knowledge), but the invariance is *learned* by gradient descent rather than hard-wired into a descriptor. The shift is from "representations that are invariant by construction" to "representations forced to be invariant by the loss" — and, as we will see in Section 8.4, to the discovery that invariance itself is sometimes the wrong target.

8.3 Augmentations as Inductive Bias

Key Concept — Augmentation as inductive bias

An *inductive bias* is any assumption a learning algorithm uses to choose among hypotheses that fit the training data equally well. In self-supervised learning the dominant inductive bias is not the architecture but the augmentation family: choosing that a model should be invariant (or covariant) to a transformation t is exactly the assertion that t does not (or does) change the information a downstream task will need.

Why the augmentations carry so much weight. In self-supervised learning, the augmentations *are* the supervision. The pretext labels and the positive pairs are built from

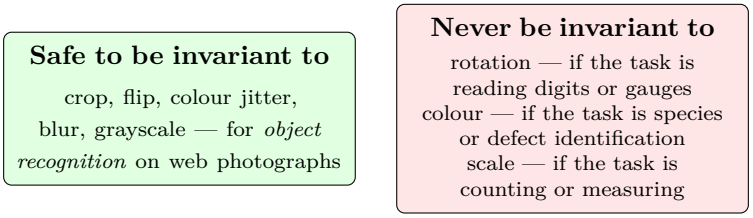


Figure 8.4. The augmentation contract. Choosing an augmentation set is choosing what the representation is allowed to forget, which makes it the most consequential and least examined decision in the whole pipeline.

nothing but $\boldsymbol{x}$ and the transformations we apply (Section 8.4). So the set of augmentations $\mathcal{T}$ decides what the network can and cannot learn to distinguish.

A concrete example with digits:

- Train for invariance to horizontal flips, and a "6" and its mirror image get the same features.
- Train for invariance to 180° rotation, and a "6" and a "9" get the same features — harmless for most objects, fatal for reading digits.

The bird pair of Section 8.4 makes the same point: whatever you decide about the rotated photo is a choice with consequences.

Supervised learning hides this choice in the labels. Annotators agree that a dog facing left, flipped to face right, is still the same dog. So the label file *already* says "flips don't matter", and augmentation only nudges the classifier towards what the labels imply (the Common Confusion box of Section 8.4).

Self-supervised learning removes the labels, so our assumptions have to stand alone: the augmentation set is the *only* place our beliefs about the domain enter. That is why Section 8.2 listed the risks next to each transformation, and why Section 8.8 calls the augmentation set the least principled part of the whole pipeline.

Networks take the cheapest route. If a pretext label can be predicted from some low-level statistic that needs no understanding, gradient descent will find it. "Match the colour histograms" will beat "understand the object" whenever it works.

So augmentations are partly *defences* against specific cheats:

Defence	Cheat it blocks
colour jitter	matching colour histograms
gaps between patches (Section 8.6.1)	matching patch borders
only right-angle rotations (Section 8.7)	spotting interpolation artefacts

The augmentation set is where we both *state* our assumptions and *protect* them.

Everyday analogy — writing an exam

Setting a pretext task is like writing an exam question. You want a question that can only be answered by a student who understands the material. But students are resourceful: if the longest multiple-choice option is always the correct one, they will learn *that* instead of the physics. A network is the most resourceful student imaginable, with infinite patience and no shame. So the craft is not "write a hard question" but "close every cheap route to the right answer".

Where the analogy breaks. A student who cheats still knows they cheated. A network has no notion of a legitimate answer — the shortcut *is* the solution as far as the loss is concerned, and nothing in training will ever report the cheat. You only find out downstream, when the features fail at a task the shortcut never touched.

Key takeaway

In supervised learning the inductive bias hides in the labels. In self-supervised learning there are no labels, so the bias has nowhere to hide: it sits, fully exposed, in the augmentation family. Choose $\mathcal{T}$ and you have chosen what the representation is permitted to forget.

8.4 The Organizing Question: Invariance versus Covariance

Suppose we have chosen a family $\mathcal{T}$ and a transformation $t \in \mathcal{T}$ applied to an image $\boldsymbol{x}$. What should the encoder $f_{\boldsymbol{\theta}}$ do with $\boldsymbol{x}_t = t(\boldsymbol{x})$? Lecture 06 poses exactly this question with two photos of the same bird, one rotated by 180° (slide 38; Figure 8.6). There are two legitimate answers, and they lead to two different method families.

Covariance. The representation should *co-vary* with the transformation: $f_{\boldsymbol{\theta}}(t(\boldsymbol{x}))$ should encode which t was applied. The encoder must learn what an "unperturbed" version of the world looks like in order to tell the transform apart from content. This leads to *pretext classification*: predict the augmentation (Section 8.6).

Invariance. The representation should *ignore* the transformation: either reconstruct the un-transformed $\boldsymbol{x}$ (Section 8.5) or make the features of $\boldsymbol{x}$ and $t(\boldsymbol{x})$ agree directly (Chapters 9–11).

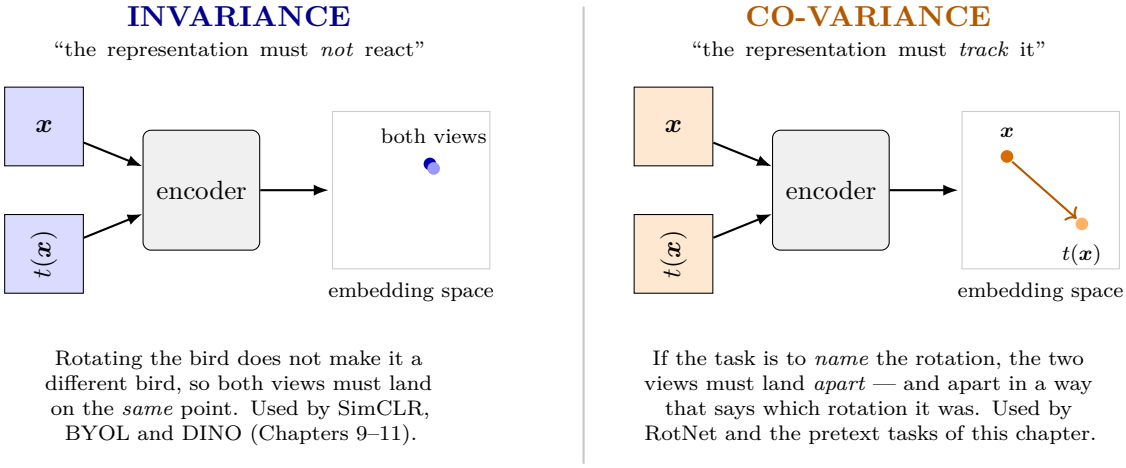


Figure 8.5. The organising question of this chapter, in one picture. The same input, the same transformation and the same encoder — and two opposite demands on where the two views should land.

How to read it: Look only at the two small squares on the right of each half. One has the points on top of each other; the other has them deliberately separated. Everything else is identical.

Figure 8.6. The question the whole chapter turns on, as the lecture poses it. Two photographs of the same bird on the same trunk, one of them turned upside down. Is it more important that the two feature vectors can tell the rotation apart, or that both say "bird"? Both answers are defensible; they produce different methods and different failures.

Source: Lecture 06, slide 38.

Written as equations, for a transformation t and encoder $f_{\boldsymbol{\theta}}$:

$$\text{invariance: } f_{\boldsymbol{\theta}}(t(\boldsymbol{x})) = f_{\boldsymbol{\theta}}(\boldsymbol{x}), \qquad \text{covariance: } f_{\boldsymbol{\theta}}(t(\boldsymbol{x})) = t'(f_{\boldsymbol{\theta}}(\boldsymbol{x})),$$

where t' is some matching change in feature space.

- **Invariance**: after the transformation the features are *exactly the same*. No trace of t is left.
- **Covariance**: the features change, but predictably, so you can still tell which t was applied. For a transformation from a fixed list ("which of four rotations?") this just means a simple classifier can read the answer off the features.

Where the methods sit:

- **RotNet** is covariant: its features must reveal the rotation, or its head could not predict it.
- **PIRL and the contrastive methods** of Chapter 9 train invariance directly.
- **Reconstruction methods** are in between: the encoder must keep enough to undo t, but the decoder does the t-specific work — and the decoder is thrown away.

In information terms, with y the identity of the transformation: invariance means the features tell you nothing about y, $I(\boldsymbol{z}; y) = 0$; covariance means they do, $I(\boldsymbol{z}; y) > 0$.

In plain words

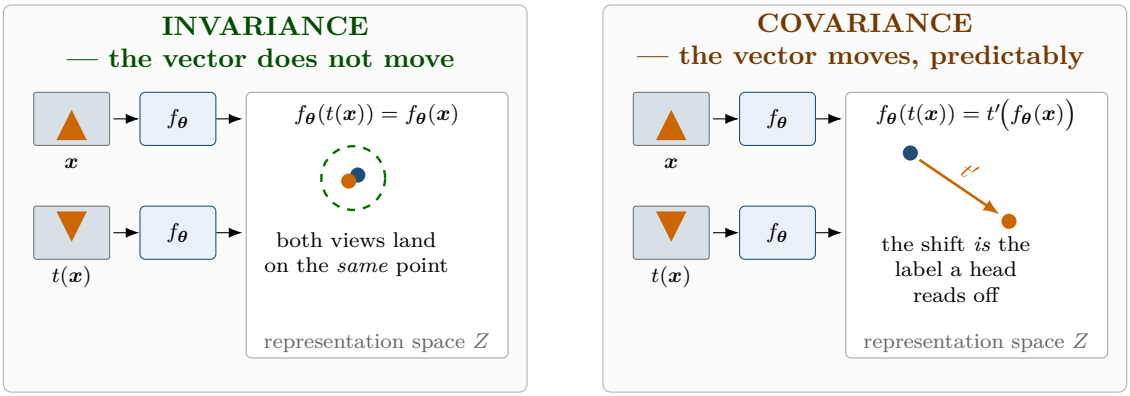
Take a photo. Turn it upside down. Feed both to the encoder and look at the two vectors it produces.
If the two vectors are *in the same place*, the encoder has forgotten the turn. That is **invariance**. You can no longer ask it which way up the photo was, but you also never have to worry that turning a photo changes what it thinks the photo shows.
If the two vectors are *in different places, in a predictable way* — turning the photo always moves the vector in the same direction — the encoder has remembered the turn. That is **covariance**. Now a tiny readout can recover the angle, but the encoder has spent part of itself storing something most tasks do not want.
Figure 8.7 draws exactly these two pictures.

Simplified intuition vs. more precise view

Simple story. "Good representations are invariant: nuisance variation goes away, semantic content stays." Mostly true for the downstream tasks this course cares about (classification, detection, retrieval).
More precise view. Invariance is *task-relative*, and total invariance is information loss. Whether rotation should be discarded depends on the task: for "bird or cat" it is a nuisance; for "which way is up" (autonomy, photography quality control, cranial-nerve exams in video) it is the label. A representation maximally invariant to t provably throws away whatever a downstream task might need about t; conversely, predicting t (covariance) forces the network to keep information that *may* be irrelevant downstream. The design question of SSL is therefore not "be invariant" but "choose which variation to preserve and which to destroy — and let representations keep both options open by making the choice at the level of the augmentation family, not the architecture."

Common Confusion

Augmentation as regularizer versus augmentation as supervision. In supervised learning, data augmentation is a *regularizer*: it synthetically multiplies the training set, teaches the classifier an invariance we assume the true labels have, and improves general-



What each one costs. Invariance throws the transform away: no downstream task can ever recover it, however much it would like to. Covariance keeps the transform: the representation now spends capacity on something most semantic tasks will never use. Sections 8.6–8.7 build covariant methods; Section 8.9 and all of Chapter 9 build invariant ones.

Figure 8.7. Invariance and covariance, drawn. The same image and the same transformed copy are pushed through the same encoder in both panels; only the training objective differs. On the left the two embeddings are forced onto one point, so the transform leaves no trace. On the right the embedding is displaced by a fixed, transform-indexed move t', so a linear head can read the transform back out.

How to read it: Compare the two blue-and-orange dot pairs in the middle of each panel. Left: they coincide. Right: they are far apart, and the arrow between them is reproducible — apply the same t to a different image and the embedding moves the same way. Everything else on the two sides is identical.

ization. The labels y still come from human annotators; the augmented samples borrow the original label. In self-supervised learning the same function t plays a fundamentally different role: it *creates* the label. RotNet's label is "which rotation did we apply"; the contrastive label is "these two views came from the same image." No human annotation is involved — the loss is a function of $\boldsymbol{x}$ and t alone. Confusing the two roles leads to mis-analyzing SSL results as "just augmentation regularization." The regularization interpretation is that we know the invariance; the SSL interpretation is that the act of *producing or imposing* the invariance forces the network to model the content that survives it.

8.5 Reconstruction-Based Pretext: Inpainting

The reconstruction schema (Lecture 06, slide 13) says: corrupt the input, ask the network to undo the corruption,

$$\boldsymbol{x} \xrightarrow{\text{augmentation } t} \tilde{\boldsymbol{x}} \xrightarrow{f_\theta} \hat{\boldsymbol{x}} \approx \boldsymbol{x},$$

where the network f_θ maps the corrupted $\tilde{\boldsymbol{x}}$ back to a reconstruction $\hat{\boldsymbol{x}}$ of the clean input. To undo a masking transformation, the network must model what the masked region *should plausibly contain* — and that is representation learning. The denoising and masked autoencoders

of Chapters 5 and 7 are the "noise" and "mask" instances of this schema. The historically pivotal *spatial* instance is inpainting.

8.5.1 The reconstruct-versus-predict split

Before walking through examples it is worth seeing the two families of self-supervision side by side, because the rest of the chapter is an extended argument between them. The reconstruction family of (8.5) must output a *high-dimensional object* — a full image or signal — approximating the undistorted $\boldsymbol{x}$; the prediction family of (8.6) outputs a single discrete label or a handful of numbers. The differences matter beyond architecture.

- **Where capacity goes.** Reconstruction pays for pixel-level synthesis: the decoder must model high-frequency texture, color calibration, and edge placement, most of which the transferred feature never uses again. The encoder we care about is merely the input end of a pipeline whose expensive half (decoder, likelihood) is thrown away at transfer time.
- **Where the content shows up.** Under reconstruction, semantic information appears in the encoder's bottleneck; the transform-specific burden (what exactly was masked, where noise landed) lives mainly in the decoder. Under prediction, the network's internal features must instead *retain* the applied transform — covariance in the sense of (8.4).
- **How training behaves.** Prediction reduces to a standard cross-entropy classification objective — stable, cheap, and well understood — while reconstruction inherits the optimization quirks of generative modeling: regressions to the blurry mean, or the need for adversarial losses, or multibin targets as in colorization.

Historically the reconstruction family won the first round (autoencoders, VAEs) before the prediction pretexts overtook it in transfer benchmarks; the jigsaw layer-specialization finding of Section 8.6.2 and RotNet's clean win of Section 8.7 show why. PIRL then captures the advantages of both by *using an augmentation as data* but *abandoning its prediction game*, training pure invariance in feature space — the inversion that opens the contrastive part of the course (Section 8.9).

8.5.2 Context Encoders

Paper Analysis: Context Encoders (Pathak et al., 2016)

Problem. Learn visual features without labels by requiring a network to fill in a large missing region of an image.
Prior work. Image completion existed in classical graphics (patch-matching), but its features were hand-built, not learned.
Insight. If a network must generate the contents of a masked central region from its surroundings, it must understand *what kind of scene and object* it is looking at — a precondition for semantics. Inpainting is completion under uncertainty, and the uncertainty forces abstraction.
Method. An encoder maps the masked image to a bottleneck representation; because images are spatial and the bottleneck must carry global information, the encoder feeds a *channel-wise fully connected* layer (each channel communicates across spatial locations, without the parameter explosion of a dense map); a decoder upsamples back to image resolution. Training masks are large (about a quarter of the image area).
Objective. A combination of a per-pixel ℓ_2 reconstruction loss on the masked region and an *adversarial* loss (Chapter 6): a discriminator judges whether the completed patch looks real. The ℓ_2 term anchors content; the adversarial term prevents the blurry average-over-possibilities that pure ℓ_2 regression produces.
Results. The completions are semantically plausible (a masked facade is filled with window-like structure) though not pixel-perfect; more importantly, features from the encoder transfer: the method appears in the transfer-learning table of Figure 8.17 and, while weaker than later methods, demonstrated that reconstruction pretexts yield semantic features at all.
Limitations / what to remember. Pure ℓ_2 reconstruction blurs; the necessary adversarial component entangles the method with GAN training instability. The lasting lessons: (i) masking creates information asymmetry that forces modeling content, and (ii) the bottleneck — not the reconstruction quality itself — is the deliverable.

Why does reconstruction produce semantics at all? Because masking creates an *information asymmetry*: the visible context constrains the missing content only through the world's regularities (facades have repeated windows; roads continue; skies are above). The only way to minimize reconstruction error across a dataset is to internalize those regularities in $f_{\boldsymbol{\theta}}$'s weights. This is the same argument that will return, inside out, for masked language modeling in Chapter 11.

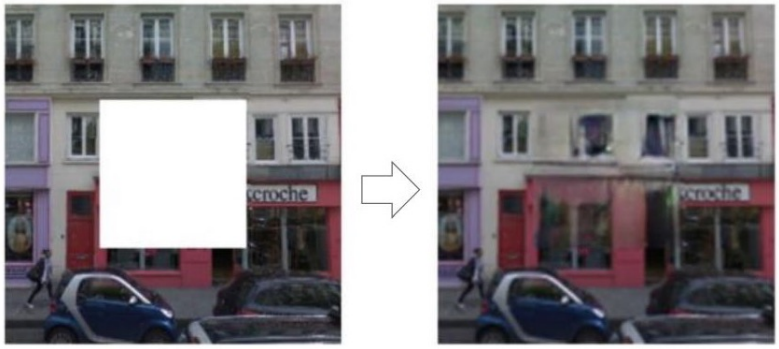


Figure 8.8. Context Encoders schematic, redrawn after Pathak et al. (2016). A 2×2 panel: (top-left) input image with a large central region masked (white); (top-right) encoder–decoder with a channel-wise fully connected bottleneck $\boldsymbol{z}$; (bottom-left) inpainted result with the ℓ_2 + adversarial objective; (bottom-right) the learned features applied downstream. Gray mask, reconstruction target = the missing content.

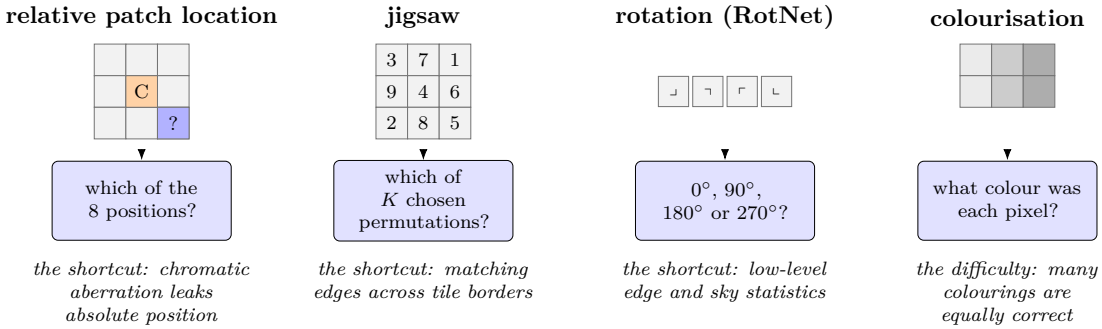


Figure 8.9. Four prediction-based pretext tasks, and — more importantly — the cheat each one leaves open. Every pretext task is a bet that the only way to answer is to understand the scene. The bet is never entirely safe, and most of the research in this chapter is about closing the gaps.

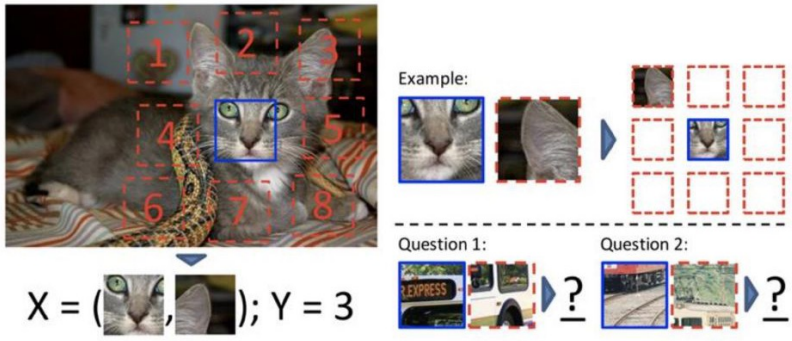
How to read it: Read the bottom row first. Those italic lines are why pretext tasks were eventually displaced by the joint-embedding methods of Chapter 9.

8.6 Prediction-Based Pretext Tasks

The alternative schema (Lecture 06, slide 15) does not reconstruct anything. Take a family of parameterized transformations, apply one, and train the network to *predict which transformation was applied*:

$$\boldsymbol{x} \xrightarrow{t_y \sim \mathcal{T}} \tilde{\boldsymbol{x}} \xrightarrow{f_{\boldsymbol{\theta}}} \hat{y} \approx y,$$

where t_y is the transformation with parameter y and the network answers a classification or regression question about y. The features must encode whatever makes y recoverable — and the art is to choose y such that recovering it requires scene understanding.



(Image source: Doersch et al., 2015)

Figure 8.10. The relative-patch-location task of Doersch et al. (2015). *Left*: an image with the eight candidate neighbor slots (red dashes) around a chosen center patch (blue) — the input is the pair $X = (\text{center patch}, \text{offset patch})$ and the label Y is the index of the slot the second patch came from. *Top right*: one worked pair mapped back into the 3×3 answer grid. *Bottom right*: the task as the network sees it — two crops and a question mark. Answering requires knowing which parts of a scene plausibly sit next to which (a window above a wheel, an ear above a face), not merely matching local texture; Section 8.6.1's shortcut discussion is precisely about the cases where texture *does* suffice.

Source: Lecture 06, slide 16; task graphic after Doersch et al. (2015) (via the CS231n course slides shown in the lecture).

8.6.1 Relative Patch Location

Doersch et al. (2015) asked a disarmingly simple question: given two patches from the same image, *where is the second patch relative to the first*? Fix a center patch; the eight neighbor positions of a 3×3 grid define an 8-way classification task (Lecture 06, slide 16): the input is the pair (center patch, offset patch), the label is one of the eight relative positions. To answer, the network cannot rely on local texture alone — it must, e.g., know that a bus window sits above a bus wheel, that grass lies below a cat. Relative position is a proxy for part-of-object reasoning.

Failure Modes

Shortcut learning in spatial pretexts. Both patch-location and jigsaw tasks admit *trivial solutions* that bypass semantics entirely. Neighboring patches can be located by matching *chromatic aberration* — the color fringing that camera lenses imprint as a consistent displacement between color channels — or by matching the pixel-level statistics of the shared boundary (edge continuation), or even JPEG block artifacts, none of which requires object understanding. The standard fixes: leave a *gap* between patches (no shared boundary), jitter each patch independently (break pixel-level alignment), and strip or decorrelate low-level color statistics. The general lesson exceeds these two tasks: *any signal correlated with the pretext label will be found by gradient descent*, so designing a pretext task is inseparable from adversarially auditing it for shortcuts. A second, subtler

failure is *task–distribution shift*: the network sees pairs of offset patches (or shuffled tiles) during pretraining, but will be evaluated on intact images — the pretext input distribution is not the deployment distribution, and part of the transfer gap in Figure 8.17 is attributable to this mismatch.

8.6.2 Jigsaw Puzzles

Noroozi and Favaro (2016) generalized the pair task: split the image into a 3×3 grid of tiles, shuffle, and ask the network to *recover the permutation* (Lecture 06, slide 17). With $9! = 362{,}880$ permutations this is intractable as classification and also unnecessary: the method restricts to a small set of permutations (100 in the figure; the paper's best setting used 1,000) chosen to be maximally distant from each other (selected iteratively by Hamming distance over permutation vectors), so classes are well-separated. Each tile is processed by a *shared-weight* convolutional tower (a context-free network: tiles never see each other), the nine feature vectors are concatenated, and fully connected layers produce a softmax over the permutation classes. Sharing weights across tiles is the critical design choice — it forces the tower to extract features that are useful *for every* tile regardless of whether it currently sits top-left or bottom-right, i.e., object-part features rather than position-specific ones.

Where the useful features live. The jigsaw paper made an observation every later evaluation follows: features from the *middle* of the network transfer much better than features from the last layers (fc7–fc8 of its AlexNet).

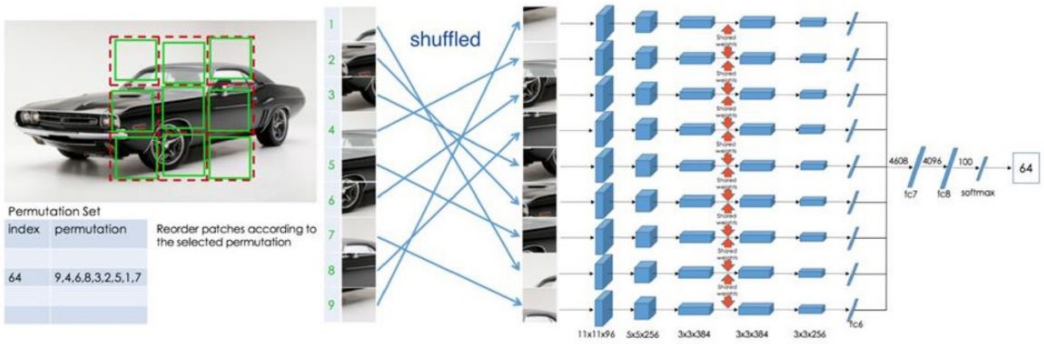
The reason is structural. The last layers are where the network actually solves the puzzle — turning nine tile features into one permutation number, a task with no value anywhere else. Going deeper, features shift from "useful in general" to "useful for this puzzle". That is why the RotNet table reports fc6–8 columns separately. The rule it left behind: *evaluate where general features live, not where the pretext was solved.*

8.6.3 Colorization and Split-Brain

Zhang et al. (2016) turned grayscale conversion into supervision. Every color image provides a free input–target pair: take the lightness channel L as input $\boldsymbol{x} \in \mathbb{R}^{H \times W \times 1}$ and ask the network to predict the opponent color channels (a, b),

$$\mathcal{F} : L \mapsto \widehat{ab}, \qquad (L, \widehat{ab}) \mapsto \text{color image},$$

where (a, b) encode chroma in the Lab color space — chosen because lightness is exactly the information the grayscale input keeps, so the prediction target is cleanly separated from the input. Recovering color requires semantic reasoning: what makes a plausible fish color is that the region is a fish; grass, sky, and skin have characteristic priors. Two implementation points matter. First, color prediction is *multimodal* (a plausible shirt could be red or blue), so the



(Image source: Noroozi & Favaro, 2016)

Figure 8.11. The jigsaw-puzzle pipeline of Noroozi and Favaro (2016). *Left*: the image is cut into a 3×3 grid of tiles, which are reordered by one permutation (here index 64) drawn from a pre-computed set of maximally distant, Hamming-selected permutations (bottom left). *Right*: every shuffled tile passes *independently* through the same convolutional tower — the "shared weights" markers at each stage (conv 11×11 down to fc6) mean no tile ever sees its neighbors, which is what forces the tower to learn object-part features valid at any position. The nine fc6 vectors are concatenated and fed through fc7–fc8 to a softmax over the permutation classes (100 in this figure), which must output the index, 64. The plumbing also shows the layer-specialization finding of the text: semantics live in the shared tower, while fc7–fc8 merely solve the puzzle — hence transfer probes are read from the tower, not from the head.

Source: Lecture 06, slide 17; architecture graphic after Noroozi and Favaro (2016) (via the CS231n course slides shown in the lecture).

paper predicts a *distribution over quantized* (a, b) *bins* (multinomial classification) rather than regressing a point value, and re-weights rare colors to avoid the desaturated average. Second, the features transfer — colorization rows appear throughout the transfer table of Figure 8.17. The *Split-Brain* variant (Zhang et al., 2017) turns the idea into an architectural principle: split the input channels into disjoint groups and predict each group from the other (L from ab, ab from L), then concatenate the two half-network representations. Cross-channel prediction generalizes beyond color (e.g., predicting depth from RGB), anticipating the cross-modal prediction methods of Chapter 12.

8.6.4 Video Colorization: The Emergent-Capability Moment

Vondrick et al. (2018) lifted colorization from images to video and, in passing, produced one of the canonical demonstrations that self-supervised pretraining can yield capabilities nobody trained directly. The task (Lecture 06, slides 19–20): given a *color* reference frame at time $t = 0$ and *grayscale* frames at $t = 1, 2, 3, \ldots$, color the gray frames. Color is temporally coherent — the red car does not change color between frames — so a good strategy is to *copy color from wherever the same stuff is* in the reference frame. The method implements exactly that with a soft attention mechanism: for each pixel (or patch) $\boldsymbol{q}$ in a gray target

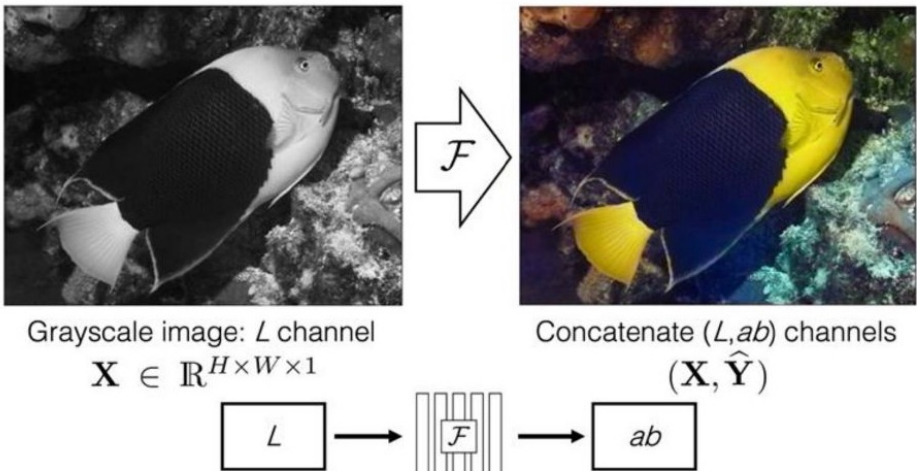


Figure 8.12. Colorization as self-supervision, after Zhang et al. (2016). The input is the lightness channel alone, $\boldsymbol{x} \in \mathbb{R}^{H \times W \times 1}$ (left, the grayscale fish); the network $\mathcal{F}$ must predict the missing opponent channels $\widehat{ab}$, which are concatenated with L to reassemble the color image (right). Since (L, ab) pairs come free with every color photograph, the supervision costs nothing — and the task is hard in a useful way: painting a plausible yellow head onto this fish requires knowing it *is* a fish. The prediction is inherently multimodal (many colorings are correct), which is why $\mathcal{F}$ outputs a distribution over quantized ab bins rather than a point regression (8.6.3).

Source: Lecture 06, slide 18; panel after Zhang et al. (2016) (image credit on the slide: Richard Zhang / Phillip Isola).

frame, compute its embedding, retrieve the most similar embeddings $\boldsymbol{k}_i$ from the reference frame, and transfer their colors by a similarity-weighted combination

$$\hat{\boldsymbol{y}}(\boldsymbol{q}) \;=\; \sum_i \alpha_i \, \boldsymbol{y}(\boldsymbol{k}_i), \qquad \alpha_i \;=\; \mathrm{softmax}_i\big(\mathrm{sim}(\boldsymbol{q}, \boldsymbol{k}_i)/\tau\big),$$

where $\boldsymbol{y}(\boldsymbol{k}_i)$ is the reference color at key location i, α_i is the attention weight, sim is the cosine similarity of the notation table, and τ a temperature. Because the supervision is the *true color* of the target frame (free: the video was color all along, we only hid it), the network is trained end-to-end to make its attention weights land on the physically corresponding region.

The payoff: tracking for free. The attention weights α_i were trained only to copy *colour* from one frame to the next. But to copy colour correctly they had to find *which pixel went where* — and that is a physical fact, not a colour fact.

So after training, swap colour for any per-pixel label — a segmentation mask through a judo match, human skeleton keypoints through an activity video — and the frozen network *tracks* them, with no further training (Lecture 06, slides 21–22; Figure 8.13).

This is the course's first *emergent capability*: an ability that was never in the loss appears because the proxy task could only be solved by solving the real problem underneath. DINO's

Figure 8.13. Emergent tracking from video colorization, after Vondrick et al. (2018). The attention weights of (8.6.4) — trained only to copy color — point from each pixel of a target frame to the corresponding region of the reference frame. Replacing color with a task-specific label (a segmentation mask on the grapplers; human skeleton keypoints) and copying it along the same attention turns the frozen model into a zero-shot tracker: the mask and the skeleton propagate through subsequent frames with no further training.

attention-based segmentation (Chapter 11) and DIFT's correspondences (Chapter 7) are later examples.

8.6.5 Temporal Order

Video also supplies order supervision with zero labels. *Shuffle-and-verify* (frame-order verification) presents short frame tuples and asks whether they are in correct temporal order or shuffled; a sibling task asks the network to identify the *odd one out* among clips, one of which has scrambled frames (Lecture 06, slide 23). Ordering requires models of *how the world evolves*: people fall down, not up; a thrown ball has a trajectory. These video pretexts anticipate the video contrastive methods of Chapter 9, and temporal order is a special case of the general plan "predict something about the augmentation."

The shortcut in video: motion blur. Fast-moving objects leave streaks that show how and how fast things were moving. Combined with other low-level cues across frames, they can let a network order frames without learning anything about how the world changes.

The defences follow the usual pattern: heavy spatial augmentation (crops, flips) scrambles the streaks, and sampling frames at irregular intervals makes them unreliable. The general law holds in every modality: *any signal correlated with the pretext label will be found* — chromatic aberration for patch location, motion blur for frame order, interpolation artefacts for rotation.

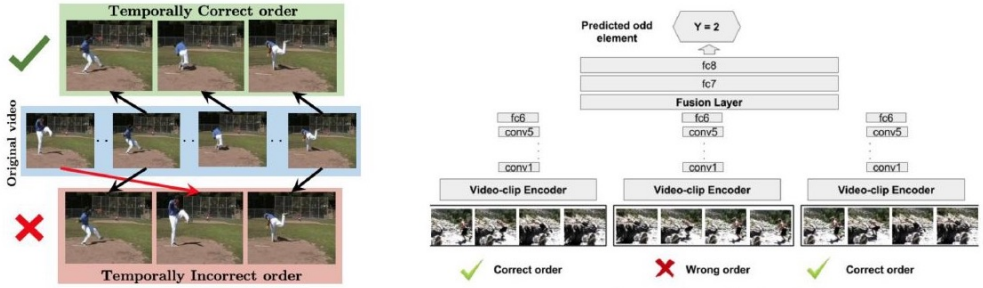


Figure 8.14. Two temporal-order pretexts for video. *Left* (shuffle-and-verify, Misra et al. 2016): a short tuple of frames is either in temporally correct order (green check) or shuffled (red cross), and the network must tell which — a binary label that comes free with every clip. *Right* (odd-one-out, Fernando et al. 2017): several clips are shown, exactly one with scrambled frames; shared-weight video-clip encoders feed a fusion layer whose top fully connected stage predicts the index of the odd clip ($Y = 2$ in the example). Both tasks turn the arrow of time into a classification label — and both invite the motion-blur shortcut analyzed below: blur streaks already reveal how things were moving, without any model of how the world evolves.

Source: Lecture 06, slide 23; panels after Misra et al. 2016 and Fernando et al. 2017 as shown in the lecture.

8.7 Worked Example: RotNet

We now walk through one complete prediction pretext, end to end, because it is the cleanest, the most reproducible, and the one whose transfer numbers anchored the field's first serious evaluation table (Gidaris et al., 2018) (Lecture 06, slides 24–28).

8.7.1 The hypothesis: visual commonsense

To recognize that a photograph is rotated by 90°, a model must already know what the object in it looks like upright. A sideways bus is recognized as sideways because buses have a canonical orientation. RotNet's claim is that this *visual commonsense* — knowing how objects are usually posed — is exactly the semantic content we want features to carry, and it can be extracted by a classifier with no labels other than the rotation applied.

Learning note from study sessions

A student midway through the lectures on which Part III is based asked for this section to go deeper — the only mid-course interruption of the whole arc. The depth below (right-angle mechanics, the transfer-table reading, the symmetry caveat) is the direct answer to that request.

Why "which way is up?" teaches anything. Most photographs have a ground, gravity, and objects with a natural orientation: eyes above mouths, tree trunks below leaves, buses on their wheels. In ordinary photo collections this holds almost always, so "which way is up" is a reliable *statistical* fact about images — and one available for free.

RotNet's bet is that answering it requires knowing what objects look like. To say "this bus is lying on its side", the network needs some idea of what an upright bus is. *Recognising upright requires knowing the upright thing.*

By the same token we can read off where the task must fail. Two image classes supply no uprightness signal: top-down aerial or satellite imagery, where there is no gravitational pole and the compass is in the scene rather than imposed on it, and rotationally symmetric objects — a ball, a circular clock face, many textures — where "up" is undefined even in the physical world. For such images the rotation classifier is at chance not because it learned nothing, but because the signal itself vanished. Exercise 12 will return to this as an example of matching the pretext family to the corpus.

8.7.2 Method

RotNet's task. For each image $\boldsymbol{x}$, make *four* training inputs by rotating it by 0°, 90°, 180° and 270°:

$$\tilde{\boldsymbol{x}}_y \;=\; g(\boldsymbol{x}, y), \qquad y \in \{0, 1, 2, 3\},$$

where $g(\boldsymbol{x}, y)$ rotates $\boldsymbol{x}$ by $90y$ degrees. The label y is free — we know which rotation we applied. One ConvNet $f_{\boldsymbol{\theta}}$ with a four-way softmax is trained to predict it with ordinary cross-entropy:

$$\mathcal{L}_{\text{rot}}(\boldsymbol{\theta}) \;=\; -\,\mathbb{E}_{\boldsymbol{x},y}\Big[\log\big(\text{softmax}\big(f_{\boldsymbol{\theta}}(g(\boldsymbol{x}, y))\big)_y\big)\Big].$$

Figure 8.15 draws the whole loop. Two design choices deserve attention.

Why $\{0, 90, 180, 270\}$ and not arbitrary angles? Right-angle rotations are *exact*: a quarter-turn maps pixels to pixels with no resampling. Arbitrary-angle rotation would require interpolation and would twist boundary pixels in ways that differ systematically by angle — the network could then key on interpolation artifacts instead of content, a shortcut. Consider the failure explicitly. Rotating a square canvas by a non-multiple of 90° does two damaging things at once: it resamples the image through subpixel interpolation (bilinear or bicubic), producing a characteristic blur whose strength depends on the angle, and it vacates the frame's corners, which must be filled by padding or cropping. Both traces are low-level and instantly readable by a convolutional network — the blur level and the black-corner mask are cheap proxies for the angle, so the loss stops falling without any detector for eyes or wheels. And 45°-spaced classes are separately ill-posed: adjacent angles are hard for humans to tell apart, let alone a softmax. The four-way choice is therefore not aesthetics but shortcut defense: the label must be readable only from what the picture *is*, never from *how it was resampled*.

Why four samples per image? Drawing all four rotations guarantees that the same content appears under every label, so the label cannot be guessed from content identity; only

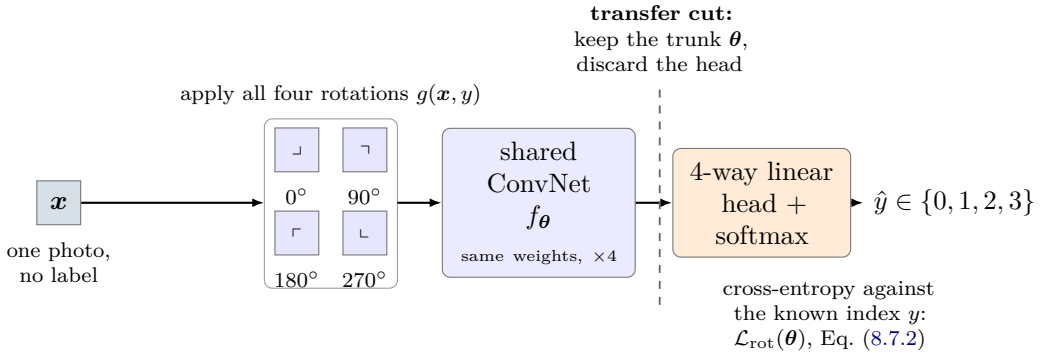


Figure 8.15. The complete RotNet training loop. One unlabeled photo $\boldsymbol{x}$ spawns four training samples: $g(\boldsymbol{x}, y)$ rotates by $90y$ degrees as a pure index permutation (transpose plus flips), so no resampling artifact can leak the label. A single shared-weight ConvNet $f_{\boldsymbol{\theta}}$ processes all four, and a 4-way softmax head is trained by ordinary cross-entropy against the known index y (8.7.2). At transfer time the dashed cut applies: the head that played the guessing game is discarded, and the trunk $\boldsymbol{\theta}$ is kept as the feature extractor whose quality Table 8.2 measures.

How to read it: Notice what is absent: labels, a decoder, and anything image-specific beyond the four rotations. The entire supervision lives in the boxed column on the left — which is why Section 8.7.2 spends its care on exactly that step.

pose carries signal. The loss then has nowhere to hide: any shortcut to high accuracy must expose y itself as visible in the image, and since all four rotations of the same $\boldsymbol{x}$ receive the full content, that visibility is exactly the uprightness cue. In implementation, the four rotations of (8.7.2) are pure index permutations — transpose plus flips of the pixel grid — rather than resampling operations, so the training batch simply becomes four copies of the input with labels $\{0, 1, 2, 3\}$.

Worked Example — RotNet on one image

Data: one unlabeled photograph $\boldsymbol{x}$ of a bird on a branch.
Pretext input generation (8.7.2): $\tilde{\boldsymbol{x}}_0 = \boldsymbol{x}$ (label 0), $\tilde{\boldsymbol{x}}_1 = \boldsymbol{x}$ rotated 90° (label 1), $\tilde{\boldsymbol{x}}_2$ rotated 180°, $\tilde{\boldsymbol{x}}_3$ rotated 270°.
Model: AlexNet-style ConvNet with a 4-way linear head.
Loss: average cross-entropy over the four samples (8.7.2). At initialization the head is at chance (25%). After training, the head exceeds 90% on held-out images — but that is *not* the metric we care about.
Evaluation: throw the head away. Freeze the conv trunk, extract features, and train linear classifiers on them for new tasks (Figure 8.17). The pretext accuracy on the rotation task is a means; the deliverable is the trunk.

The worked example hides an important asymmetry: the head's 4-way accuracy and the trunk's downstream quality are different objects, and they need not move together. A perfectly accurate rotation classifier could, in principle, have learned nothing generalizable (if,

Figure 8.16. First-layer ConvNet filters learned (a) with ImageNet labels and (b) self-supervised by rotation prediction, redrawn after Gidaris et al. (2018). Both sets are dominated by oriented edge and color-contrast detectors; the low-level structure of vision is recoverable without labels.

say, it exploited a blur shortcut the linear probe cannot use); conversely, a middling classifier could harbor excellent mid-level features. The evaluation ladder of Section 8.8 exists precisely to separate these two numbers — which is why the transfer table, not the training curve, is the deliverable of the RotNet story.

8.7.3 What the network actually learns

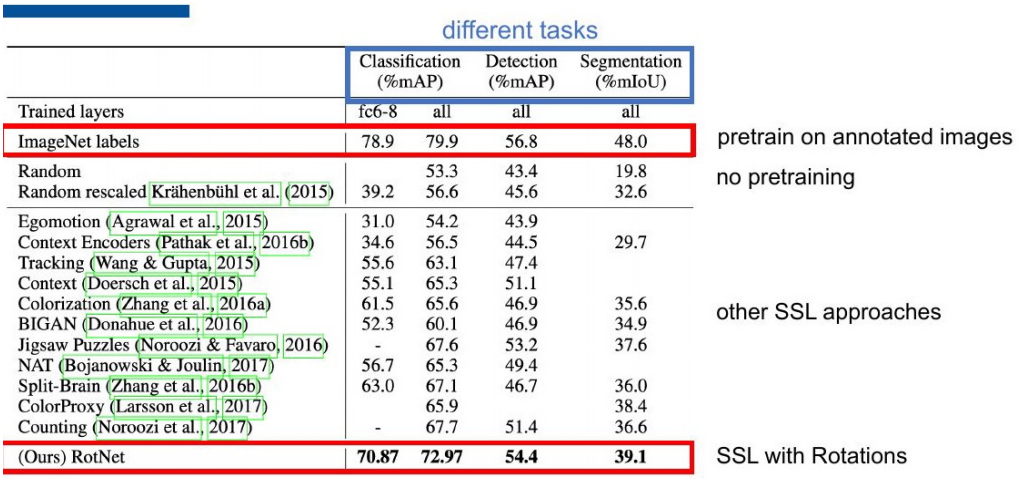
The first sanity check is qualitative, and it is one of the lecture's quiet pivots (slide 27): visualize the first-layer filters and compare a supervised AlexNet against the self-supervised rotation network. Both converge to the same *kind* of filters — oriented, Gabor-like edge detectors and color-blob detectors; the self-supervised grid is, if anything, denser. The message: the low-level vocabulary of vision is not a property of labels; it is a property of images and the tasks that reason about them. We will reuse this evidence in Chapter 9 when comparing filter-level and feature-level evaluation.

What the filter grid really shows. Supervised AlexNet was never told to learn edge detectors — nothing in the ImageNet loss mentions edges. They appear because recognising objects needs them. RotNet gets the same detectors from the *images* and the *task* alone, no labels: oriented edge detectors and colour-contrast detectors, much like early stages of biological vision.

So low-level vision seems to come from the data itself; only the deeper layers depend on the supervision (the transfer table below shows them diverging).

Two cautions:

- Looking at filters is *direct inspection* — suggestive, never proof.
- But it is a useful cheat detector. A network exploiting the blur shortcut of Section 8.7.2 would show broad, artefact-hunting filters instead (Section 8.8).



	Classification (%mAP)	Detection (%mAP)	Segmentation (%mIoU)		
Trained layers	fc6-8	all	all	all	
ImageNet labels	78.9	79.9	56.8	48.0	pretrain on annotated images
Random		53.3	43.4	19.8	no pretraining
Random rescaled Krähenbühl et al. (2015)	39.2	56.6	45.6	32.6	
Egomotion (Agrawal et al., 2015)	31.0	54.2	43.9		
Context Encoders (Pathak et al., 2016b)	34.6	56.5	44.5	29.7	
Tracking (Wang & Gupta, 2015)	55.6	63.1	47.4		
Context (Doersch et al., 2015)	55.1	65.3	51.1		
Colorization (Zhang et al., 2016a)	61.5	65.6	46.9	35.6	other SSL approaches
BIGAN (Donahue et al., 2016)	52.3	60.1	46.9	34.9	
Jigsaw Puzzles (Noroozi & Favaro, 2016)	-	67.6	53.2	37.6	
NAT (Bojanowski & Joulin, 2017)	56.7	65.3	49.4		
Split-Brain (Zhang et al., 2016b)	63.0	67.1	46.7	36.0	
ColorProxy (Larsson et al., 2017)		65.9		38.4	
Counting (Noroozi et al., 2017)	-	67.7	51.4	36.6	
(Ours) RotNet	**70.87**	**72.97**	**54.4**	**39.1**	SSL with Rotations

Figure 8.17. Transfer of features pretrained on ImageNet, redrawn after Gidaris et al. (2018). Rows are pretext/supervision sources; the columns are PASCAL-VOC-style classification (linear probes on the frozen trunk, both mid-level fc6–8 and all-layers fine-tune), detection, and segmentation. RotNet (bottom row) outperforms every prior self-supervised method on classification and detection — but remains well below the fully supervised ImageNet-labels ceiling. The figure is the field's first widely-shared scoreboard.

8.7.4 Results: the first honest transfer table

RotNet's real legacy: how it was evaluated. Pretrain without labels on ImageNet images, then test the features on labelled benchmarks — classification, detection, segmentation — and compare against both the fully supervised result and every earlier self-supervised method, under the *same* protocol (Figure 8.17, Table 8.2). Two choices make that table trustworthy:

1. **Several depths of evaluation** — frozen probes on fc6–fc8, then full fine-tuning. That separates "the features are good" from "the weights are a good starting point", so a method cannot look strong merely because it fine-tunes well.
2. **Everything else held fixed** — same kind of backbone, same unlabelled images. Differences between rows therefore come from the *pretext task*, not the data.

That protocol gave the field a common yardstick.

Two readings of Table 8.2 matter for the rest of the book. *First*, the ordering: a deliberately simple task (predict the quarter turn) beats the sophisticated reconstruction and patch tasks of Sections 8.5 and 8.6 — evidence that what matters is less the cleverness of the label and more whether the label forces genuine scene understanding at the right abstraction. *Second*, the gap: RotNet leaves roughly 6–8 mAP points to the full ImageNet-labels ceiling on classification. The whole next phase of the field — contrastive learning, Chapters 9–11 — is the effort to close that remaining gap without labels. Hold this number in mind as the baseline against which SimCLR, MoCo, and DINO declare victory.

Table 8.2. Selected transfer results redrawn from the RotNet transfer table (Lecture 06, slide 28; numbers from Gidaris et al. (2018)). Columns: PASCAL classification with linear probing (fc6–8) and all-layers fine-tune (% mAP); detection (% mAP); segmentation (% mIoU). Dashes: not reported in the source table.

Pretraining	Classif. fc6–8	Classif. all	Detection	Segmentation
ImageNet labels	78.9	79.9	56.8	48.0
Random (no pretraining)	–	53.3	43.4	19.8
Context Encoders (Pathak et al., 2016)	34.6	56.5	44.5	29.7
Patch location (Doersch et al., 2015)	55.1	65.3	51.1	–
Colorization (Zhang et al., 2016)	61.5	65.6	46.9	35.6
Split-Brain (Zhang et al., 2017)	63.0	67.1	46.7	36.0
Jigsaw (Noroozi and Favaro, 2016)	–	67.6	53.2	37.6
RotNet (Gidaris et al., 2018)	**70.87**	**72.97**	**54.4**	**39.1**

How to read the RotNet table. Read it in two directions.

Within one row: fine-tuned versus frozen. Fine-tuning every layer brings everyone close together — even random initialisation reaches 53.3, against 79.9 for supervised — because downstream training reshapes the features. The frozen probes (fc6–8) are the honest test of what the features already know, and there the older self-supervised methods fall far behind: Context Encoders 34.6, patch location 55.1, colourisation 61.5. The drop from "all layers" to "frozen" tells you how much of a method's quality was really in the features, and how much was only a good starting point for fine-tuning.

Across rows: RotNet versus the rest. RotNet is the first method that stays strong when frozen — 70.87, second only to supervised. Its features are already organised by meaning, with no downstream training: the claim this chapter set out to test. It beat the more elaborate jigsaw and inpainting methods with a much simpler task. Its remaining gap of 6–8 mAP to supervised is the number the contrastive methods of Chapter 9 set out to close.

8.8 Judging What We Learned: An Evaluation Segue

Figure 8.17 quietly forced a methodological question that the lecture answers next (slides 29–37): how should one *measure* the quality of an unlabeled representation? Chapter 1 introduced the evaluation protocols; here we only name them precisely and flag which appear in this chapter, deferring the full treatment to the evaluation discussion of Chapter 1 and its return in Chapter 16's method comparisons.

The lecture splits evaluation into two families.

Look at it directly: plot the features with PCA or t-SNE, show attention maps, or display the first-layer filters. Good for intuition; gives no number.

Use it for a task, in increasing order of how much the evaluator is allowed to adapt:

- k**NN** — no training at all; "a head of capacity zero".
- **Linear probe** — freeze the network, train one linear layer on a small labelled set. Measures what is *linearly readable*; the field's default.
- **Small non-linear head** — accepts features that are present but not linearly separable.
- **Fine-tuning** — unfreeze everything. Measures how good a *starting point* the network is, which mixes feature quality with how easy it is to keep training.

The RotNet results already used three of these types without naming them:

- the filter grid (Figure 8.16) — direct inspection;
- the fc6–8 columns of Table 8.2 — linear probing;
- the all-layers columns — fine-tuning.

The keypoint tracking of Figure 8.13 is a fourth kind the taxonomy barely covers: *emergent behaviour*. No head, no labelled data — just read an internal quantity (attention) as the answer to a task the network was never trained on. As methods rely less on heads, this kind of evaluation becomes common (DINO's attention maps, Chapter 11).

A caution the slides leave between the lines: linear probes measure *linear separability*, not usefulness. A representation can be "worse" by the probe yet better for fine-tuning, clustering, or nearest neighbors — one reason the field now reports several protocols per method, and a theme Chapter 16 picks up when comparing families.

What each protocol actually tests. Line the protocols up by how much the evaluator is allowed to learn, and each answers a different question:

Protocol	Trained parameters	Question it answers
kNN	none	Are same-class images already neighbours?
linear probe	one layer	Can one matrix read the class out?
fine-tuning	everything	Is this a good *starting point*?

A model can be excellent as a starting point yet poor under a linear probe — which is exactly the within-row gap in Table 8.2. That is why careful evaluation is half of any self-supervised result: a claim is only as strong as the protocol that tested it.

Learning note from study sessions

The study session spent a long exchange on exactly this ladder — why a model with tangled features and a great initialization can match supervised baselines after fine-tuning while scoring near-random under a linear probe. The capacity spectrum above is the distilled answer.

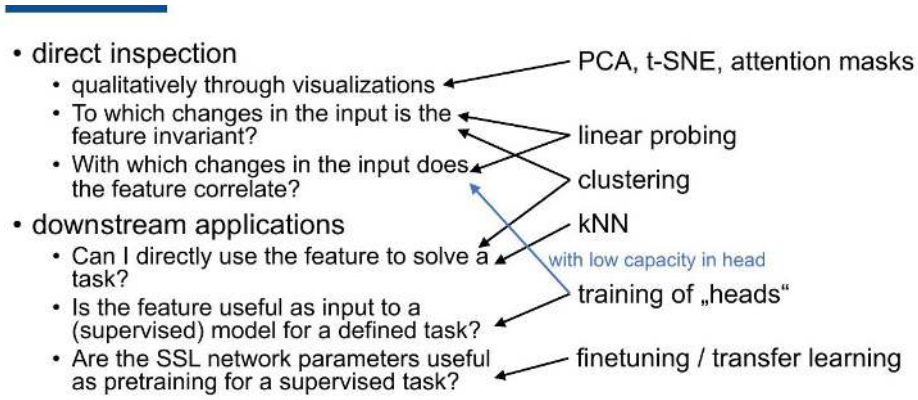


Figure 8.18. The evaluation taxonomy of Lecture 06 (slides 29–33). Left: two families — direct inspection (visualizations of filters, PCA/t-SNE embeddings, attention maps) and downstream use (features directly with kNN, a trained head / linear probe, or fine-tuning as initialization). Right: the protocol ladder ordered by how much target supervision each consumes — zero-shot emergent evaluation, kNN, linear probe, head, fine-tune.

Common Confusion

Pretext accuracy versus representation quality. "The model reaches 90% at predicting rotations, so the representation is good" is a fallacy. The pretext metric is at best a diagnostic: driving rotation accuracy to 100% can even be *anti*-correlated with transfer, since perfect rotation classification may be achievable through shortcut features (such as the interpolation artifacts of Section 8.7.2, if arbitrary angles were used) that carry no semantics. The representation's quality is only ever established downstream — hence the evaluation ladder of Figure 8.18. The contrastive methods of Chapter 9 inherit this and add a second trap: their training loss InfoNCE is itself an evaluation-shaped quantity (a cross-entropy for picking the positive out of the negatives), which makes it even easier to mistake a well-optimized loss for a good representation.

Failure Modes

Augmentations as a source of failure. The augmentation family is the weakest, least-principled link in the SSL pipeline, and two failure patterns recur. (i) *Destroying the object:* a random crop that lands off-center and contains only background produces a "view" whose correct embedding is "clutter"; the model is then told (by any instance-level method) to make this embedding equal the intact object's — teaching nonsense. Extreme rotation/flip parameters do the same for arbitrary natural images. (ii) *Destroying the label:* photometric augmentations erase exactly the channels a downstream task needs. The canonical case is color: jittering hue or converting to grayscale breaks features for any class that *is* a color — ripe from unripe fruit, team jerseys, traffic-light states, blood oxygenation in medical images. The general law: every entry of $\mathcal{T}$ is a bet that the downstream tasks are invariant to that transform; the bet is invisible at pretraining time

and only fails at the downstream evaluation, which is why augmentation audit must be part of SSL evaluation, not an implementation detail.

8.9 Closing the Arc: From Predicting the Augmentation to Ignoring It

We have now seen both answers to the question of Section 8.4. Reconstruction (Section 8.5) asks the network to undo the augmentation; prediction (Section 8.6) asks it to identify the augmentation. RotNet's success validates the second answer — but it forces features to carry rotation information, while semantic tasks would rather discard it. PIRL (Misra and van der Maaten, 2020) is the closing step of this chapter's arc, and we walk through it in three steps: the architectural compromise, then the loss, then why it closes the story.

8.9.1 The compromise: keep jigsaw, throw away jigsaw

Misra and van der Maaten (2020) resolve the tension by splitting the architecture's responsibility: keep a pretext task (their choice is jigsaw), but train the *representation* to be invariant to the transform — an image I and its transformed version I^t should produce similar features — while pushing features of different images apart. The machinery (a memory bank of negative samples, an NCE loss, a separate head g absorbing the transform-specific processing) delivers precisely what this hypothesis predicts: features that out-transfer rotation prediction across classification, detection, and low-shot benchmarks, as verified in the lecture's Table 2 comparison. PIRL is the pivot into Chapter 9: once we are matching positive pairs against negatives with an NCE loss, the pretext task has become a contrastive task, and the specific transform family becomes an augmentation choice rather than the supervision itself.

Two details of PIRL. **Two heads, not one.** With a single shared head, the backbone alone would have to absorb the difference between a clean image and a scrambled one. Giving the scrambled branch its own small head g lets that head soak up the jigsaw-specific quirks, so the shared backbone can stay focused on meaning.

The memory bank. PIRL needs thousands of negatives per step. Recomputing their features every step would be far too slow, so it stores one 128-number vector per training image and refreshes each one only occasionally. The cost: stored vectors grow *stale* as the encoder changes. Fixing that staleness is exactly why MoCo (Chapter 9) introduced its momentum encoder and queue.

8.9.2 The loss

Concretely, the NCE objective (Gutmann and Hyvärinen, 2010) scores a positive pair (I, I^t) against negatives I' drawn from the dataset,

$$h(\boldsymbol{v}_I, \boldsymbol{v}_{I^t}) \;=\; \frac{\exp\big(\mathrm{sim}(\boldsymbol{v}_I, \boldsymbol{v}_{I^t})/\tau\big)}{\exp\big(\mathrm{sim}(\boldsymbol{v}_I, \boldsymbol{v}_{I^t})/\tau\big) \;+\; \sum_{I' \in \mathcal{D}_N} \exp\big(\mathrm{sim}(\boldsymbol{v}_I, \boldsymbol{v}_{I'})/\tau\big)}, \tag{8.1}$$

where $\boldsymbol{v}_I$ is the projected feature of image I, $\mathcal{D}_N$ is a set of negative images, sim is cosine similarity, and τ is a temperature. So $h \in (0, 1)$ is the model's probability that I^t, and not one of the negatives, is the true partner of I. Training *minimizes* the loss $\mathcal{L} = -\log h$ (equivalently, it maximizes $\log h$):

h	0.99	0.5	0.1	0.01
$-\log h$ (loss)	0.01	0.69	2.30	4.61

The loss is 0 only when $h = 1$ and grows without limit as $h \to 0$. This is ordinary cross-entropy for an $(N{+}1)$-way question — "which candidate is the true partner of I?" — with the positive as the correct class and sim $/\tau$ as the logits. The numerator is the "similarity of the positive pair"; the sum in the denominator is the "similarity to negative samples" (Figure 8.19). Chapter 9 takes exactly this equation, removes the pretext machinery, and builds the modern field on it.

8.9.3 Why this closes the arc

Stepping back. The chapter asked one question — *what should an encoder do with a transformed input?* — and found three answers:

- **Undo it** (reconstruction: Context Encoders, colourisation).
- **Name it** (prediction: patch location, jigsaw, RotNet).
- **Ignore it** (PIRL): keep the content, discard the transformation.

Each answer decides how much of the representation is spent on the transformation and how much on the content. RotNet must spend some on the rotation, because its loss asks for it; PIRL is penalised for telling I and I^t apart, so it spends none.

The general lesson: *a pretext task is an incentive, and the network learns whatever satisfies that incentive most cheaply* — blind spots included (Section 8.3). Designing a good pretext task means first imagining the cheapest wrong way to solve it, then making sure that way fails.

Chapter 9 starts from here: if ignoring the transformation (PIRL) beats naming it (RotNet), then the classification head was never the point. Augmentation plus a contrastive loss is.

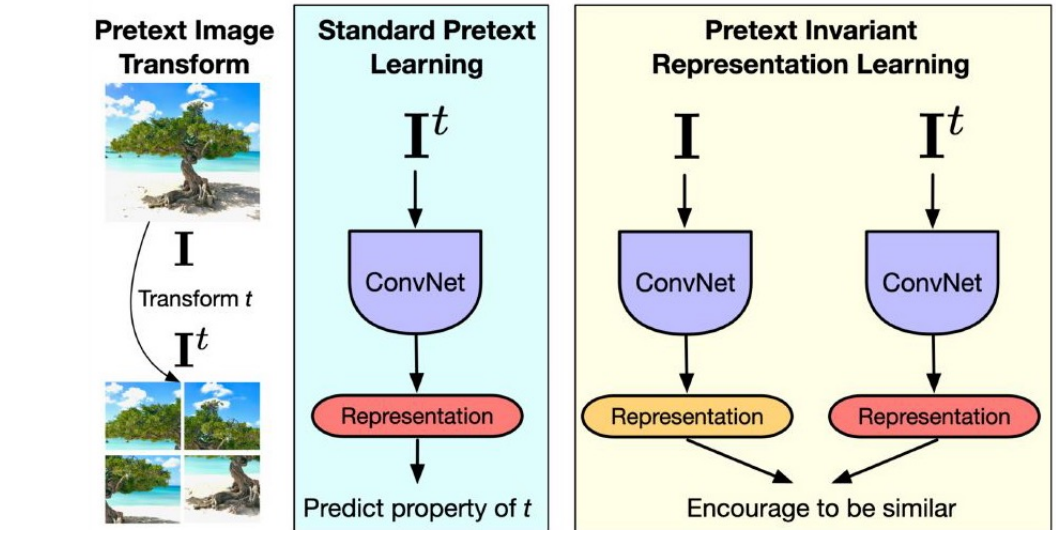


Figure 8.19. Pretext-invariant representation learning as the bridge from this chapter to the next, after Misra and van der Maaten (2020). *Left* (RotNet, Section 8.7): predict a property of the transform — the representation co-varies with the augmentation. *Right* (PIRL): encode the clean image I and its transformed version I^t with a shared trunk $\boldsymbol{\theta}$, but push their projected features together (positive pair) while moving them apart from negatives $\boldsymbol{v}_{I'}$ drawn from a memory bank — the representation is trained to *discard* the transform. The NCE loss (8.1) supplies the push–pull dynamics.

Check Your Understanding

Level 1 (recall).

1. Give the two senses of "invariant" used in this chapter (undo the augmentation; make features agree across augmentations) and one method of each kind.
2. Why are rotations restricted to $\{0, 90, 180, 270\}$ in RotNet rather than arbitrary angles?
3. In the colorization pretext, why is the target formulated in Lab space as a distribution over quantized ab bins rather than regressed as two continuous channels?

Level 2 (comprehension).

4. Relative-patch-location and jigsaw are predict-the-augmentation tasks. Name two low-level shortcuts a network could use instead of scene understanding, and one design fix for each.
5. In the NCE objective (8.1), which term corresponds to "similarity with the positive pair" and which to the negatives? What happens to the loss as the temperature $\tau \to 0$?

6. The lecture slides report that RotNet's latent features, probed linearly, beat every prior SSL row of Table 8.2 yet remain below the ImageNet labels ceiling. State the gap in one sentence as a motivation for contrastive learning.

Level 3 (application/transfer).

7. A researcher proposes color jitter (hue rotation) as an augmentation for a lepidopteran (butterfly/moth) classifier whose classes are defined largely by wing coloration and pattern. Predict the failure mode, and design a verification experiment that would detect it from pretraining artifacts before any downstream evaluation.
8. Video colorization (Section 8.6.4) copies color along learned attention and yields zero-shot tracking. Explain why the attention cannot be solved by low-level motion cues alone, and why the emergent skill is tracking of *regions*, not just pixels.
9. You want an SSL representation for time series of industrial pumps whose fault labels manifest mainly as changes in a specific frequency band's amplitude. Which entries of Table 8.1 are safe, which are dangerous, and why?

Level 4 (analysis).

10. RotNet's premise is "visual commonsense." Construct an argument that the *same* premise, taken to its logical conclusion, predicts RotNet should underperform contrastive methods on transfer — and reconcile this with RotNet's strong showing.
11. Formalize the statement at the end of Section 8.4: "a representation maximally invariant to t provably throws away information about t." Use the mutual-information notation $I(\boldsymbol{z}; y)$ with y the augmentation parameter, and state the trade-off that augmentation design then becomes.
12. Table 8.2 compares methods pretrained on ImageNet only. Predict two distinct reasons that a method's ranking on this exact table may not transfer to pretraining on a different unlabeled corpus (e.g., medical scans), and name the evaluation protocol most sensitive to each reason.

Level 5 (research-level). Continued in the breakable box below — the shared `exercisebox` environment is not breakable, and a box taller than a page would otherwise silently truncate its tail.

Check Your Understanding — Level 5 (research-level)

13. Video colorization's tracking strength scales with the difficulty of the color-prediction proxy. Propose an ablation to separate "correspondence from attention" from "correspondence from optical flow," and say what each arm would show if attention were merely recovering flow.
14. Design a pretext task for a modality of your choice (point clouds, graphs, code, audio) that (a) is not reconstruction and (b) is not instance discrimination, following from first principles the design constraints this chapter identifies. State what shortcut your task invites, and what architectural or data-processing choice removes it.

15. PIRL's claim is that for semantic tasks, invariance is preferable to covariance. Identify a concrete downstream scenario in which this claim *fails*, argue why, and sketch the minimal change to (8.1) that would restore covariance where you want it while keeping invariance elsewhere.

Solution sketches.

1. Undo: Context Encoders / inpainting (reconstruct the clean $\boldsymbol{x}$); agree: contrastive/PIRL – feature of $\boldsymbol{x}$ and $t(\boldsymbol{x})$ pulled together in Z.
2. Right angles are exactly representable without interpolation, so low-level resampling artifacts cannot become label cues; also separates classes maximally and keeps classification well-posed.
3. Color is multimodal (many plausible ab for one L); point regression averages modes to desaturated mush. Lab separates lightness (input) from chroma (target); quantized bins + rebalancing recover saturation.
4. Chromatic aberration or boundary continuation / JPEG lattice; fixes: enforce a gap between patches, jitter each patch, strip color statistics, and (for jigsaw) shuffle tiles only within a distant permutation set.
5. Numerator $\exp(\mathrm{sim}(\boldsymbol{v}_I, \boldsymbol{v}_{I^t})/\tau)$: positive pair; sum in the denominator: negatives. As $\tau \to 0$ the softmax sharpens to a hard max, concentrating gradient on the current hardest negative/positive comparison.
6. Something like: RotNet closes roughly half of the SSL-to-supervised classification gap of its day (frozen probe: from Split-Brain's 63.0 to 70.87, against 78.9 supervised), leaving a residual — $\sim$6–8 mAP on the classification probes — that contrastive methods then set out to remove.
7. Hue rotation destroys the class-defining signal, producing features that encode only shape and strip species identity. Detect by auditing: pretrain, freeze trunk, fit a linear probe on a few labeled wing crops, and report per-class recall for color-defined species — collapse here flags the augmentation before any large downstream run.
8. Optical flow gives motion of pixels, not region-level identity; attention trained on temporal color coherence must aggregate coherent region structure (the whole car stays red), so the emergent object is the region (and its mask/skeleton), which zero-shot propagation of masks and keypoints demonstrates.
9. Safe: time-shift, small jitter, window slicing (if windows keep the band), DC-shift only if it does not remove absolute level. Dangerous: band-stop filtering near the fault band, aggressive amplitude scaling or zero-masking that erases the band's amplitude.
10. If commonsense were sufficient, the task-specific supervision RotNet uses would outperform instance-level alignment — yet contrastive methods beat it. Reconcile: RotNet's four classes are coarse and its signal is easy to saturate; contrastive objectives scale the "commonsense" per-instance, which is the stronger constraint.

11. Maximal invariance to t is $I(\boldsymbol{z}; y) = 0$: the data-processing inequality then bounds anything downstream about t that $\boldsymbol{z}$ could enable. Augmentation design becomes allocation of $I(\boldsymbol{z}; \cdot)$ across factors — keep what tasks need, zero out the rest.
12. (i) Corpus/domain shift: linear probe ranking is driven by the match between pretraining distribution and probe corpus — most sensitive protocol: linear probe. (ii) Augmentation mismatch: e.g., rotations meaningless for axial CT — most sensitive protocol: rotation pretext itself / any protocol whose evaluation images break the assumed invariance, e.g. kNN with default metric.
13. Add an arm feeding *precomputed optical flow fields* as input instead of RGB; if attention-detection performance tracks the RGB arm qualitatively and improves with better color diversity but not with better flow, the correspondence is not merely recovered flow — control by degrading flow while holding color constant.
14. Example: point clouds — predict the rigid transform (SE(3) pose regression) between two partial views of the same underlying scene; shortcut: matching overlap by point-wise geometric features, fixed by asymmetric partial cropping and removing low-level overlap cues; the task forces view-invariant geometry understanding without reconstruction.
15. Fails when the transform is the label: orientation-sensitive tasks (e.g., gait asymmetry from video, "is this photo upside down", reading rotated text). Restore covariance by adding a per-transform attribute head before the contrastive pooling, or by including transform-classification as an auxiliary loss on a parallel branch, so the shared trunk can still discard it where the semantic objective wants invariance.

Concept Map — how this chapter connects

Backward. The augmentation primitives extend L03's autoencoders and L05's denoising/masking (Chapters 4, 5, 7): masking is an augmentation here. The distance and clustering machinery of Chapter 2 (cosine similarity, kNN, k-means pseudo-labels, probing protocols) is the evaluation language of Section 8.8. The generative view that "good internals come from modeling the data" (Chapters 4–7) is what made looking for features outside generators natural.

Sideways. RotNet's transfer table sets the quantitative bar that the next three chapters are judged against.

Forward. Chapter 9 generalizes PIRL's NCE machinery into SimCLR and MoCo — same pair-alignment, but with the pretext reduced to augmentation alone, a projection head, and explicit negatives — and derives its mutual-information lower-bound reading of InfoNCE. Chapter 10 removes negatives entirely (SimSiam, BYOL, Barlow Twins), isolating exactly what the negative pairs of (8.1) were buying. Chapter 11 teaches the same representations without labels via self-distillation and the DINO family, whose emergent attention echoes Figure 8.13. Chapter 1 (Section "evaluation protocols") and Chapter 16 (family comparisons, evaluation artifacts) pick up the method-evaluation thread opened in Section 8.8.

CHAPTER 9

Contrastive Learning

In this chapter

9.1 From Pretext Prediction to Pretext Invariance . 414 • The invariance–covariance dilemma • PIRL: pretext-invariant representation learning
9.2 Deep Roots: Siamese Networks and the Agreement Principle ... 417 • A modality beyond vision: audio
9.3 The Loss Ladder: Triplet → N-pair → NCE → InfoNCE .. 422 • Rung 1: Triplet loss and the margin • Rung 2: N-pair loss — multi-negative softmax • Rung 3: NCE as density-ratio estimation • Rung 4: InfoNCE — cross-entropy over negatives • A coda on the fifth rung: clustering as discrete contrasting
9.4 The Temperature τ on the Hypersphere 444
9.5 InfoNCE as a Mutual-Information Lower Bound 446 • Step-by-step derivation of the bound • Batch size and the MI-bound trade-off
9.6 False Negatives and Class Collision 452
9.7 SimCLR: A Simple Framework in Nine Lines ..... 454 • Augmentation composition • The algorithm, walk by walk • The projection head: information is kept in $\boldsymbol{h}$, not $\boldsymbol{z}$
9.8 MoCo: Momentum Contrast and the Dictionary as a Queue 465 • The EMA encoder • The pseudocode listing
9.9 Putting the Pieces Together: SimCLR vs. MoCo 472
9.10 Why Does Contrastive Learning Work? (Interpretation, Not Settled Theory) ..... 473 • Empirical picture: alignment and uniformity • Comparison to the previous paradigm
9.11 Chapter summary 475
9.12 Exercises 477

Chapter roadmap

Before you start. You need three things from earlier chapters. From Chapter 8: the augmentation toolkit (crop, colour jitter, blur) and the invariance-versus-covariance question — this chapter answers it by picking invariance and building a whole training signal around it. From Chapter 8 again: how to read a linear-probe / kNN evaluation table, since every method below is judged that way. From Chapter 2: what a softmax and a cross-entropy loss are, and roughly what mutual information measures — the loss ladder of Section 9.3 turns a scalar distance into exactly these.

The idea in one sentence. Take one image, make two different-looking copies of it, and train a network to pick out — from a whole crowd of unrelated images — which one is the other copy.

What we cover.

- *PIRL* (Section 9.1): the hinge from Chapter 8's pretext tasks into this chapter — keep the transformation, drop the job of predicting it.
- *Where the idea comes from* (Section 9.2): Siamese networks and a 1992 stereogram experiment, because the "two branches, one agreement signal" picture predates deep learning by decades.
- *The loss ladder* (Section 9.3): triplet → N-pair → NCE → InfoNCE, one small fix at a time, until a plain distance becomes a K-way classification problem.
- *Temperature* (Section 9.4): the one number that decides which negatives the loss actually listens to.
- *Why it should work at all* (Section 9.5): the mutual-information bound, and where it quietly runs out of batch size.
- *What can go wrong* (Section 9.6): false negatives — when the loss pushes apart two images that secretly share a class.
- *SimCLR and MoCo* (Sections 9.7 –9.8): the two ways of affording enough negatives — a huge batch, or a queue paired with a slowly-moving encoder.

Afterwards you will be able to (1) name the one failure mode every contrastive loss is designed around, and say in one sentence how each rung of the loss ladder attacks it; (2) predict, for a given temperature, which negatives will dominate a gradient step; (3) explain why SimCLR discards its projection head at transfer time but MoCo keeps a whole extra encoder; and (4) read a batch-size-vs-accuracy plot and connect its shape to the mutual-information bound that predicts it.

Key Concept — Contrastive learning

Contrastive learning trains an encoder so that *positive pairs* — two views of the same underlying sample — land close together in embedding space, while *negative pairs* — views of unrelated samples — are pushed apart. The loss is defined directly on pairs (or tuples) of embeddings; the labels saying *which is which* come from the data pipeline, not from a human.

The previous chapter ended at a fork. Pretext tasks ask the network to *predict* how the input was changed (which rotation, which patch position). That forces the representation to *keep track of* those changes — things most real tasks do not care about.

The other road, drawn on the lecture's slide 44, is simple:

1. take one input $\boldsymbol{x}$;
2. make two augmented views of it;

3. pass both through an encoder;
4. demand that the two outputs *be the same*.

Add one crucial rule — *outputs from different images should be different* — and you have contrastive learning. It powered the first self-supervised systems to match supervised ImageNet pretraining, and its core loss, InfoNCE, is still a workhorse of multimodal representation learning.

This chapter follows the lecture's build-up (Lecture 06, slides 44–73). We first trace the lineage: the Siamese networks of the 1990s and the agreement principle of Becker and Hinton. We then climb the *loss ladder* — triplet loss, N-pair loss, NCE, InfoNCE — deriving each step and showing which weakness of one rung motivated the next. PIRL provides the bridge from pretext tasks into contrastive objectives; SimCLR and MoCo are the two landmark frameworks; and we close with a careful, labeled-as-interpretation discussion of *why* any of this should work at all, plus a failure-modes inventory.

9.1 From Pretext Prediction to Pretext Invariance

9.1.1 The invariance–covariance dilemma

Slide 38 poses the dilemma with a picture of a bird and its 180°-rotated copy. There are two defensible stances:

- **Covariance (pretext prediction).** A good feature *discriminates* the transformation: rotating the image visibly changes the features, so a linear readout can recover the rotation. Chapter 8's RotNet showed this works — the learned filters resemble supervised ones and transfer well. The risk: the same sensitivity makes the feature change under *every* nuisance transformation, and semantics rarely depend on rotation.
- **Invariance.** A good feature *ignores* the transformation : both birds map to (nearly) the same point. Whatever survives invariance to rotations, crops, and color jitter is close to what one would want for classification: a signature of the object itself.

Figure 9.1 draws the two stances as what they do to a feature space. The picture is worth holding onto, because the rest of the chapter is built on the right-hand panel.

Which is right depends on the downstream task. Rotation-sensitive features are useful for orientation estimation; rotation-invariant ones are the natural input to a bird classifier. For the broad class of *semantic* tasks that feature evaluation benchmarks measure — classification, detection, segmentation — the bulk of evidence since 2019 favors invariance. The question is how to *train* for invariance without labels telling you which pairs belong together.

Common Confusion

Invariance is not the same as *destruction* of information. A feature invariant to rotation can retain all the information a downstream task needs — it just cannot tell you the

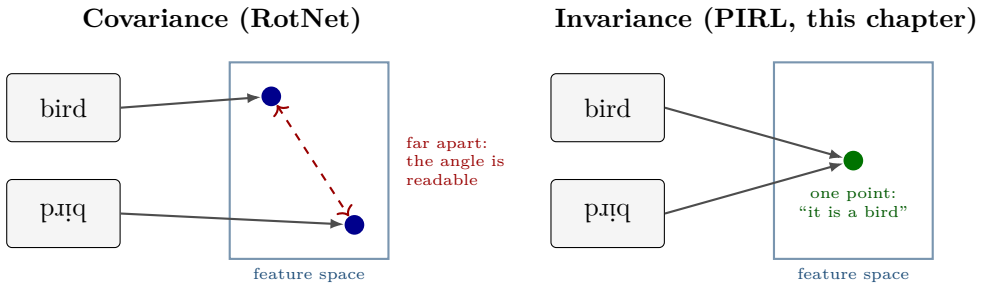


Figure 9.1. The invariance–covariance dilemma as two maps into feature space. *Left:* a covariant feature sends the bird and its rotated copy to different places, so a linear readout can tell you the rotation. *Right:* an invariant feature sends both to the same place, so the only thing left to read is what the object *is*.

How to read it: the two arrows on the right that land on one dot. Contrastive learning is a machine for producing exactly that picture: “these two inputs are the same thing, so they must land in the same place.”

rotation angle. Conversely, invariance is not free: every transformation we declare a “nuisance” is information we throw away, and if some future task *needs* that information, the feature fails there (Section 9.6 returns to this).

9.1.2 PIRL: pretext-invariant representation learning

Slides 39–43 introduce **PIRL** (Pretext-Invariant Representation Learning; Misra and van der Maaten (2020)), which makes the invariance stance concrete by literally combining a pretext task with contrastive alignment. The prior art is standard pretext learning: feed the transformed image I^t (e.g., a jigsaw-shuffled image) to a ConvNet and ask it to predict a property of the transformation t. PIRL asks instead for the representation of I^t to be *similar to the representation of the original image* I — the network sees the jigsaw task's input distribution, but it is not asked to solve the jigsaw; it is asked to be unmoved by it.

Two design problems appear immediately, and PIRL's answers to them are the paper's technical core (slide 41):

1. **Where do negatives come from?** Pushing one positive pair together only makes sense against a background of negatives, but encoding many negatives every step is expensive. PIRL keeps a *memory bank* $\mathcal{M}$: a dictionary of previously computed features, one per dataset image, which acts as a reservoir of negatives. Because the memory bank is only updated occasionally, its features are slightly stale — a weakness MoCo later fixes by Queue-plus-momentum (Section 9.8).
2. **How to keep the jigsaw statistics from corrupting the representation?** PIRL uses *separate heads*: the encoder backbone θ produces features $\boldsymbol{v}_I$ and $\boldsymbol{v}_{I^t}$; two heads f and g then map them into the space where similarity is measured, with g processing the jigsawed input so the backbone never has to ingest the grid artifacts directly.

The loss is NCE (slide 42), which we analyze in full generality in Section 9.3; here it appears as the probability

$$h(\boldsymbol{v}_I, \boldsymbol{v}_{I^t}) = \frac{\exp(s(\boldsymbol{v}_I, \boldsymbol{v}_{I^t})/\tau)}{\exp(s(\boldsymbol{v}_I, \boldsymbol{v}_{I^t})/\tau) + \sum_{I' \in \mathcal{D}_N} \exp(s(\boldsymbol{v}_{I^t}, \boldsymbol{v}_{I'})/\tau)}, \tag{9.1}$$

where $\boldsymbol{v}_I = f(\boldsymbol{v}_I^{\text{enc}})$ is the head-processed feature of image I, $\boldsymbol{v}_{I^t} = g(\boldsymbol{v}_{I^t}^{\text{enc}})$ is the head-processed feature of the transformed image, $s(\cdot, \cdot)$ is cosine similarity, τ is a temperature, and $\mathcal{D}_N \subset \mathcal{M}$ is the set of negatives drawn from the memory bank. Reading (9.1) as the model's belief that the positive pair $(\boldsymbol{v}_I, \boldsymbol{v}_{I^t})$ came from the data distribution rather than from noise, minimizing its negative log-likelihood pulls the positive pair together and pushes all negatives apart.

Paper Analysis: PIRL (Misra & van der Maaten, 2020)

Paper: *Self-Supervised Learning of Pretext-Invariant Representations* (Misra and van der Maaten, 2020).

Problem. Pretext tasks teach representations to co-vary with the transformation. For semantic transfer benchmarks, the literature suggests invariance is preferable — but how to combine the pretext task's label-free structure with an invariance objective?

Prior work. Pretext tasks (jigsaw, rotation, colorization, Chapter 8); instance discrimination with memory banks and NCE (Wu et al.'s NPID lines in PIRL's comparison tables); early examples of pairing a positive view against memory-bank negatives.

Insight. Use the pretext task only to *generate the transformation*; the objective then aligns $\boldsymbol{v}_I$ with $\boldsymbol{v}_{I^t}$ against the memory bank. The jigsaw task itself is largely a source of strong image-shuffling augmentations – its semantics are deliberately *killed* by the separate heads f and g.

Method. Equation (9.1) applied to pairs (I, I^t) with negatives cached in a memory bank $\mathcal{M}$; separate projection heads for the original and jigsawed branches.

Experiments. Linear classifiers trained on frozen PIRL representations of a ResNet-50 pretrained on unlabeled ImageNet, evaluated across four benchmarks (the table shown on slide 43).

Results. PIRL reaches 63.6% top-1 on ImageNet linear probing, 81.1 mAP on VOC07, 49.8% on Places205, and 34.1% on iNaturalist — clearly improving on its covariance-oriented cousin Rotation (48.9 / 63.9 / 41.4 / 23.0) and on the then-recent MoCo (60.6 on ImageNet) in the linear-probe protocol used on the slide.

Limitations. Memory-bank staleness: stored features lag the encoder. Only image-to-image views are explored; the augmentation-invariance menu is limited to jigsaw and crops at this stage.

What to remember. Two ideas outlive PIRL itself: align across transformations rather than predict them, and use external storage (memory bank) to cheaply assemble negatives — both are stepping stones to SimCLR and MoCo.

Method	Parameters	Transfer Dataset			
		ImageNet	VOC07	Places205	iNat.
ResNet-50 using evaluation setup of [19]					
Supervised	25.6M	75.9	87.5	51.5	45.4
Colorization [19]	25.6M	39.6	55.6	37.5	–
Rotation [18]	25.6M	48.9	63.9	41.4	23.0
NPID++ [72]	25.6M	59.0	76.6	46.4	32.4
MoCo [24]	25.6M	60.6	–	–	–
Jigsaw [19]	25.6M	45.7	64.5	41.2	21.3
PIRL (**ours**)	25.6M	**63.6**	**81.1**	**49.8**	**34.1**

Figure 9.2. The transfer table the paperbox above quotes (PIRL paper): linear classifiers on frozen ResNet-50 features across four benchmarks. PIRL (bold) leads its covariance-oriented cousin Rotation on every column and edges past the contemporary MoCo entry on ImageNet, while trailing the fully supervised row — the gap this whole chapter is about closing.

How to read it: the Rotation row versus the PIRL row. Same backbone, same training budget, same evaluation — the only difference is invariance instead of covariance as the training signal, and every column moves in PIRL's favour.

Source: Lecture 06, slide 43 (data: Misra and van der Maaten, 2020).

PIRL is the hinge in the course's narrative. Look again at slide 44: two augmentation branches, two networks, and the same curved double-arrow between the outputs — it is exactly PIRL's picture, stripped of the jigsaw and generalised to any pair of augmentations. The remainder of the lecture—and of this chapter—develops and refines that one picture.

9.2 Deep Roots: Siamese Networks and the Agreement Principle

Historical Note

The two-branch "same vs. different" schema is much older than ImageNet-scale self-supervision. *Becker and Hinton* (1992) (Becker and Hinton, 1992), in their IMAX paper, trained a small self-organizing network on random-dot stereograms — the input is split into two strips A and B — and maximized agreement between the two branches' scalar outputs a and b. Their insight, stated programming-language free, is that a random-dot stereogram is a *clean statistical construction*: both strips encode the same underlying depth, but their random-dot textures are sampled independently, so the noise is independent. Any feature of the local input that a and b could agree on *by exploiting the texture* is a false agreement — short-lived and uninformative. The only stable way for the two branches to agree is to extract the shared latent cause: the depth. Hence maximizing $I(a;b)$,

the mutual information between the two outputs, provably drives the network toward depth-like features with no supervision at all. The principle — two views of one underlying scene, agreement as a learning signal — is exactly the contrastive-positive idea minus the negatives. (Modern contrastive learning re-discovers the IMAX trick each time it treats "two augmentations of one image" as "two views of one scene.")

Bromley et al. (1993) (Bromley et al., 1993) introduced the name "Siamese network" for the architecture: two identical, weight-sharing time-delay CNNs consume two candidate signatures and produce two "signature feature vectors"; a scalar distance measure between them is trained against TARGET labels indicating forgery or authenticity. The key engineering insight is the *shared weights*. Because the two branches are constrained to implement the *same* function f_{θ}, any parameter change from backpropagation is applied to how *both* inputs are processed; distance in feature space can therefore be trusted as a real, semantic notion. The architecture enforces what the loss alone could not.

Chopra, Hadsell, and LeCun (2005) (Chopra et al., 2005) extended the idea to face verification, with a margin-based contrastive loss that directly optimizes intra-class distance and inter-class distance with a margin. FaceNet-style *triplet* losses (next section) are the natural inheritance of this line.

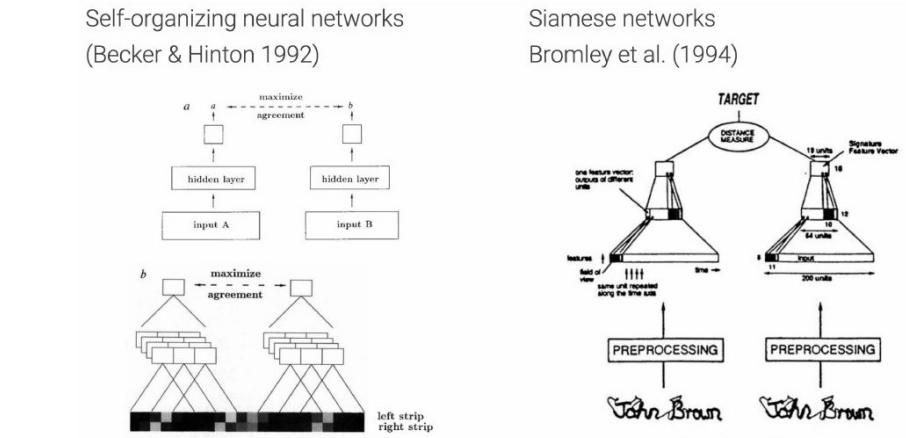


Figure 9.3. The two founding figures side by side, reproduced from the original papers. *Left:* Becker and Hinton's self-organizing network (Becker and Hinton, 1992) — two hidden layers driven by two strips of one random-dot stereogram, trained to maximize agreement between scalar outputs a and b, no labels anywhere. *Right:* Bromley et al.'s Siamese network (Bromley et al., 1993) — two weight-sharing towers turn two signature images into feature vectors, compared by a trained distance unit against a genuine/forged target.

How to read it: compare what sits at the top of each diagram. IMAX's target is a number to maximize *(agreement, unsupervised); the Siamese network's target is a label to* fit *(TARGET, supervised). Everything below that — two branches, shared structure — is already the modern schema.*

Source: Lecture 06, slide 45 (reproducing Becker & Hinton, 1992, and Bromley et al., 1993).

Key Concept — Siamese schema

A *Siamese network* is a two-branch architecture in which both branches run the *same* parameterized function $f_{\boldsymbol{\theta}}$ on two inputs; training compares the two outputs. Positive pairs must agree; negative pairs must be separated by some margin or softmax competition.

Model at a glance: Siamese Network Bromley, Guyon, LeCun, Säckinger & Shah, 1993

In one line	Two copies of the same network, forced to share weights, each looking at one of two inputs; a distance between their outputs is trained against a same/different label.
Problem it solves	An ordinary classifier needs a fixed set of classes decided in advance. Signature verification — and later face verification — needs to compare two arbitrary inputs never seen in training and say "same" or "different."
Key intuition	Do not learn a class boundary; learn a *distance*. If two branches are forced to compute the identical function, any distance between their outputs is a fair, symmetric comparison — neither branch got an unfair head start.
Inputs	Two signals of the same type: two handwritten signatures in the original paper, later two augmented views of one image.
Architecture	One function $f_{\boldsymbol{\theta}}$, applied twice with identical, shared weights — literally the same parameters in memory, not merely the same initialization.
Loss	A distance measure (the original paper trains a small scalar "distance unit") between the two output feature vectors, supervised by a human label: forged or genuine.
Training	Supervised, on labeled pairs. This is the detail the rest of the chapter removes: contrastive self-supervised learning keeps the architecture and the pairwise loss shape, but replaces the human same/different label with one manufactured by the augmentation pipeline.
What you keep	The shared-weight branch, which at inference time is just one encoder producing one embedding per input.
Strengths	Generalizes to inputs never seen in training — the whole point of a verification system — and weight-sharing is a free, architectural regularizer: half as many parameters to overfit with.
Main failure modes	Needs labeled pairs, and says nothing about *how many* negatives to contrast against or how to get enough of them cheaply — exactly the gap the loss ladder (Section 9.3) fills.
Leads to	Every method in this chapter: the two-branch, shared-weight, distance-on-a-pair schema of Figure 9.4 survives untouched from here on; only the loss and the source of pairs change.

In plain words

Build one network. Run it twice — once on each input — using the *exact same weights* both times, like photocopying a machine instead of building two similar ones. If the two inputs are meant to match, push the two outputs together; if not, push them apart. Because both runs used the identical machine, the comparison is fair: any difference

between the two outputs is a difference in the *inputs*, not in which copy of the network happened to process them.

Everyday analogy — twins who trained identically

Imagine two identical twins, raised and trained in exactly the same way, each shown one of two handwriting samples and asked to describe what they see using a fixed vocabulary. Because they were trained identically, if their descriptions differ, the difference must be in the handwriting, not in how the twins were raised. That is weight sharing: not two similar judges, but one judge consulted twice.

Where the analogy breaks. Real twins still have separate brains and could drift apart with different experience after training. A Siamese network's two branches cannot drift: they are the same parameters in memory, updated once per step, so "consulted twice" is literal, not a metaphor for similarity.

Step by step — one Siamese comparison, start to finish

Step 1. Take two inputs. A pair $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ — two signatures in the original paper, or (later in this chapter) two augmented views of one image.

Step 2. Encode both with the same weights. Compute $\boldsymbol{h}_1 = f_{\boldsymbol{\theta}}(\boldsymbol{x}_1)$ and $\boldsymbol{h}_2 = f_{\boldsymbol{\theta}}(\boldsymbol{x}_2)$, using one shared parameter vector $\boldsymbol{\theta}$ for both calls.

Step 3. Measure a distance. Compute a scalar $D(\boldsymbol{h}_1, \boldsymbol{h}_2)$ — Euclidean distance, cosine similarity, or a small learned "distance unit," depending on the paper.

Step 4. Compare against a target. If $\boldsymbol{x}_1, \boldsymbol{x}_2$ are a positive pair, push D down (or agreement up); if negative, push D up (or agreement down), typically with a margin.

Step 5. Backpropagate through both branches at once. Because the weights are shared, the gradient from *both* passes accumulates into the same $\boldsymbol{\theta}$ — one update, informed by both inputs.

What changed — the agreement principle → a trainable verification network

Before (Becker & Hinton's IMAX, 1992). Two branches maximize *mutual information* between two scalar outputs, with no negatives and no labels at all — an elegant proof of principle on random-dot stereograms, but never scaled past small self-organizing networks.

What changed. Bromley et al. turned the two-branch idea into an engineered, supervised tool: real labels (forged/genuine), a trained distance head, and — the detail that outlives everything else in this chapter — *literal weight sharing* as an architectural constraint rather than a statistical goal.

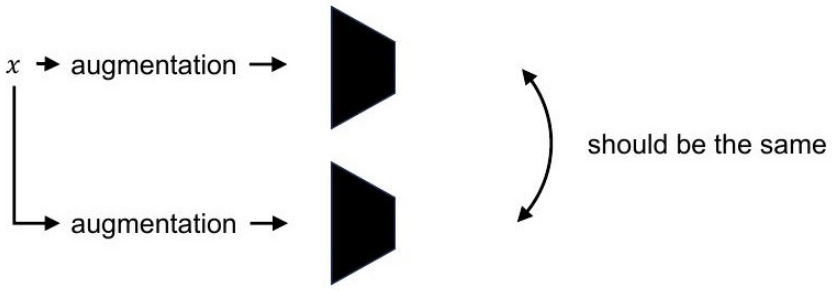


Figure 9.4. The canonical two-branch contrastive schema: two augmentations of the input $\boldsymbol{x}$ feed the (possibly shared) encoder; a similarity-aware loss acts on the pair of embeddings. A modality-general schema: the same picture works for images (two crops), audio (two clips of one recording), or text (two sampled passages).

How to read it: the loss box at the top — everything this chapter adds from here is a different way of computing that one box. The two encoder boxes are tied together (shared weights, Siamese-style) in every method except MoCo (Section 9.8), which deliberately unties them.

Source: Lecture 06, slide 44.

What it cost. Supervision. IMAX needed no labels; Siamese networks need a human-labeled pair for every training example, which is exactly the bottleneck self-supervised contrastive learning spends the rest of this chapter removing.

9.2.1 A modality beyond vision: audio

The lecture makes the point that the schema travels across modalities by showing an *audio* example (slide 46, inspired by Saeed et al.'s COLA). Two clips $\boldsymbol{x}$ and $\boldsymbol{x}'$ cropped from the same minute-long recording are a positive pair; clips from unrelated recordings are negatives. Nothing—the encoder, the loss, the intuition — needs to change; only the augmentation family and the sampling procedure do. The strength of contrastive learning is that "what makes a positive pair" is a property of the *data pipeline*, not of the network.

Research Frontier

Modern contrastive learning *between* modalities — text against image (CLIP), audio against video — is the same schema with $f_{\boldsymbol{\theta}}$ split into two modalities-specific encoders. The pictures look the same as Figure 9.4 because the logic is the same.

Key takeaway

Contrastive learning did not start with deep learning. The recipe is old: two branches, shared structure, a signal that rewards agreement on true pairs and disagreement on false ones. What changes across three decades of research is only *where the pairs come from*

— hand-labeled forgeries in 1993, a jigsaw transformation in PIRL, an augmentation pipeline in everything after it — and *how many negatives the loss can see at once*, which is the subject of the next section.

9.3 The Loss Ladder: Triplet → N-pair → NCE → InfoNCE

Slides 47–52 climb what we call the *loss ladder*: the short sequence, in which each rung fixes one weakness of the previous one. Climb it once, carefully, and the difference between the four losses is almost automatically visible in their formulas.

Why negatives are needed at all. Suppose the only rule were "two views of the same image must get the same embedding". There is a trivial winner: send *every* input to one constant vector $\boldsymbol{c}$. The rule is satisfied perfectly, the embedding carries no information, and gradient descent finds it easily. This is called *collapse*.

So contrastive losses have two parts:

- **alignment** — pull positives together;
- **repulsion** — push negatives apart, which makes collapse impossible.

Which negatives take part, and how many, is exactly what separates one rung of the ladder from the next.

Figure 9.5 shows why both parts are needed, on a toy example you can check by eye: four images, two views each, embedded on a circle (the 2-D version of the unit hypersphere the chapter works on).

New words used from here on

Every loss on the ladder is built from the same handful of words. Here they are in plain terms, before any formula uses them.

Anchor (also called the query). The one example we are currently asking about: "this photo". It is the fixed reference point, the way a ship's anchor is the fixed point the ship is measured from. Everything else is judged by how close it lands to the anchor. In training, every image in the batch takes a turn being the anchor. Triplet papers say "anchor"; InfoNCE papers say "query". It is the same role.

Positive. An example that *should* land close to the anchor. In this chapter it is another augmented view of the same photo.

Negative. An example that should land *far* from the anchor: a view of some other photo.

Distance vs. similarity. Two opposite ways of scoring closeness. A *distance* is small when two embeddings are alike (triplet loss uses this). A *similarity* is large when they are alike (InfoNCE uses cosine similarity). For unit-length embeddings they are tied by

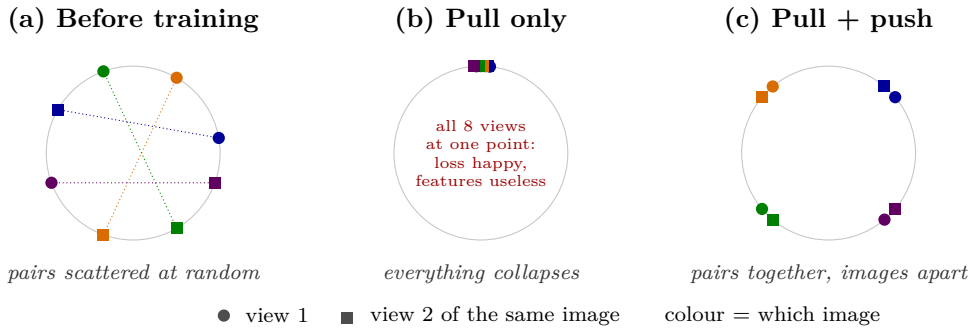


Figure 9.5. Why a contrastive loss needs both halves. Four images (four colours), two augmented views each (dot and square). *(a)* At initialization, the two views of one image land anywhere. *(b)* A loss that only pulls views together is perfectly satisfied by sending all eight points to one spot: every pair agrees, and the embedding says nothing. *(c)* Adding repulsion between *different* images forbids that solution; the only way to satisfy both forces is to keep each image's two views together while spreading the images around the circle.

How to read it: panel (b). It has zero "pull" error, so an alignment-only loss calls it perfect. Every loss on the ladder below is a different recipe for the push that rules (b) out. Section 9.10.1 puts numbers on all three panels.

distance$^2 = 2 - 2 \times$ similarity, so "distance goes down" and "similarity goes up" mean the same thing.

Margin α (alpha). A required safety gap: "the negative must be further away than the positive by *at least* this much". Just a number you choose, like 0.2.

Hinge, $\max(0, x)$. "If x is positive, keep it; if it is negative, replace it with 0." So $\max(0, 3) = 3$ and $\max(0, -2) = 0$. The graph is flat along zero and then bends upward like a door hinge, hence the name. (If you know neural networks, this is exactly the ReLU function.) In a loss it means: *penalize only when there is a problem, and never reward overachieving.*

Logit, softmax, $-\log$, temperature τ. Used from Rung 2 on and explained there, from zero, when first needed.

One lineup to carry up the ladder. To make the four rungs easy to compare, we will keep coming back to the same small scene. The query is a photo of a tabby cat. It is compared with four candidates, and the encoder currently gives these cosine similarities to the query:

candidate	role	similarity s
same photo, re-cropped and colour-jittered	positive	0.85
a *different* tabby cat	negative (secretly same class)	0.75
a puppy	negative	0.30
a cargo ship	negative	−0.10

The ranking is already right (the positive has the highest score), but only barely: the other cat is close behind. Each rung of the ladder will read this same table differently, and Table 9.1 at the end of the ladder puts all four readings side by side.

Simplified intuition vs. more precise view

Simple story. Triplet loss trains a search dog by showing it one target scent, one matching scent, and one distracting scent; it rewards the dog for ignoring the distractor by a safety margin. InfoNCE trains the dog by dropping the matching scent into a room with *many* distractors and asking it to pick the exact match out of the lineup. The second game is strictly harder, and so teaches finer-grained discrimination per update step.

Precise story. The ladder Triplet $\to$ N-pair $\to$ NCE $\to$ InfoNCE is the sequence of conversions of a margin hinge on *pairs* of distances into a softmax cross-entropy over *sets* of candidates. Each conversion step adds negatives to the update, tightens the decision boundary, and moves the loss from constraint-satisfaction language to classification-likelihood language.

9.3.1 Rung 1: Triplet loss and the margin

Before the formula, let us build this loss ourselves, one piece at a time, so that every symbol in it has a reason to be there.

The three players. The loss was invented for face recognition, so use faces. The **anchor** is a photo of Alice at her desk. The **positive** is another photo of Alice, outdoors. The **negative** is a photo of Bob. The encoder turns each photo into a point. Write

$$d_p^2 = (\text{distance from anchor to positive})^2, \qquad d_n^2 = (\text{distance from anchor to negative})^2.$$

What we want, in one sentence: Alice's two photos should be closer together than Alice and Bob are. In symbols, d_p^2 should be smaller than d_n^2. It helps to name the difference:

$$\text{lead} = d_n^2 - d_p^2 \qquad \text{(how much further away Bob is than Alice's other photo).}$$

A big positive lead is good. A negative lead means the encoder thinks Bob looks more like Alice than Alice does: bad.

Now try to turn "we want a big lead" into a loss (a number that training makes smaller). Throughout, keep Alice's photos at $d_p^2 = 0.09$.

Attempt 1: loss $= d_p^2 - d_n^2$ (that is, minus the lead). Smaller loss means bigger lead, so this points the right way. But it never stops. Bob at $d_n^2 = 0.25$ gives loss -0.16; Bob at 2.0 gives -1.91; anywhere further gives a lower score again. There is always a "better" score available by shoving Bob even further away, so training keeps pouring effort into negatives

that are already obviously different, instead of fixing the ones that are actually confused. We need the loss to say "good enough, stop".

Attempt 2: loss $= \max(0,\ d_p^2 - d_n^2)$. Wrap attempt 1 in the hinge $\max(0, \cdot)$: whenever the lead is positive, the inside is negative and gets replaced by 0. Now the loss is zero as soon as Bob is further than Alice's other photo, and positive (a real penalty) only when he is closer. Much better, but two problems remain.

- **No safety gap.** Bob at $d_n^2 = 0.0901$, a hair further than Alice's photo at 0.09, already scores a perfect 0. The slightest change of lighting could flip the order.
- **Collapse wins.** Put *every* photo at the same point. Then $d_p^2 = d_n^2 = 0$, the loss is $\max(0, 0) = 0$, and a useless encoder gets a perfect score (the trap of Figure 9.5(b)).

Attempt 3: loss $= \max(0,\ d_p^2 - d_n^2 + \alpha)$. Demand a lead of *at least* α, not just any lead. Rearranged, the inside is $\alpha -$ lead, so

$$\text{loss} = \max(0,\ \alpha - \text{lead}) = \text{how far short of the required lead we are.}$$

That is the whole idea of the triplet loss in one line: *the loss is the shortfall.* With $\alpha = 0.2$:

Bob at	lead	loss	meaning
$d_n^2 = 0.04$	-0.05	0.25	wrong order: big penalty
$d_n^2 = 0.25$	0.16	0.04	right order, gap too small
$d_n^2 = 0.29$	0.20	0	exactly the required gap: done
$d_n^2 = 2.00$	1.91	0	far more than needed: still 0
everyone collapsed	0	0.2	collapse is now *punished*

Both problems of attempt 2 are gone. A hair's-breadth lead is not enough, and the collapsed encoder now pays α on every triplet. Figure 9.6 draws the three attempts side by side.

Everyday analogy — an entrance exam with a required lead

A scholarship goes to Alice only if she beats the other applicant by at least 20 points. Her "loss" is how many points short of that she fell. Beat him by 5: loss 15. Beat him by 20: loss 0. Beat him by 60: still loss 0, with no bonus for overachieving (that is the hinge). The 20 points is α. Without the "at least 20" rule, a one-point win would be enough, and a tie where both applicants score zero would count as a win too (that is collapse).

Where the analogy breaks. In the exam the scores are fixed and Alice just learns her loss. In training, the loss is used to *move* the three points: the positive is pulled in, the negative is pushed out, until the shortfall is gone.

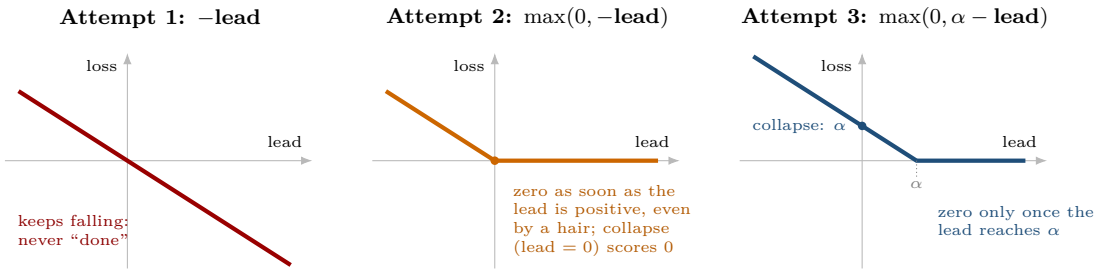


Figure 9.6. Three attempts at a triplet loss, as a function of the lead $d_n^2 - d_p^2$. *Attempt 1* rewards pushing negatives away forever. *Attempt 2* adds the hinge $\max(0, \cdot)$, which flattens the curve at zero once the order is right, but it bends at lead $= 0$, so a tiny lead (or total collapse) already counts as perfect. *Attempt 3* slides the bend to the right by α: now the lead must reach α before the loss lets go, and a collapsed encoder (lead 0) is left paying α.

How to read it: where each curve bends. The hinge decides that *there is a point where the loss stops caring; the margin α decides* where *that point is.*

Now the formula, with every piece named. Attempt 3 is exactly the triplet loss of Schroff et al. (2015). With anchor $\boldsymbol{x}$, positive $\boldsymbol{x}^+$ (same identity) and negative $\boldsymbol{x}^-$, and f the encoder:

$$\mathcal{L}_{\text{triplet}}(\boldsymbol{x}, \boldsymbol{x}^+, \boldsymbol{x}^-) = \max\Big(0,\ \big\|f(\boldsymbol{x}) - f(\boldsymbol{x}^+)\big\|_2^2 - \big\|f(\boldsymbol{x}) - f(\boldsymbol{x}^-)\big\|_2^2 + \alpha\Big), \tag{9.2}$$

Read it piece by piece:

$$\underbrace{\max\Big(0,}_{\substack{\text{hinge: negative}\\ \text{results become 0}}} \quad \underbrace{\big\|f(\boldsymbol{x}) - f(\boldsymbol{x}^+)\big\|_2^2}_{\substack{d_p^2\text{: anchor to}\\ \text{positive}}} - \underbrace{\big\|f(\boldsymbol{x}) - f(\boldsymbol{x}^-)\big\|_2^2}_{\substack{d_n^2\text{: anchor to}\\ \text{negative}}} + \underbrace{\alpha}_{\substack{\text{required}\\ \text{gap}}} \Big)$$

and the whole thing equals the shortfall from the required lead. Here α is the *margin*. Setting the hinge to zero whenever the anchor–positive distance is at least α smaller than the anchor–negative distance is the loss's *geometric content*: it is not asking for positive distance $= 0$, only that positives be closer to anchors than negatives by a safety buffer.

In plain words

Show the network three things at once: a reference photo (the anchor), a second photo of the *same* thing, and a photo of something *different*. Tell it: the same-thing photo should end up closer to the anchor than the different-thing photo, by at least a safety gap. Once that is true, leave this triplet alone and spend effort on a harder one.

Model at a glance: Triplet Loss (used in FaceNet) Schroff, Kalenichenko & Philbin, 2015

In one line	Given an anchor, a matching example, and a non-matching example, push the anchor closer to the match than to the non-match by at least a fixed margin.

Problem it solves	A plain Siamese pairwise loss has no sense of *scale*: "close" and "far" need an absolute threshold, and that one threshold is the same for every pair regardless of how hard the pair is. Triplet loss replaces an absolute threshold with a *relative* ordering.
Key intuition	Never ask for a positive distance of exactly zero. Ask only that positives beat negatives by a safety margin — once true, stop spending gradient on that triplet.
Inputs	A triplet $(\boldsymbol{x}, \boldsymbol{x}^+, \boldsymbol{x}^-)$: anchor, positive (same identity), negative (different identity).
Architecture	The Siamese schema, run three times with shared weights instead of two.
Loss	Equation (9.2): a hinge on the difference of two squared distances, offset by margin α.
Training	The paper's practical core is not the loss itself — it is *which* triplets to show the network. Random triplets are almost always already satisfied; the paper mines semi-hard negatives online, inside each mini-batch.
What you keep	The single encoder f; at inference, embed once and compare by distance.
Strengths	Directly optimizes the distances a verification system actually uses at test time — is this the same face, yes or no.
Main failure modes	One negative per update wastes most triplets on already-satisfied constraints; hard-negative mining is unstable if pushed too far (failure-modes box below).
Leads to	N-pair loss (next), which puts *many* negatives into one update instead of mining one at a time.

Key Concept — Margin

The *margin* α in a triplet loss is a fixed additive gap: training stops improving the embedding as soon as positives are α-closer to anchors than negatives are. Without the margin ($\alpha = 0$), mapping every input to the same point makes both distances zero and the loss exactly zero — collapse would be a perfect solution. The margin forbids that by demanding a real gap. It also turns the problem into a constraint — as in the SVM — rather than an open-ended optimization.

Failure Modes

Triplet learning has two classic failure modes, both discussed by Schroff et al. (2015). (i) *Uninformative triplets.* Random triplets are almost always already satisfied (zero hinge), so most updates do nothing — walls of compute wasted. (ii) *Hard-negative mining instability.* Selecting the hardest negatives (smallest anchor–negative distance) concentrates training on a few pathological examples and on label noise; in metric-learning practice this can cause gradient blow-ups or mode collapse, where all embeddings map to nearly the same point so that all triplets are already violated in the same way and the gradient is zero.

Semi-hard mining — choose negatives that are farther than the positive but only just — is the usual middle ground.

Worked Example — Triplet margin arithmetic

Suppose the encoder embeds the anchor at $f(\boldsymbol{x}) = (0.0, 1.0)$, the positive at $f(\boldsymbol{x}^+) = (0.3, 1.0)$, and the negative at $f(\boldsymbol{x}^-) = (0.5, -1.0)$. Then $\|f(\boldsymbol{x}) - f(\boldsymbol{x}^+)\|_2^2 = 0.3^2 + 0^2 = 0.09$ and $\|f(\boldsymbol{x}) - f(\boldsymbol{x}^-)\|_2^2 = 0.5^2 + 2^2 = 4.25$. With margin $\alpha = 0.2$, the hinge is $\max(0,\ 0.09 - 4.25 + 0.2) = 0$: the triplet is already satisfied. A closer negative at $(1.0, 0.0)$ gives $\|f(\boldsymbol{x}) - f(\boldsymbol{x}^-)\|_2^2 = 1.0^2 + 1.0^2 = 2.0$, so $\max(0,\ 0.09 - 2.0 + 0.2) = 0$ as well: still no gradient. Only a genuinely *semi-hard* negative — farther than the positive but inside the margin, $0.09 < d_n^2 < 0.29$ — would teach anything; at $(0.5, 1.0)$, for example, $d_n^2 = 0.25$ and the hinge is $0.09 - 0.25 + 0.2 = 0.04 > 0$. Figure 9.7 draws these exact points and both zero-loss checks.

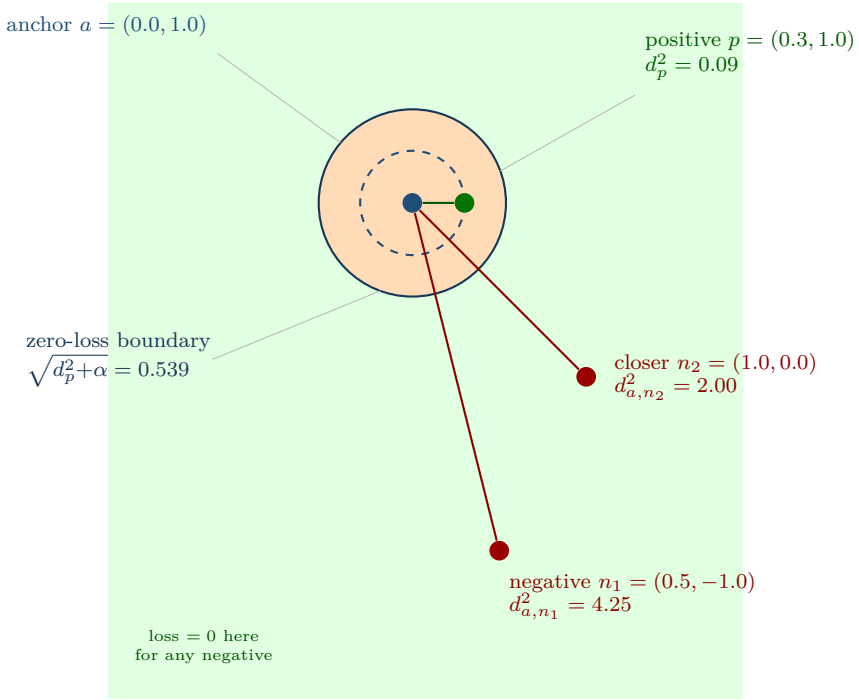


Figure 9.7. The triplet-margin worked example, drawn to scale. The dashed circle sits at the positive's own distance from the anchor ($d_p = \sqrt{0.09} = 0.3$); the solid circle inflates that by the margin, to the zero-loss boundary $\sqrt{d_p^2 + \alpha} = \sqrt{0.29} \approx 0.539$. Both candidate negatives — the "far" one at squared distance 4.25 and the closer one at squared distance 2.00 — land far outside the solid circle, in the shaded region where Equation (9.2)'s hinge is exactly zero.

How to read it: the two circles are not decoration — everything outside the solid one contributes zero gradient, by construction. Both negatives already sit far outside it, which is exactly why the hinge evaluates to 0 *for each of them: the triplet loss stopped caring about this pair before it was ever computed.*

The whole loss in one plot. Hold the anchor and the positive fixed (so $d_p^2 = 0.09$) and slide the negative from right on top of the anchor to far away. The triplet loss is then a function of one number, the negative's squared distance d_n^2:

$$\mathcal{L}_{\text{triplet}} = \max\big(0,\ \underbrace{0.09 + 0.2}_{0.29} - d_n^2\big).$$

That is a ramp that falls in a straight line and then goes flat, a "hockey stick". Figure 9.8 draws it and names the three zones practitioners talk about.

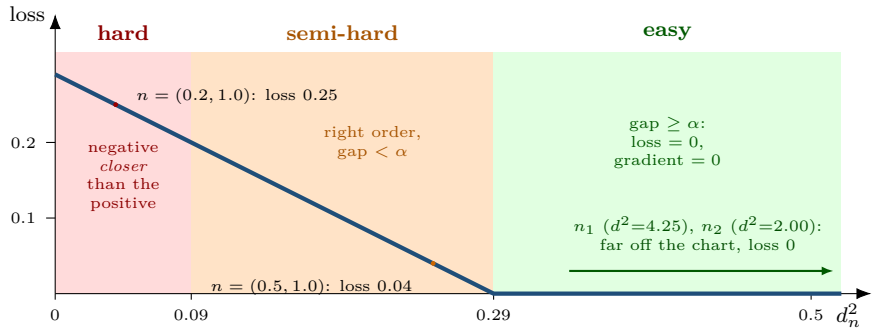


Figure 9.8. The triplet loss as a function of how far the negative is, with the anchor and positive of the worked example held fixed ($d_p^2 = 0.09$, $\alpha = 0.2$). The ramp hits zero at $d_n^2 = d_p^2 + \alpha = 0.29$ and stays there forever. The slope of the ramp is the gradient, so only negatives in the red and orange zones produce any learning signal.

How to read it: how much of the axis is flat. Every negative to the right of 0.29*, which in a real dataset is almost all of them, sits on the flat part. That is the "uninformative triplets" failure mode in one picture, and it is why FaceNet mines semi-hard negatives (orange zone) instead of sampling at random.*

Why not always pick the hardest negative? The red zone looks like the most useful place to be (largest loss, steepest push), but it is also where the mistakes live. A negative that the encoder places *closer* than a true positive is often either mislabeled, or a near-duplicate, or a genuinely ambiguous image. Training mostly on those makes the gradient noisy and can drag the whole embedding towards collapse, as the failure-modes box above describes. The orange zone is the compromise: the order is already right, so the example is probably clean, but the gap is still too small, so there is something left to learn.

Which way does the anchor move? One line of calculus answers it. Write $a = f(\boldsymbol{x})$ (anchor), $p = f(\boldsymbol{x}^+)$ (positive), $n = f(\boldsymbol{x}^-)$ (negative), and suppose the hinge is active (the loss is positive). Differentiating (9.2) with respect to a:

$$\frac{\partial \mathcal{L}_{\text{triplet}}}{\partial a} = \frac{\partial}{\partial a}\Big(\|a - p\|_2^2 - \|a - n\|_2^2\Big) = 2(a - p) - 2(a - n) = 2(n - p).$$

Notice what is missing: a itself. Gradient descent moves the anchor along $-(n-p) = p-n$ — away from the negative, towards the positive — by an amount set only by how far apart p and n are, wherever the anchor happens to be.

The same one-line calculation gives the other two gradients: $\partial\mathcal{L}/\partial p = 2(p-a)$ (the positive is pulled towards the anchor) and $\partial\mathcal{L}/\partial n = 2(a-n)$ (the negative is pushed away from it). In a real network these three gradients flow back into the shared encoder weights, but it helps to watch them act on the points directly.

Worked Example — One gradient step on the semi-hard triplet, watched

Take the semi-hard case from above: $a = (0, 1)$, $p = (0.3, 1)$, $n = (0.5, 1)$, $\alpha = 0.2$. The loss is $0.09 - 0.25 + 0.2 = 0.04$, so the hinge is active. All three points share $y = 1$, so only the x-coordinate moves and we can draw the whole update on a number line (Figure 9.9).

Gradients. $\partial\mathcal{L}/\partial a = 2(n-p) = (0.4,\, 0)$, $\partial\mathcal{L}/\partial p = 2(p-a) = (0.6,\, 0)$, $\partial\mathcal{L}/\partial n = 2(a-n) = (-1.0,\, 0)$.

One step with learning rate 0.1 (each point moves by $-0.1\times$ its gradient): the anchor moves from 0 to -0.04, the positive from 0.3 to 0.24, the negative from 0.5 to 0.6.

Check. New distances: $d_p^2 = 0.28^2 = 0.078$ and $d_n^2 = 0.64^2 = 0.410$. The hinge is now $\max(0,\, 0.078 - 0.410 + 0.2) = 0$. One step was enough: the triplet has left the orange zone of Figure 9.8, and from now on it will contribute nothing.

Notice who did most of the work: the negative, which had the largest gradient (1.0), moved furthest. The anchor moved *away from both* other points, because p and n sit on the same side of it; what matters is only that it moved away from n faster than from p.

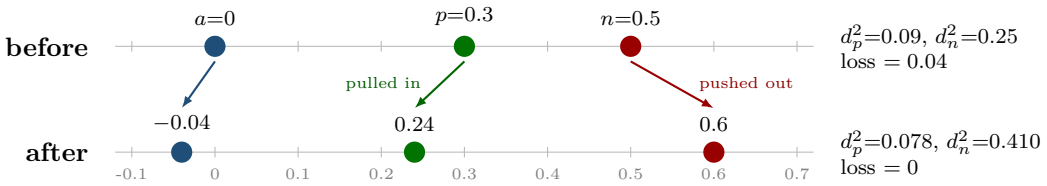


Figure 9.9. The worked gradient step on a number line (all three points have $y = 1$, so only x changes). The positive slides towards the anchor, the negative is pushed away, and the anchor drifts away from the negative. After one step with learning rate 0.1 the gap $d_n^2 - d_p^2$ grows from 0.16 to 0.33, past the margin $\alpha = 0.2$, and the loss is exactly zero.

How to read it: the red arrow. It is the longest of the three because the negative gets the biggest gradient. Once it crosses the margin, all three arrows vanish for this triplet on every future step.

And when the hinge is inactive, the gradient is exactly zero: the triplet teaches nothing. That is the failure the next rung fixes.

Everyday analogy — a rope line, not a bullseye

Triplet loss does not ask the positive to land exactly on top of the anchor, the way a dart has to hit a bullseye. It asks for a *minimum gap*, the way a venue's rope line only needs

the queue to stay a fixed distance back from the door: squeezing closer than that earns nothing extra, and the bouncer (the loss) only intervenes once someone crosses the line.
Where the analogy breaks. A rope line marks one fixed position. The margin α in Equation (9.2) is not a position in space at all — it is a required *gap between two distances*, so the line it draws moves with the anchor and the negative on every single triplet.

Step by step — computing one triplet update

Step 1. **Sample a triplet.** Pick an anchor $\boldsymbol{x}$, a positive $\boldsymbol{x}^+$ of the same identity, and a negative $\boldsymbol{x}^-$ of a different one.

Step 2. **Embed all three.** Run the shared encoder: $a = f(\boldsymbol{x})$, $p = f(\boldsymbol{x}^+)$, $n = f(\boldsymbol{x}^-)$.

Step 3. **Compute two distances.** The anchor–positive distance $\|a - p\|_2^2$ and the anchor–negative distance $\|a - n\|_2^2$.

Step 4. **Subtract, add the margin, clip at zero.** This is the hinge in Equation (9.2): positive only when the positive is not yet closer than the negative by α.

Step 5. **Backpropagate only if the hinge is active.** If the margin is already satisfied, the gradient is exactly zero — this triplet taught the network nothing this step, which is exactly why hard-negative mining exists.

What changed — Siamese pairwise loss → triplet loss

Before. A Siamese network trained on pairs needs an *absolute* distance threshold: push positive pairs below some value, negative pairs above it. That threshold must be hand-tuned, and it is the same for every example, easy and hard alike.
What changed. Compare three embeddings per update instead of two, and ask only for a *relative* ordering: positive closer than negative, by margin α. No absolute threshold is ever chosen.
What it cost. A third forward pass per update (the negative), and a new, non-trivial problem: which negative to sample. Most random triplets are already satisfied and teach the network nothing (the failure-modes box above), so triplet mining becomes its own research problem.

9.3.2 Rung 2: N-pair loss — multi-negative softmax

Sohn (2016) observed two triplet weaknesses: the information content of one update is $O(1)$ in the number of negatives, and the hinge remains piecewise-linear, so gradients saturate at constraint satisfaction.

In plainer words, the two complaints are these. *First*, a triplet looks at only one negative per update, so it learns about one wrong answer at a time. *Second*, the hinge is all-or-nothing: the moment the shortfall reaches zero, the push switches off completely (the flat part of

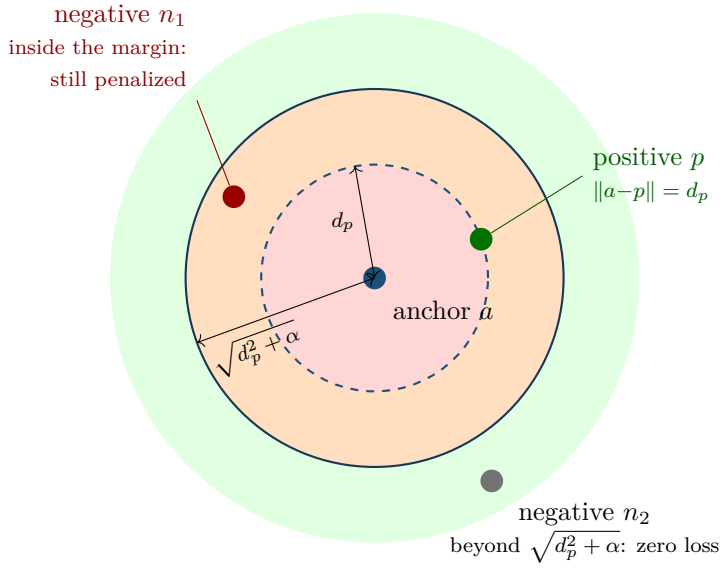


Figure 9.10. Triplet-loss geometry around one anchor. The positive p sits at distance d_p from the anchor (green); the dashed circle marks that radius, the solid circle marks the inflated radius $\sqrt{d_p^2+\alpha}$ (the loss compares *squared* distances). A negative inside the solid circle (red or orange zone) still receives gradient pushing it outward; once a negative crosses into the green zone beyond that radius, the hinge in Equation (9.2) is exactly zero and training ignores it completely.

How to read it: find the two circles around the anchor first. The dashed one is set by the positive's own distance; the solid one adds the margin α on top. Negative n_1 is still inside the solid circle, so it is still being pushed away; negative n_2 is already outside it and gets no gradient at all — the "no penalty once far enough" region this loss is built to create.

Figure 9.6), even for negatives that are only just past the margin. The fix replaces the hinge with a smooth scoring rule built from two tools, the softmax and $-\log$. If those are new, the box below is all you need.

New words used from here on

Scores, not distances. From this rung on, closeness is measured by a *similarity score* s (a dot product or cosine): bigger means more alike. Score the anchor against every candidate: one positive, several negatives.

Softmax: turning scores into percentages. Take each score s, compute e^s (which makes every number positive, and makes bigger scores much bigger), then divide by the total so everything sums to 100%. With scores $(2, 1, 0)$ for (positive, negative 1, negative 2):

$$(e^2, e^1, e^0) = (7.39,\ 2.72,\ 1.00), \quad \text{total } 11.11, \quad \text{softmax} = (66.5\%,\ 24.5\%,\ 9.0\%).$$

Read this as the model's answer to "which candidate is the positive?": 66.5% sure it is the right one, 24.5% fooled by negative 1. The scores that go *into* a softmax are called **logits**.

$-\log P$: turning a percentage into a penalty. Let P be the percentage the softmax gives the *true* positive. The loss is $-\log P$:

P (confidence in the right answer)	1.00	0.90	0.665	0.50	0.10	0.01
loss $-\log P$	0	0.11	0.41	0.69	2.30	4.61

Sure and right costs nothing; unsure costs a little; confidently wrong costs a lot. And unlike the hinge, it is never *exactly* zero unless $P = 1$, so there is always a small push towards more confidence. This pair, softmax followed by $-\log$, is the ordinary *cross-entropy* loss of every image classifier; the only twist here is that the "classes" are the candidates in the lineup.

In plain words

Instead of comparing the anchor to one negative at a time, line up *all* of the negatives at once and ask the network to give the highest score to the positive among the whole line-up — the way a multiple-choice question scores one right answer against every wrong option simultaneously, not one wrong option per question.

Model at a glance: N-pair Loss Sohn, 2016

In one line	Replace the triplet's single negative with all of a batch's negatives at once, scored by one softmax instead of one hinge.
Problem it solves	Triplet loss updates on one negative at a time, so each step carries little information, and the piecewise-linear hinge saturates the moment the margin is satisfied.
Key intuition	Put every negative into the denominator of one softmax. The gradient on the positive logit now depends on *all* negatives simultaneously, not on whichever single one happened to be sampled.
Inputs	One anchor $\boldsymbol{x}$, one positive $\boldsymbol{x}^+$, and $N-1$ negatives $\boldsymbol{x}_1^-, \dots, \boldsymbol{x}_{N-1}^-$ — typically one negative drawn from each of the other classes represented in a batch.
Loss	Equation (9.3): algebraically a softmax cross-entropy over N candidates, derived from a log-one-plus-sum-of-exponentials soft-margin form.
What you keep	The encoder f; no extra components over a plain Siamese network.
Strengths	One update now touches every negative in the batch; gradients no longer saturate the way a satisfied hinge does.
Main failure modes	The softmax has an implicit, fixed temperature of 1, and the number of negatives is still limited by how the batch happens to be structured.
Leads to	NCE and InfoNCE (next), which put the same softmax shape on a principled statistical footing and add an explicit temperature.

The fix is to put *all* of a sample's negatives into one softmax denominator:

$$\begin{aligned} \mathcal{L}_{N\text{-pair}} &= \log\!\left(1+\textstyle\sum_{i=1}^{N-1} e^{\delta_i}\right) = -\log \frac{e^{s^+}}{e^{s^+}+\sum_{i=1}^{N-1} e^{s_i}} \\ \text{with}\quad s^+ &:= f(\boldsymbol{x})^\top f(\boldsymbol{x}^+), \quad s_i := f(\boldsymbol{x})^\top f(\boldsymbol{x}_i^-), \quad \delta_i = s_i - s^+ , \end{aligned} \tag{9.3}$$

where the second equality multiplies top and bottom by e^{-s^+}.

That one line of algebra shows the *same* loss in two costumes:

- **Form 1** — $\log(1+\sum e^{\cdots})$**:** a soft margin, closest to the triplet hinge.
- **Form 2** — **negative log-softmax:** closest to modern cross-entropy.

Form 2 carries the useful reading: N-pair is the *negative log-likelihood of picking the positive out of* $1+(N-1)$ *candidates*, where the softmax scores are inner products. Eq. (9.3) is the first rung that is truly *multi-negative*: all the negatives sit together in the denominator.

Worked Example — N-pair on one anchor with three negatives

Let the anchor score $s^+ = 2.0$ with its positive and $s_1 = 1.5$, $s_2 = 0.0$, $s_3 = -1.0$ with three negatives. Every negative is already *below* the positive, so a triplet loss with a small margin would be satisfied for all three.

Form 1. The gaps are $\delta_i = s_i - s^+ = (-0.5, -2.0, -3.0)$, so $e^{\delta_i} = (0.607, 0.135, 0.050)$, which sum to 0.792. The loss is $\log(1+0.792) = 0.583$: *not* zero, even though the ranking is correct.

Form 2. The softmax gives the positive probability $1/1.792 = 0.558$, and $-\log 0.558 = 0.583$ again. Same number, as promised.

Who gets pushed? Each negative's share of the push is its softmax probability: $0.607/1.792 = 0.34$ for the close negative, 0.08 for the middle one, 0.03 for the far one. The close negative gets about four and a half times the push of the middle one and twelve times the push of the far one. Nobody had to *mine* it: the softmax found it.

What changed — triplet loss → N-pair loss

Before. One negative per update. Most of the gradient budget is spent on triplets that are already satisfied, and the piecewise-linear hinge gives no signal once satisfied.

What changed. All negatives enter one softmax denominator at once. The loss becomes smooth — no more flat, zero-gradient region — and every negative contributes to every update.

What it cost. More compute per update (encode $N-1$ negatives instead of one), and the margin is gone, replaced by an implicit competition with no adjustable safety buffer.

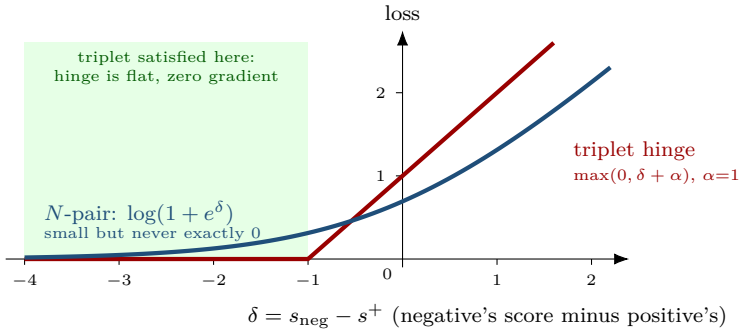


Figure 9.11. The two losses as a function of how close one negative's score is to the positive's (δ). The triplet hinge (red) is a hockey stick again: once the negative is more than α below the positive, the loss and its slope are both exactly zero. The N-pair term (blue) is a smooth "soft" version of the same shape. Far to the left it is tiny but still sloped, so a nearly satisfied negative still gets a small push; near $\delta = 0$ it bends smoothly instead of kinking.

How to read it: the region between $\delta = -2$ and $\delta = -1$. The red curve is flat there; the blue curve is not. That strip is where a triplet stops learning and N-pair keeps going, with a push that fades gradually instead of switching off.

9.3.3 Rung 3: NCE as density-ratio estimation

Noise-contrastive estimation was introduced by Gutmann and Hyvärinen (2010) — independently of representation learning — as a way to fit unnormalized models without computing partition functions. Here is the abstract idea.

In plain words

Forget embeddings for a second. NCE solves a much older problem: you have real data and want to fit a model of it, but computing exactly how "likely" the model says any point is would require an impossible sum over every possible point (the partition function). NCE's trick: do not compute that sum. Instead, train a detective that looks at one example and guesses whether it came from the real data or from a cheap noise distribution you made up yourself. A detective that gets good at this job has, without ever being told to, learned the *ratio* of the two densities — and that ratio is all representation learning actually needs.

The idea of NCE in plain terms. Noise-contrastive estimation turns "learn a density" into "tell real from fake".

- Let $p_d(\cdot)$ be the *data* distribution — where genuine positive pairs come from.
- Let $p_n(\cdot)$ be a *noise* distribution that we choose — here, just random other images from the dataset.
- Draw $K + 1$ candidates: one from p_d, K from p_n.
- Train a classifier $h(\boldsymbol{w})$ to say which candidate is the real one.

A classifier trained this way ends up learning $\log \frac{p_d(\boldsymbol{w})}{p_n(\boldsymbol{w})}$ — *how much more likely under real data than under noise*. It never learns either density by itself.

In PIRL's version (Eq. (9.1)) that log-ratio is simply the similarity score s/τ. The model learns "same image or different image", and never fits a full probability model of pairs. A ratio is all representation learning needs.

Worked Example — The detective learns a ratio, with four kinds of pair

Pretend there are only four kinds of (query, candidate) pair a cat photo can appear in. Real pairs (two views of one image) and noise pairs (the photo next to a random other image) produce them with very different frequencies:

pair kind	p_d (real)	p_n (noise)	ratio p_d/p_n	log ratio
same cat, re-cropped	0.90	0.01	90	4.50
cat + another cat	0.07	0.09	0.78	−0.25
cat + dog	0.02	0.30	0.067	−2.71
cat + ship	0.01	0.60	0.017	−4.09

Now show a detective a 50/50 mix of real and noise pairs and ask "real or noise?". The best any detective can do, on a pair of kind w, is answer "real" with probability

$$P(\text{real} \mid w) = \frac{p_d(w)}{p_d(w) + p_n(w)} = (0.989,\ 0.438,\ 0.063,\ 0.016)$$

for the four rows. Turn those probabilities back into logits, $\log \frac{P}{1-P}$, and you get exactly the last column: 4.50, −0.25, −2.71, −4.09. The detective's confidence score *is* the log ratio. It never had to know that real pairs are 90% re-crops, or that noise pairs are 60% ships; it only learned how much more often each kind shows up in real data than in noise. The second row is the interesting one. "Cat next to another cat" is almost as common in noise as in real data, so the detective is close to a coin flip (0.44). That is the false-negative problem of Section 9.6 seen from NCE's side: a same-class pair from noise looks genuinely similar to a real one.

One difference from classic NCE. In the original NCE, the "noise" you compare against is a simple distribution chosen by hand — a Gaussian, say. Self-supervised learning has nothing like that. Instead, the noise is simply *the other images in the minibatch*.

That still works, because NCE only ever needs *samples* of the noise, never its formula. But it changes what the model learns to tell apart:

- **real pair:** two views of the *same* image;
- **fake pair:** two views of *unrelated* images.

Telling those apart turns out to measure how much the two views have in common — which is exactly what the InfoNCE mutual-information bound, next, puts to use.

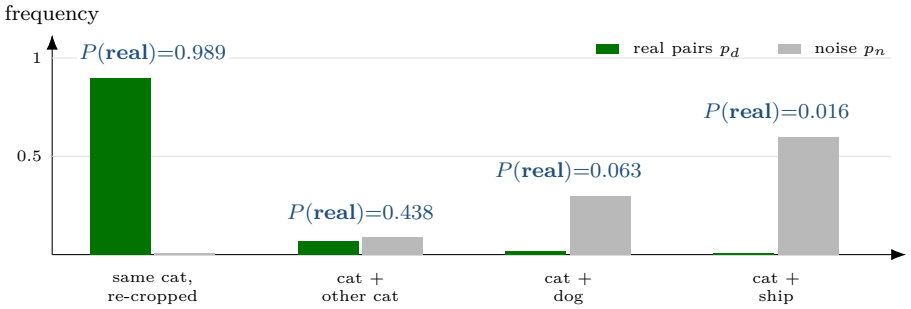


Figure 9.12. The NCE worked example as bars. For each kind of pair, green is how often real data produces it and grey is how often the noise does. The number above each pair is the optimal detective's answer to "real or noise?". It depends only on the *relative* heights of the two bars, not on either height alone.

How to read it: the second pair of bars. They are nearly the same height, so the detective is close to 0.5 *(unsure). That is the one kind of pair NCE cannot confidently call noise, and it is exactly a same-class impostor.*

Simplified intuition vs. more precise view

Simple story. NCE trains "pick the real pair out of the noise."

Precise story. NCE maximizes the likelihood of class labels under a logistic model whose logits are the log-ratio of data-pair density to noise density. In the limit of many samples the model's optimal s satisfies $s(\boldsymbol{v}, \boldsymbol{v}^+)/\tau = \log \frac{p_{\text{data}}(\boldsymbol{v}^+|\boldsymbol{v})}{p_n(\boldsymbol{v}^+)}$ up to a constant: the similarity *is* a (scaled) log-density ratio between the positive-pair conditional and the marginal. This is why h (see Eq. (9.1)) looks like a softmax: it is the softmax of a two-class classifier, generalized into a $(K+1)$-way softmax over candidates.

9.3.4 Rung 4: InfoNCE — cross-entropy over negatives

van den Oord et al. (2018b) repackaged the same idea in the form used by nearly all modern contrastive work. Given a query $\boldsymbol{x}$ (produced by some encoder from one view), one positive $\boldsymbol{x}^+$ and K negatives $\boldsymbol{x}_1^-, \dots, \boldsymbol{x}_K^-$, all encoded to $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\cdot)$, define per-sample similarity scores $s(\boldsymbol{x}, \boldsymbol{x}_j)$ — usually cosine. *InfoNCE* is the negative log-likelihood, under a softmax with temperature τ, of the classifier that picks the positive:

$$\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp\big(s(\boldsymbol{x}, \boldsymbol{x}^+)/\tau\big)}{\exp\big(s(\boldsymbol{x}, \boldsymbol{x}^+)/\tau\big) + \sum_{k=1}^{K} \exp\big(s(\boldsymbol{x}, \boldsymbol{x}_k^-)/\tau\big)}. \tag{9.4}$$

The *query* is simply the anchor under its InfoNCE name. Read the formula piece by piece, using the tools of the Rung 2 box:

$$-\log \frac{\overbrace{\exp\big(s(\boldsymbol{x}, \boldsymbol{x}^+)/\tau\big)}^{\text{the positive's share}}}{\underbrace{\exp\big(s(\boldsymbol{x}, \boldsymbol{x}^+)/\tau\big) + \sum_k \exp\big(s(\boldsymbol{x}, \boldsymbol{x}_k^-)/\tau\big)}_{\text{everyone's total: positive plus all negatives}}}$$

The fraction is the softmax percentage the model gives the positive, and $-\log$ turns it into a penalty. The only new symbol is the **temperature** τ, a number like 0.1 that every score is divided by before the softmax. Dividing by a small number stretches the scores apart, which makes the softmax more decisive (Section 9.4 shows this with a picture). Compared with the triplet loss: no hinge, no α, one positive against *all* negatives at once, and the job of "demand a safety gap" is now done by pushing the percentage towards 100%. Compared with NCE (slide 42 style), nothing structural has changed — it is "the same idea as NCE, but based on cross-entropy" in the lecture's phrasing (slide 52) — but the K-way cross-entropy reading unlocks the information-theoretic interpretation that made InfoNCE famous.

In plain words

Take one query and put it in a line-up with its true match and K impostors. Score every candidate by similarity to the query, turn the scores into probabilities with a softmax, and train the network so the true match gets almost all of the probability. That is the entire loss: pick the right answer out of a crowd.

Model at a glance: InfoNCE (introduced with CPC) van den Oord, Li & Vinyals, 2018

In one line	Score a query against one positive and K negatives with a temperature-scaled softmax, and train with ordinary cross-entropy to pick the positive.
Problem it solves	N-pair fixed the multi-negative softmax but left the temperature implicit and the statistical story informal. InfoNCE gives the same shape an explicit temperature and a clean information-theoretic reading (Section 9.5).
Key intuition	Turn similarity into a classification problem: out of $K+1$ candidates, which one is the true positive? Cross-entropy on that $(K+1)$-way choice is the whole loss.
Inputs	A query $\boldsymbol{x}$, one positive $\boldsymbol{x}^+$, K negatives $\boldsymbol{x}_1^-, \ldots, \boldsymbol{x}_K^-$, all encoded into a shared embedding space.
Loss	Equation (9.4): negative log-softmax of the positive's similarity score among $K+1$ candidates, temperature τ.
What you keep	The encoder producing $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$; the loss itself has no parameters beyond τ, usually fixed rather than learned.

Strengths	The same mathematical shape as NCE, framed as ordinary supervised cross-entropy — every deep learning toolbox already optimizes this loss efficiently — and it lower-bounds a mutual information (Section 9.5).
Main failure modes	Every non-positive candidate is treated as a negative regardless of its true class — the false-negative problem of Section 9.6 — and the bound it optimizes saturates at $\log(K+1)$, capping what a small candidate pool can claim.
Leads to	SimCLR and MoCo (next two sections): two different engineering answers to "where do K negatives come from, cheaply and at scale?"

The picture to keep in your head: a line-up. Slide 51 draws InfoNCE as a game.

- The model is shown a **query** $\boldsymbol{x}$ — a photo of a cat.
- Next to it stands a line-up of $K+1$ candidates: one **positive** $\boldsymbol{x}^+$ (the same cat, rotated) and K **negatives** $\boldsymbol{x}_1^-, \dots, \boldsymbol{x}_K^-$ (a puppy, a cargo ship, ...).
- The model must *pick* the positive out of the line-up.

So InfoNCE is just an ordinary $(K+1)$-way classification loss, where the "right class" is always the positive. Number the candidates $\boldsymbol{z}_0 = \boldsymbol{z}^+, \boldsymbol{z}_1, \dots, \boldsymbol{z}_K$ and give each a score (a *logit*) $\ell_i = s(\boldsymbol{x}, \boldsymbol{x}_i)/\tau$. A softmax turns the scores into probabilities, so the chance of picking the right one is $P(\text{class } 0) = e^{\ell_0} / \sum_{i=0}^{K} e^{\ell_i}$, and the loss is $-\log P(\text{class } 0)$ — which is exactly (9.4).

Two structural observations fall out of that reading. First, the denominator is the source of the repulsion: it contains every candidate that is *not* the positive, so minimizing $-\log P(\text{class } 0)$ simultaneously raises the positive logit and suppresses every negative logit — both pressures come from one term. Second, the retrieval game makes the *false negative* problem visible before any math does: the lineup on slide 51 deliberately includes a *different cat* among the negatives, which the model counts as a distractor even though a human would call it a match. Section 9.6 picks this up.

How hard is each negative pushed? Take the derivative of the loss with respect to one negative's score. Write $D = \sum_{i=0}^{K} e^{\ell_i}$ for the denominator of (9.4), so the loss is $\mathcal{L}_{\text{InfoNCE}} = -\ell_0 + \log D$. Then

$$\frac{\partial \mathcal{L}_{\text{InfoNCE}}}{\partial \ell_j} = \frac{1}{D} \cdot e^{\ell_j} = P(j), \qquad j \geq 1,$$

where $P(j)$ is the probability the model currently gives to negative j.

In words: *each negative is pushed away exactly as hard as the model currently believes in it.*

- A **hard** negative (easily confused with the positive) gets a big push.
- An **easy** negative (already far away) gets almost none.

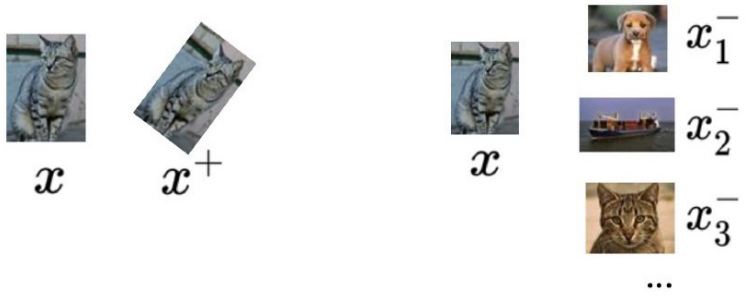


Figure 9.13. The retrieval game, drawn as the lecture draws it. *Left:* a query x and its positive x^+ — the same cat, rotated. *Right:* the same query against a lineup of negatives x_1^-, x_2^-, x_3^- — a puppy, a cargo ship, and (deliberately) a *different* cat. InfoNCE scores every candidate and trains the softmax to favor the true positive.

How to read it: the third negative, x_3^-. It is not the query's cat, but it is still a cat — a same-class distractor the loss has no way to recognize as "basically correct." Section 9.6 follows this exact image through the rest of its consequences.

Source: Lecture 06, slide 51.

This automatic weighting is called *hardness awareness*, and it is the thing the temperature τ controls.

Where does the query itself move? The derivative above is about scores. To see what happens in embedding space, take the common case where the score is a dot product of unit vectors, $\ell_i = \boldsymbol{q}^\top \boldsymbol{k}_i / \tau$ (query $\boldsymbol{q}$, candidates $\boldsymbol{k}_0 = \boldsymbol{k}^+, \boldsymbol{k}_1, \ldots, \boldsymbol{k}_K$). The chain rule gives

$$-\frac{\partial \mathcal{L}_{\text{InfoNCE}}}{\partial \boldsymbol{q}} = \frac{1}{\tau}\Big(\underbrace{\boldsymbol{k}^+}_{\text{the right answer}} - \underbrace{\textstyle\sum_{i=0}^{K} P(i)\, \boldsymbol{k}_i}_{\text{the model's current guess}} \Big).$$

This is the most useful way to picture InfoNCE. The weighted average $\sum_i P(i)\boldsymbol{k}_i$ is "where the model currently thinks the answer is": a blend of all candidates, weighted by how much it believes in each. Gradient descent moves the query along the arrow *from that guess to the true answer*. If the model is already sure ($P(0) \approx 1$), the guess sits on $\boldsymbol{k}^+$ and the arrow has length zero. If it is torn between the positive and a hard negative, the guess sits between them, and the arrow points away from the negative. (We ignore the small correction from re-normalizing $\boldsymbol{q}$ to unit length after the step; it only removes the part of the arrow that points straight out of the sphere.)

Common Confusion

A triplet has one negative per update step; N-pair and InfoNCE have $N-1$ or K. *Notation.* This chapter reserves N for *batch size* (Section 9.7) and uses K uniformly for the number of *negatives*, so InfoNCE's denominator is the $(K+1)$-way softmax over candidates and mutual-information ceilings read $\log(K+1)$; the CPC paper's N

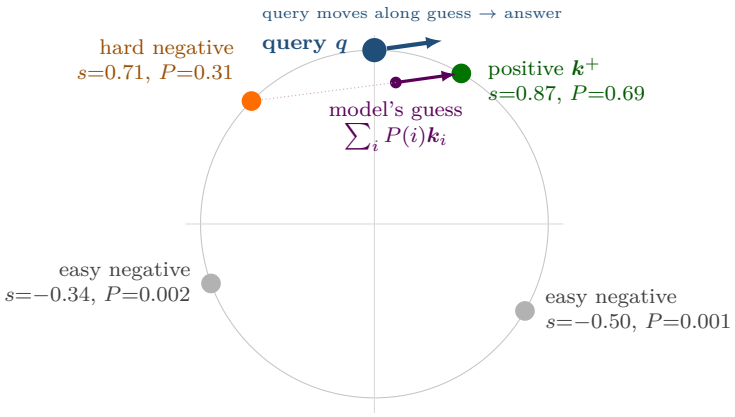


Figure 9.14. The InfoNCE gradient on the query, drawn on the unit circle at $\tau = 0.2$. The query (top) has one positive and one hard negative on either side of it, plus two easy negatives far away. The softmax gives the positive 69% and the hard negative 31%, so the model's current "guess" (purple dot, the probability-weighted average of all candidates) sits between those two. The query is moved along the purple arrow, from the guess towards the true positive: towards $\boldsymbol{k}^+$ and away from the hard negative. The two easy negatives have almost no weight, so they barely affect the direction.

How to read it: the purple dot. Everything InfoNCE does to the query is "move from where you currently think the answer is to where it really is". The easy negatives at the bottom never show up in that arrow, which is hardness awareness seen geometrically.

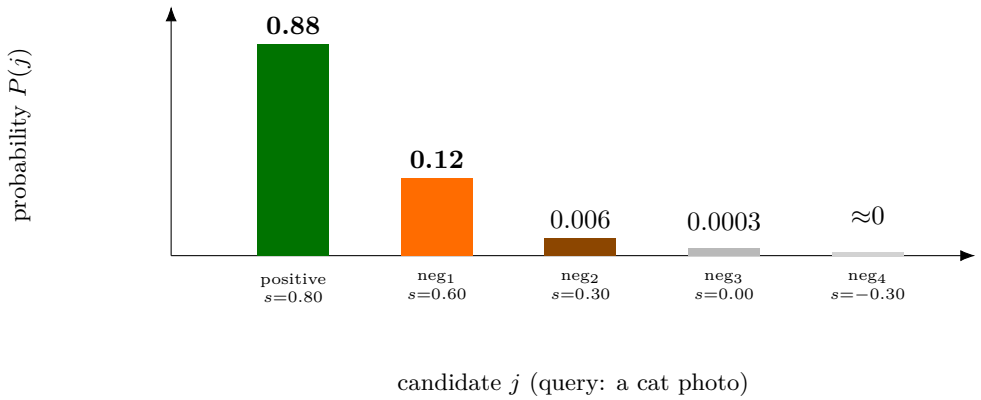


Figure 9.15. InfoNCE at temperature $\tau = 0.1$, drawn as a classification softmax over one positive and four negatives (raw similarities s printed under each bar). Bar heights use a square-root scale so the smallest probabilities stay visible; the printed numbers above each bar are the true values. The positive (green) captures 88% of the probability mass; the closest negative, neg$_1$, still takes 12%; everything past it is negligible. The InfoNCE loss for this query is simply $-\log(0.88) \approx 0.13$ — read directly off the height of the green bar.

How to read it: the gap between the green bar and neg$_1$'s bar. That gap — not the absolute similarity values — is what the loss actually penalizes. Section 9.4 shows how τ stretches or shrinks this same gap.

counts candidates instead. Confusingly, SimCLR uses batch size N but produces $2N$ representations and $2(N-1)$ negatives per loss term, while MoCo uses a queue of size K with K negatives — the notation wars obscure the invariant point: *the denominator should contain as many plausible negatives as compute allows.*

The running lineup, read by all four rungs. Go back to the cat lineup from the start of this section (positive 0.85, other cat 0.75, puppy 0.30, ship −0.10). For the triplet row we turn similarities into squared distances between unit vectors, $d^2 = 2 - 2s$, giving 0.30 for the positive and 0.50, 1.40, 2.20 for the three negatives. For N-pair we plug in the same cosines as its inner products, so that all rows score the same geometry. Table 9.1 shows what each loss makes of the scene.

loss	what it sees	loss	other cat's share of the push
Triplet, $\alpha = 0.5$	one sampled negative	0.30 (cat drawn) 0 (puppy or ship)	all or nothing
N-pair ($\tau = 1$)	all three	1.05	48%
InfoNCE, $\tau = 0.5$	all three	0.83	63%
InfoNCE, $\tau = 0.1$	all three	0.32	99%

Table 9.1. One scene, four losses. The query is a cat photo; the negatives are another cat, a puppy and a ship. The last column is the other cat's fraction of all the push that goes to negatives, $P(\text{cat})/\sum_{j\geq 1} P(j)$.

Three things are visible in that one table. The triplet loss is a lottery: two out of three random draws give zero loss, and the scene teaches nothing. The softmax losses always see the whole lineup, and they all agree that the problem is the other cat. The temperature decides how *exclusively* they focus on it: N-pair's implicit $\tau = 1$ spreads the push fairly evenly (the puppy and the ship still get about half of it), while $\tau = 0.1$ spends 99% of the push on the single hardest negative. That last number returns in Section 9.6, where the other cat turns out to be a false negative.

Everyday analogy — a police lineup with a graded confession

NCE trains a detective to answer one yes/no question: real or fake? InfoNCE puts the same detective in front of a full lineup and asks for a confidence split across every suspect at once — not "is it him," but "how sure are you it's him, versus each of the others." The softmax is that confidence split, and training pushes the split toward total certainty on the true suspect.

Where the analogy breaks. A real detective's confidence does not have to sum to something fixed. The softmax's probabilities are constrained to sum to exactly 1 by construction — raising confidence in the positive mechanically *steals* probability mass from every negative at once, even ones the model never separately reconsidered.

	Sampling	Negative coupling	Form	Role of τ
Triplet	(a,p,n) triplets	Hinge	margin on distances	none
N-pair	(a,p,$N-1$ n)	Softmax	NLL of picking positive	implicit 1
NCE	(pair, K noise)	Two-class	density-ratio logistic	in log-ratio
InfoNCE	(q, $K+1$ cands)	$K+1$-way softmax	cross-entropy	sharpens softmax
Cluster	Centroid + members	Pseudo-label CE	own centroid pos., others neg.	on logits

Table 9.2. The loss ladder: each rung broadens the negative set and clarifies the objective, ending in InfoNCE's softmax-over-candidates form, plus the clustering coda of Section 9.3.5. τ shifts effective logit scale; only the softmax losses have an explicit temperature.

What changed — N-pair / NCE → InfoNCE (CPC)

Before. N-pair already put every negative into one softmax, but with an implicit, fixed scale and no statistical story for what the score means. NCE, developed independently, explained *why* a same-vs-noise classifier recovers a density ratio, but framed the problem as binary (real or noise), not as a K-way pick.

What changed. InfoNCE repackages NCE's density-ratio logic as ordinary $(K+1)$-way cross-entropy, with an explicit temperature τ controlling how sharply the softmax favors the current best guess.

What it cost. Nothing computationally — it is still one softmax, one cross-entropy. What it bought was conceptual: the same loss now has a provable connection to mutual information (Section 9.5), which is what turned a good empirical trick into a principle other papers could build on.

9.3.5 A coda on the fifth rung: clustering as discrete contrasting

Rung five: clustering (DeepCluster). Slide 53 ends the ladder with a loss from a different family: *DeepCluster* (Caron et al., 2018). It repeats two steps:

1. **Cluster.** Run k-means on the current features of the whole dataset, giving K groups. Each image's group number becomes its *pseudo-label* (a label the model made up for itself).
2. **Classify.** Train the backbone with ordinary cross-entropy to predict each image's pseudo-label.

Seen through the ladder, this is contrastive learning with coarser targets: each feature is pulled towards its own cluster centre (the positive) and pushed away from all the other centres (the negatives). The negatives are no longer individual images, as in N-pair — they are *cluster centres*, computed from the whole dataset.

Conceptually this closes a loop with Chapter 2's alternating-minimization view of k-means. It also warns about a failure pattern the pairwise losses lack: if the network collapses every input to nearly the same embedding, k-means happily puts everyone in one cluster and the classification task becomes trivial. DeepCluster blocks this with *cluster re-balancing* —

re-sampling images so that pseudo-label classes stay close to uniform — which is the first appearance in this book of the equipartition devices that will return, under other names, in Chapters 10 and 11.

Key takeaway

Every rung of the ladder answers the same question a little better: *how many negatives can one update see, and how cheaply?* Triplet: one, hand-mined. N-pair: all of a small batch, in one softmax. NCE: reframes the softmax as a real-vs-noise classifier with a provable density-ratio interpretation. InfoNCE: the same classifier, written as $(K+1)$-way cross-entropy with an explicit temperature — the form every method in the rest of this chapter uses. DeepCluster is the odd one out: it swaps individual negatives for cluster centroids, trading precision for the ability to contrast against the whole dataset's structure at once. None of these losses is free by itself; Table 9.2 is the receipt.

9.4 The Temperature τ on the Hypersphere

Because representations in contrastive learning are usually ℓ_2-normalized before the cosine similarity is computed, embeddings live on the unit hypersphere $\mathbb{S}^{d-1}$, and the InfoNCE logits are $\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau \in [-1/\tau, 1/\tau]$. The parameter τ therefore sets the *effective contrast scale* on the sphere. Small τ sharpens (any difference in similarity becomes a large difference in logit); large τ flattens (logits concentrate near zero and the softmax is close to uniform over candidates).

A slightly more formal way to say the same thing reuses the derivative computed at the end of Section 9.3: the gradient of the loss with respect to the logit of any negative $\boldsymbol{x}_j^-$ is $P(j) = \exp(s_j/\tau)/\sum_i \exp(s_i/\tau)$, the model's own softmax confidence in that negative. All τ does is change the *shape* of those coefficients before they hit the negative:

- $\tau \to \infty$: all $P(j) \approx 1/(K+1)$ — updates are spread evenly over all negatives, so easy negatives consume gradient as often as hard ones. Training is stable but slow: the softmax is nearly uniform, and the cross-entropy's gradient magnitude per step is small.
- Small but nonzero τ (e.g. 0.07): the softmax over similarities sharpens into a "top-few" distribution. The few hardest negatives — those with the highest s_j — absorb most of the gradient; the easy ones are effectively skipped without any explicit mining heuristic.
- $\tau \to 0$: the softmax degenerates into an $\arg\max$. Gradient variance spikes because a tiny change in scores can swing which negative currently is the hardest; and the focused fire lands disproportionately on *false negatives* — semantically-same-class samples the batch incorrectly classifies as negatives (slide 51's other cat). Section 9.6 quantifies the latter cost.

This is why the number τ deserves a subsection rather than a footnote: it is not a numerical stabilizer, it is the one knob through which the loss decides *which mistakes to care about*.

The picture: τ is a magnifying glass on the gaps. Cosine similarities are stuck between -1 and 1, so the gap between a positive and a hard negative is always a small number, like 0.4. Dividing by τ stretches every gap by the factor $1/\tau$ before the softmax sees it, and the softmax turns a gap g into a probability ratio e^g. Figure 9.16 shows the same three similarities stretched three ways. The positive-to-hard-negative ratio goes from 1.5× (no stretch) to 2.2× ($\tau = 0.5$) to 55× ($\tau = 0.1$), with nothing about the embeddings changing at all.

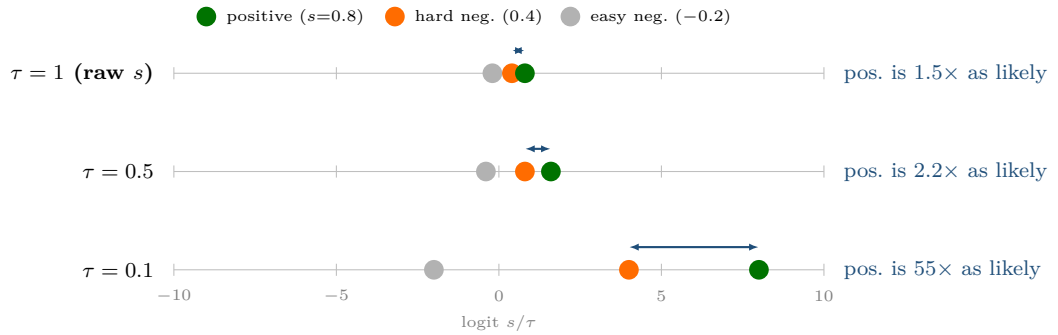


Figure 9.16. Temperature as a stretch. The same three similarities, divided by three temperatures and drawn on one logit axis. The blue arrow is the gap between the positive and the hard negative. At $\tau = 1$ it is 0.4, and the softmax barely prefers the positive ($e^{0.4} \approx 1.5$). At $\tau = 0.1$ the same gap has become 4, and the preference is $e^4 \approx 55$.

How to read it: the blue arrow growing from row to row. Nothing about the encoder changed. Only the ruler did, and with it how loudly the loss complains about the hard negative.

Worked Example — Three pairs at $\tau = 0.1$ vs. $\tau = 0.5$

Suppose a query has, after the encoder and ℓ_2 normalization, cosine similarity 0.8 with the positive and 0.4 and -0.2 with two negatives. The unnormalized InfoNCE logits are $s_+/\tau, s_{-,1}/\tau, s_{-,2}/\tau$.

At $\tau = 0.1$: logits $(8, 4, -2)$. Exponentiating gives $(e^8, e^4, e^{-2}) = (2981.0, 54.6, 0.135)$; summing $= 3035.7$. Softmax over candidates: positive probability $\approx 2981.0/3035.7 = 0.982$. The model is already confident, and the per-example InfoNCE loss is $-\log 0.982 \approx 0.018$.

At $\tau = 0.5$: logits $(1.6, 0.8, -0.4)$. Exponentiating gives $(4.95, 2.23, 0.67)$; sum $= 7.85$. Positive probability $\approx 4.95/7.85 = 0.63$. The loss jumps to $-\log 0.63 \approx 0.46$ — roughly *25 times larger* (0.018 nats versus 0.46 nats), with the *same* features. Intuition: at small τ small differences in similarity become large differences in logit, so the push-apart pressure focuses almost entirely on the *single hardest* negative. At large τ the loss spreads its gradient over many negatives — useful early in training, but loses resolution later. SimCLR's reported best values sit around $\tau \approx 0.1$; MoCo uses 0.07.

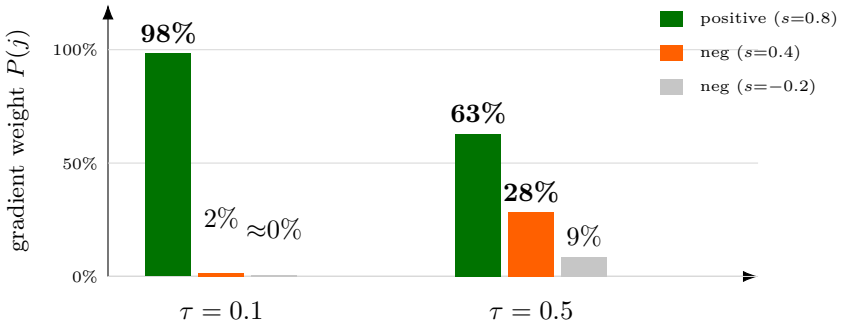


Figure 9.17. Same three candidates as the worked example above (similarities 0.8, 0.4, −0.2), shown at two temperatures. Bar height is each candidate's share of the softmax — exactly $P(j)$, the gradient weight from $\partial \mathcal{L}_{\text{InfoNCE}}/\partial \ell_j = P(j)$ (Section 9.3). At $\tau = 0.1$ the positive absorbs 98% of the mass and both negatives are pushed with almost no force. At $\tau = 0.5$ the positive still wins, but the nearer negative now carries 28% of the gradient and even the far negative gets 9% — temperature is a dial on *how many* negatives receive a meaningful push, not only on the loss value.

How to read it: compare the orange bar's height across the two groups. That is the number that moves the most: the harder negative goes from receiving almost no gradient to receiving more than a quarter of it, purely by raising τ from 0.1 to 0.5.

Key takeaway

Temperature is not a numerical-stability knob; it decides *which mistakes the loss cares about*. Small τ sharpens the softmax until only the single hardest negative gets any real gradient — efficient, but volatile, and it aims its sharpest fire at false negatives (Section 9.6). Large τ spreads the gradient across many negatives — gentler, slower to discriminate late in training. There is no free lunch here, only a trade a practitioner sets once and rarely revisits: SimCLR settles near $\tau \approx 0.1$, MoCo near $\tau = 0.07$.

9.5 InfoNCE as a Mutual-Information Lower Bound

The celebrated "why should this work" answer for InfoNCE passes through mutual information. Recall (notation table): mutual information between two random variables is $I(X; Z)$. Let the two views be random variables $\boldsymbol{x}$ and $\boldsymbol{x}^+$ (the underlying image is dropped from the notation), and let K denote the number of negative samples — one positive plus K negatives gives $K + 1$ candidates. The CPC paper instead uses N for the number of *candidates*; this chapter reserves N for batch size (Section 9.7) and always uses K for negatives, so the paper's $\log N$ is this chapter's $\log(K + 1)$.

Paper Analysis: Contrastive Predictive Coding (van den Oord et al., 2018)

Paper: *Representation Learning with Contrastive Predictive Coding* (van den Oord et al., 2018b).

Problem. Learn representations that preserve "what matters for the future" from raw sequential or structured data — without density modeling, which spends model capacity on low-level statistics (noise, pixels).

Prior work. Autoregressive generative models (PixelRNN); predictive coding in neuroscience; NCE for unnormalized models (Gutmann and Hyvärinen, 2010).

Insight. Predict a future or paired *representation* rather than the raw signal, via a contrastive (softmax/NCE) objective. The objective lower-bounds mutual information between the two views.

Method. An autoregressive context encoder summarises history into a context $\boldsymbol{c}$; a future embedding $\boldsymbol{z}_{t+k}$ is picked out of $N-1$ distractors (the paper's N counts candidates) in a softmax scored by $\exp(\boldsymbol{z}_{t+k}^{\top} W_k \boldsymbol{c})$ — an early, non-cosine form of InfoNCE.

Experiments. Audio (speaker identity, phonemes), images (patches as sequence), NLP, and reinforcement-learning-style auxiliary losses.

What to remember. InfoNCE is introduced here, and the mutual-information lower bound is the paper's conceptual contribution. The paper uses N for the number of *candidates* (this book's $K+1$), so its statement reads $I \geq \log N - \mathcal{L}_{\text{InfoNCE}}$; in this book's convention, with K negatives, that is $I \geq \log(K+1) - \mathcal{L}_{\text{InfoNCE}}$.

In plain words

Here is the claim in one breath, before the three-step proof: if a model gets very good at picking the true positive out of a lineup of K decoys, that is only possible because the two views share a lot of information — you cannot reliably spot the real match among many convincing fakes unless the query and its positive really do have something informative in common. Driving the InfoNCE loss down is therefore indirect evidence of driving mutual information up. The catch, made precise below, is that "how good you can possibly get" is itself capped by how many decoys K you used, so a small lineup can never prove a large amount of shared information, no matter how well the model performs.

9.5.1 Step-by-step derivation of the bound

Write $p_+(\boldsymbol{z}^+) = p(\boldsymbol{z}^+ \mid \boldsymbol{x})$ for the conditional distribution of the positive view embedding given the query, and $p_-(\cdot)$ for the marginal embedding distribution of other images, our noise distribution. Draw one positive $\boldsymbol{z}^+ \sim p_+$ and K negatives $\boldsymbol{z}_k^- \sim p_-$, mix them uniformly to get $K+1$ candidates, and train the model to pick the positive via InfoNCE (9.4); averaging over which of the $K+1$ positions holds the positive leaves a residual $\log(K+1)$ guessing floor in the minimal loss.

Step 1 — Score as log-density ratio. Absorb the temperature into the score by defining $r(\boldsymbol{x}, \boldsymbol{z}) \coloneqq e^{s(\boldsymbol{x},\boldsymbol{z})/\tau}$. The InfoNCE denominator normalizes r over the candidate set, so define $f(\boldsymbol{x}, \boldsymbol{z}) \coloneqq \mathbb{E}_{p_-}[r(\boldsymbol{x}, \boldsymbol{z}')]$: the mean score under the noise distribution. The population-score ratio between picking the true positive and picking one average noise sample is

$$\frac{r(\boldsymbol{x}, \boldsymbol{z}^+)}{f(\boldsymbol{x}, \boldsymbol{z}^+)}.$$

When the critic has converged, a Bayes-optimal analysis of the "which candidate is the positive" discrimination task gives $r(\boldsymbol{x}, \boldsymbol{z}) \propto p_+(\boldsymbol{z})/p_-(\boldsymbol{z})$, hence the ratio above equals, up to a constant, the likelihood ratio of a positive under p_+ versus p_-.

Step 2 — Plug in and split. Setting the optimal critic into (9.4) and expanding the logarithm of the ratio of sums gives the expected per-example loss

$$\mathbb{E}\big[\mathcal{L}^*_{\text{InfoNCE}}\big] \approx \mathbb{E}_{\boldsymbol{x},\boldsymbol{z}^+}\Big[\log(K+1) + \log\frac{p_-(\boldsymbol{z}^+)}{p_+(\boldsymbol{z}^+)} + \log\frac{f(\boldsymbol{x}, \boldsymbol{z}^+)}{r(\boldsymbol{x}, \boldsymbol{z}^+)/\frac{p_+(\boldsymbol{z}^+)}{p_-(\boldsymbol{z}^+)}}\Big], \tag{9.5}$$

where the last term vanishes at the optimum because the optimal r exactly matches the density ratio and f is its expectation under p_-. (The "$\approx$" replaces a sum over the K negatives by K times its average, which is accurate for large K; the exact statement is the inequality below.) So the minimum loss is roughly the chance-level loss $\log(K+1)$ — what a model that guesses uniformly would score — *minus* the expected log-likelihood ratio.

Step 3 — The MI connection. The remaining expectation is

$$\mathbb{E}_{\boldsymbol{x},\boldsymbol{z}^+}\Big[\log\frac{p_-(\boldsymbol{z}^+)}{p_+(\boldsymbol{z}^+)}\Big] = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{z}^+}\Big[\log\frac{p_+(\boldsymbol{z}^+)}{p_-(\boldsymbol{z}^+)}\Big] = -I(\boldsymbol{x}; \boldsymbol{z}^+),$$

the negative mutual information between query and positive view. Re-arranging, the achieved InfoNCE loss is bounded below by $\log(K+1) - I(\boldsymbol{x}; \boldsymbol{z}^+)$, or equivalently

$$\boxed{I(\boldsymbol{x}; \boldsymbol{z}^+) \;\geq\; \log(K+1) \;-\; \mathcal{L}_{\text{InfoNCE}}.} \tag{9.6}$$

(Details of the optimal-critic claim and a catalogue of related bounds are in Poole et al. (2019).) Here K is the number of negatives (so the candidate pool has $K+1$ positions; the slide's $\log N$ uses N for the whole pool size). Since $\mathcal{L}_{\text{InfoNCE}}$ is minimized during training, the right-hand side grows — InfoNCE training is equivalent to maximizing a *lower bound* on the mutual information between the two views.

Worked Example — Reading the bound with a lineup of eight

Take $K = 7$ negatives, so the lineup has $K + 1 = 8$ candidates.
A model that knows nothing spreads its probability evenly, $P(\text{positive}) = 1/8$, so its loss is $-\log(1/8) = \log 8 = 2.08$ nats. The bound gives $I \geq 2.08 - 2.08 = 0$: no evidence of shared information. Correct, since the model is guessing.

A decent model puts $P(\text{positive}) = 0.6$ on the right answer: loss $-\log 0.6 = 0.51$, so $I \geq 2.08 - 0.51 = 1.57$ nats.
A perfect model gets $P(\text{positive}) = 1$: loss 0, so $I \geq 2.08$ nats, and that is the most it can ever certify. In bits, $\log_2 8 = 3$. This matches the intuition of a guessing game: picking one card correctly out of eight can prove you knew at most 3 bits (three yes/no questions' worth) about which card it was, even if you actually knew much more.

That last point is the ceiling, and it has a clean shape: every *doubling* of the lineup adds exactly $\log 2 \approx 0.69$ nats (one bit) of headroom. Figure 9.18 draws it against a hypothetical true mutual information.

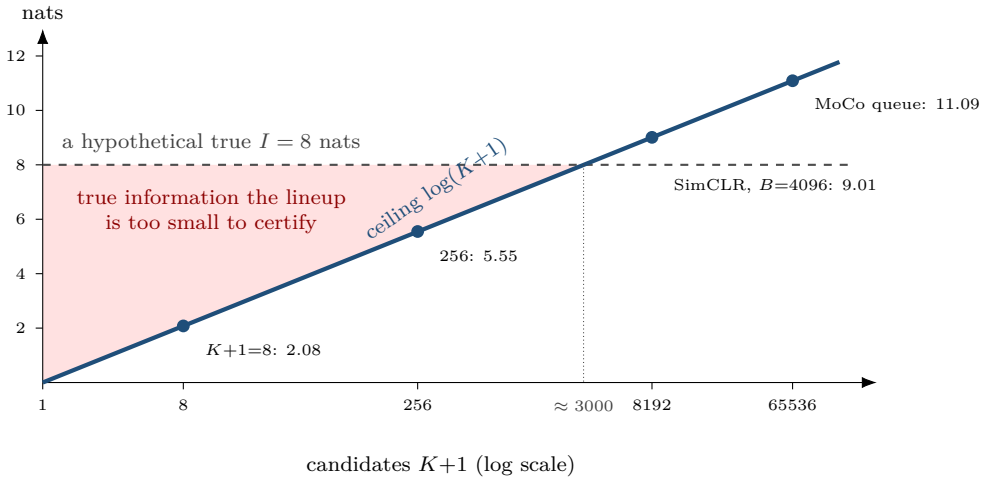


Figure 9.18. The InfoNCE ceiling. The horizontal axis doubles at every step, so the ceiling $\log(K+1)$ is a straight line: each doubling of the candidate pool buys one more bit. If the two views truly share 8 nats (dashed line, a made-up value for illustration), any lineup smaller than about 3000 candidates cannot certify all of it, however good the model is (red region). SimCLR's batch of 4096 gives $2 \cdot 4096 - 1 = 8191$ candidates per query; MoCo's queue gives 65537.

How to read it: the slope. Going from 256 candidates to 65536 is 256 times more negatives for only 8 more bits of headroom. That slow, logarithmic payoff is why both SimCLR and MoCo needed very *large numbers of negatives, and why the returns flatten.*

Failure Modes

The right-hand side of (9.6) can never exceed $\log(K+1)$, even if $\mathcal{L}_{\text{InfoNCE}} \to 0$: whatever the true I is, the bound certifies at most $\log(K+1)$ nats. It saturates at the logarithm of the candidate pool size $K+1$ (one positive plus K negatives), so it is the number of negatives — through $K+1$ — that sets the ceiling. Two corollaries, both empirically visible: (i) *MI maximization is silent about which information* — it cannot tell task-relevant from high-MI-but-useless detail. Two views generated by a perverse transformation can share maximal mutual information while agreeing only on, say, the exact pattern of JPEG compression artifacts; the resulting representation is useless for classification even as the

MI bound is saturated (Tian et al., 2020). (ii) *The bound is loose in practice* (Poole et al., 2019): the slack between $\log(K+1) - \mathcal{L}_{\text{InfoNCE}}$ and the true I grows with the MI itself, so the estimator's bias and variance both grow once the true MI is large relative to $\log(K+1)$. Treating "maximize K" as a theorem-driven prescription is therefore unwarranted; the right heuristic reading is "many negatives sharpen the estimate of a quantity that was never the real goal."

The slide's orientation (slide 62) writes the same bound as $MI[f(\boldsymbol{x}), f(\boldsymbol{x}^+)] + \log N \geq -L$ for a candidate pool of size N: multiplying by -1 and setting $N = K + 1$, this book's candidate count, gives exactly (9.6).

9.5.2 Batch size and the MI-bound trade-off

In SimCLR, the line-up is the batch. Take B images and make two views of each: $2B$ embeddings. For any one of them:

- exactly **one** is its positive (its twin view);
- **itself** is excluded (the indicator in the loss);
- the other $2B - 2 = 2(B-1)$ are negatives.

Equation (9.6) then explains the batch-size story:

- **Small batch** ⇒ few negatives ⇒ a low ceiling $\log(2B-1)$ on how much mutual information the loss can measure.
- **Large batch** ⇒ a higher ceiling, but more compute. Slide 62 shows that more epochs and bigger batches (up to 8192) both raise top-1 linear-probe accuracy, with the biggest gains when training is short. The slide writes the same bound as $MI[f(\boldsymbol{x}), f(\boldsymbol{x}^+)] + \log N \geq -\mathcal{L}$, with N the pool size.

The cost is real: at batch 8192 one step holds $2 \times 8192 = 16384$ activations in memory during backpropagation, and the lecture notes that the largest settings needed Google's TPUs with distributed training — a hardware footnote on what is supposedly an algorithmic idea.

The MI headroom is not free downstream either: growing B grows the expected number of *false negatives* — other samples in the batch sharing the anchor's semantic class. Under a uniform prior over C classes, a batch of B i.i.d. images contains on average B/C samples sharing the anchor's class; at $B = 4096$, $C = 1000$ (ImageNet), each anchor faces on the order of four same-class images — eight views, since each image contributes two — that it is trained to push away. Section 9.6 develops this.

Table 9.3 puts the two sides of that trade next to each other, for one query in a SimCLR batch on ImageNet ($C = 1000$ classes). The ceiling grows by only about 0.7 nats each time the batch doubles, while the number of same-class views being pushed away doubles too.

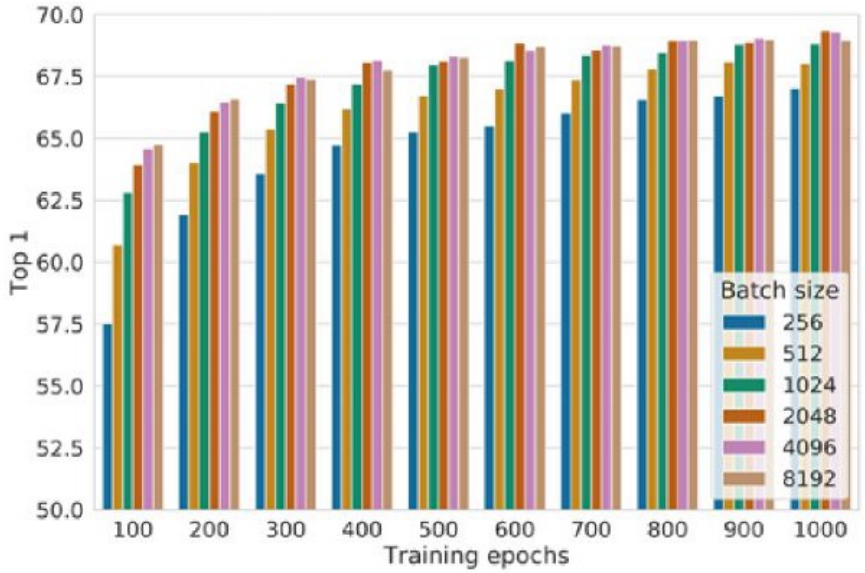


Figure 9. Linear evaluation models (ResNet-50) trained with different batch size and epochs. Each bar is a single run from scratch.[10]

Figure 9.19. SimCLR's own batch-size × training-length sweep (linear ImageNet top-1, ResNet-50). Each group of bars is one training length in epochs; within a group, batch size climbs from 256 to 8192. Two trends sit in this one plot: more epochs always help, and larger batches help most when training is short (consistent with, though not proved by, Equation (9.6): a bigger pool raises the ceiling $\log(2B-1)$).

How to read it: look within a single epoch-count group (say, the bars above "100"). The accuracy climbs steadily left to right as batch size grows — that rising staircase, at fixed training length, is the batch-size effect in isolation.

Source: Lecture 06, slide 62 (data: Chen et al., 2020a, their Figure 9).

batch B	negatives $2(B-1)$	ceiling $\log(2B-1)$	same-class negatives $\approx 2(B-1)/C$
256	510	6.24 nats	0.5
1024	2046	7.62 nats	2.0
4096	8190	9.01 nats	8.2
8192	16382	9.70 nats	16.4

Table 9.3. What a bigger batch buys and what it costs, per query. The ceiling is the most mutual information the loss can certify (Equation (9.6)); the last column is the expected number of in-batch views that secretly share the query's class and are nonetheless treated as negatives.

MoCo (Section 9.8) decouples K from batch size by keeping a queue: negatives are read from a first-in-first-out buffer encoded in previous steps by a momentum-averaged encoder. K can be large regardless of the current batch — typically $K = 65536$ at batch size 256.

Key takeaway

InfoNCE's mutual-information story is real but bounded: driving the loss down does provably raise a lower bound on $I(\boldsymbol{x}; \boldsymbol{z}^+)$, but the bound can never exceed $\log(K+1)$, so the number of negatives is a hard ceiling on what training can claim to have learned. SimCLR raises that ceiling with a bigger batch (expensive: TPU pods, more memory per step); MoCo raises it with a queue instead (Section 9.8). Either way, more negatives is not a free win — Section 9.6, next, is the bill.

9.6 False Negatives and Class Collision

New words used from here on

False negative. A sample the loss treats as a negative — something to push away — that actually shares the anchor's true class. Not a bug in the code; an unavoidable side effect of manufacturing negatives from unlabeled batches instead of from labels.
Class collision. The event that produces a false negative: two different images in the same batch happen to belong to the same underlying class.

Common Confusion

"False negative" does not mean a mistuning of τ; it is an input-level artifact: two *different* samples whose underlying class coincides, treated as if they must differ. In ImageNet minibatches only about 1 in 1,000 in-batch negatives share the anchor's class — but at batch 4096 that is still several per anchor. This is called *class collision*; InfoNCE explicitly pushes them apart.

Class collision. Look again at the line-up of slide 51. Among the negatives the model must reject there is a *different cat*. The loss cannot tell: anything that is not the positive goes into the denominator and is pushed away from the query.

Worse, it is pushed *hardest*. Each negative is pushed with strength $P(j)$ — how similar the model currently thinks it is (Equation (9.4) and the $\partial \mathcal{L}/\partial \ell_j = P(j)$ identity). Once training has made any progress, another cat *looks* similar, so it gets a high $P(j)$ and the strongest push.

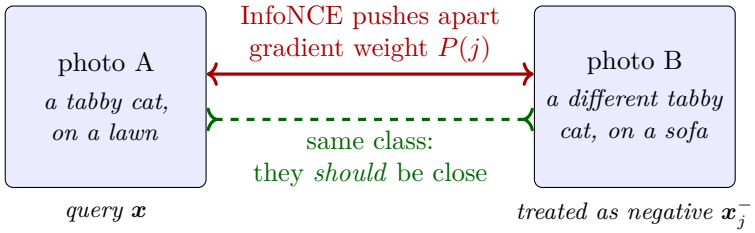
So class collision is not a small tax. The loss's focus on hard negatives is aimed exactly at the pairs the representation should keep close.

Worked Example — The running lineup, now with the truth revealed

Return to the cat lineup of Section 9.3: positive 0.85, other cat 0.75, puppy 0.30, ship −0.10. We now know the other cat is a false negative. How hard does InfoNCE push it away? Two numbers matter: its *share* of all the repulsion, $P(\text{cat})/\sum_{j\geq 1} P(j)$, and the force per unit of similarity, $\partial\mathcal{L}/\partial s_{\text{cat}} = P(\text{cat})/\tau$ (the extra $1/\tau$ comes from $\ell = s/\tau$).

	$P(\text{cat})$	share of repulsion on the cat	force $P(\text{cat})/\tau$
$\tau = 0.5$	0.36	63%	0.71
$\tau = 0.1$	0.27	99%	2.68

At $\tau = 0.1$ nearly all the push the loss hands out goes to the one negative that should not have been pushed at all, and it arrives almost four times as hard. At $\tau = 0.5$ over a third of the push is spent on the genuine negatives (the puppy and the ship) and the force on the cat is much smaller. That is the precise sense in which "moderate τ" is a mitigation for false negatives, and also its cost: the same softening weakens the focus on hard *true* negatives.



Consequence: the loss cannot see the label "cat" on either photo. It only sees "not the same augmented pair," so it spends real gradient teaching the encoder that these two cats look *different* — directly fighting the class structure a downstream classifier needs.

Figure 9.20. A class collision, drawn. Two photographs of different cats end up in the same batch; because neither is the query's own augmented twin, InfoNCE treats the pair as a negative (red) and pushes their embeddings apart, even though the pair is secretly same-class and *should* be pulled together (green, dashed — what a labeled classifier would want).

How to read it: the two arrows on the same pair, pointing opposite ways. Red is what the gradient in Equation (9.4) actually computes; green is what a supervised loss would compute if it had the label. Nothing in the pipeline can tell these two photos apart from a genuinely unrelated negative.

Failure Modes

Class collision at moderate batch size. With B samples drawn i.i.d. from a C-class dataset, the expected number of negative pairs that share a class grows roughly like B^2/C; pushing those pairs apart corrupts the very class structure InfoNCE is trying to extract. Concretely at ImageNet scale: with $C = 1000$ roughly balanced classes and

batch $B = 4096$, each anchor sees on average $4095/1000 \approx 4$ same-class "negatives" per step, and the batch as a whole contains on the order of $B^2/(2C) \approx 8400$ same-class negative pairs that the loss is actively separating. Mitigations: class-aware filtering (when approximate labels are cheap), moderate τ so hard "false" negatives are not pushed with full force, larger K (more true negatives to compete with each false one), and negative-free objectives (Chapter 10).

Augmentation invariance is information loss. Color drop + crop, the two most important SimCLR augmentations, destroy color and global layout. If a downstream task needs either, the representation has irreversibly lost signal. Tian et al. (2020) show that optimal views trade off invariance (shared information between views) against task-relevant information — one wants a *sufficient* view, not a *maximally overlapping* view.

Shortcut learning. Cheap augmentations that naively preserve semantics sometimes introduce low-level cues — e.g., fixed color shifts, crops aligned to a grid — that let the encoder match positive pairs without learning semantics. Designing a harder, more varied augmentation family is the standard countermeasure.

Key takeaway

More negatives raise the mutual-information ceiling (Section 9.5) but also raise the expected number of false negatives (this section) — the two effects trade off, they do not add. There is no way to see this from the loss curve alone: a class-colliding batch and a clean one produce identical-looking InfoNCE values. The only defenses are indirect — moderate temperature, larger K so any one false negative is diluted, and, in Chapter 10, objectives that need no negatives to begin with.

9.7 SimCLR: A Simple Framework in Nine Lines

SimCLR (Chen et al., 2020a) is the crystallization of everything above into a framework with three ingredients: *composed augmentations*, a *projection head*, and *in-batch negatives*. The paper became canonical not for introducing any one ingredient but for systematically ablating them and showing that simple, properly-tuned choices out-perform much more elaborate predecessors like PIRL and CPC.

In plain words

Take one photo, make two differently-edited copies of it, and mix them into a batch with lots of other photos' copies. Train the network so that, for every copy, its own twin scores higher than every other photo's copies in the batch. Everything else in this section — the augmentation recipe, the projection head, the batch size — exists to make that one sentence work well in practice.

Model at a glance: SimCLR	Chen, Kornblith, Norouzi & Hinton, 2020
In one line	Two composed augmentations of one image, a shared encoder plus a small projection head, and InfoNCE over everything else in the batch — no memory bank, no special architecture.
Problem it solves	By 2020, contrastive methods had accumulated machinery — memory banks, jigsaw pretext tasks, bespoke losses — and it was unclear which parts actually mattered. SimCLR strips the recipe down and asks what survives.
Key intuition	Composing two weak augmentations (crop, colour distortion) into one strong one removes the cheap shortcuts one at a time until only genuine semantic structure lets the network solve the task; a disposable projection head lets the loss demand invariances without forcing them onto the representation you actually keep.
Inputs	A batch of N images, each turned into two views by independently sampled augmentations $t, t' \sim \mathcal{T}$.
Architecture	Encoder f (a ResNet) + a small MLP projection head g; both are shared across the two views.
Loss	NT-Xent — InfoNCE with in-batch negatives — Equation (9.4) applied to every ordered pair among the $2N$ views (Listing 9.1).
Training	Adam or LARS, batch sizes up to 8192, TPU pods; temperature near $\tau \approx 0.1$.
What you keep	The backbone's output $\boldsymbol{h} = f(\boldsymbol{x})$; the projection head g is discarded at transfer time (Section 9.7.3).
Strengths	Conceptually the simplest method in the chapter, and empirically it beat every more elaborate predecessor once batch size and augmentation composition were tuned properly.
Weaknesses	Needs a very large batch (or very long training) to get enough negatives; heavy compute footprint; augmentation choices can clash with downstream tasks that need colour or fine spatial detail (Section 9.6).
Leads to	MoCo (next), which buys the same many-negatives benefit without needing a huge batch.

9.7.1 Augmentation composition

The first slide in this block (slide 55) shows the SimCLR augmentation zoo — crop-and-resize, horizontal flip, color drop, color jitter, rotation, cutout, Gaussian noise, Gaussian blur, Sobel filtering — applied to a single dog photo. The decisive ablation finding (presented in the paper, visible in the lecture's figures): *crop + color distortion* are individually weak but *together* are strong. Crop alone teaches locality; color alone teaches color-histogram sensitivity. The composition forces the encoder to recognize the same object across both nuisances at once, and it is at that intersection that semantics surface. No other pair of augmentations recovers the same gain.

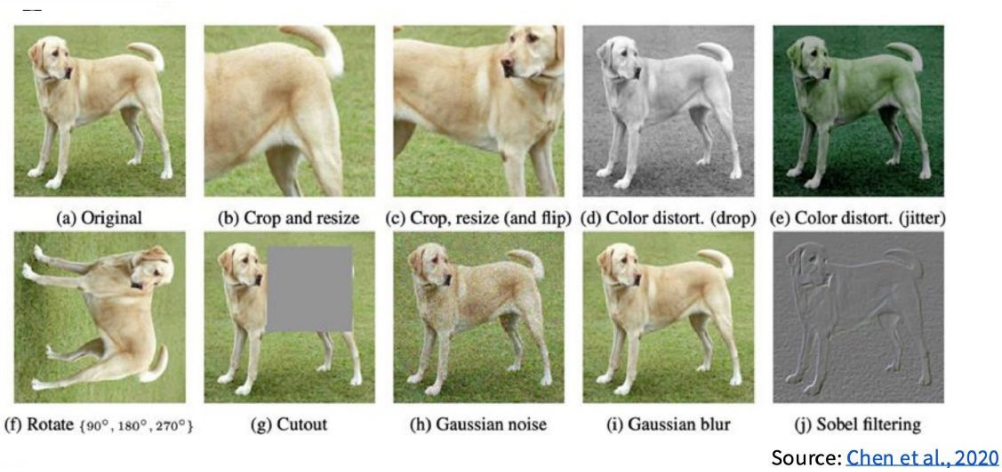


Figure 9.21. The augmentation zoo applied to one photo (SimCLR paper, Figure 4). Each panel is one candidate transformation on its own: crop, flip, colour drop, colour jitter, rotation, cutout, Gaussian noise, Gaussian blur, Sobel filtering. No single panel is the recipe — the ablation result is that *crop* (b) composed with *colour distortion* (d)/(e) is the pair that matters; every other combination trails behind it.

How to read it: panels (b), (d), and (e) specifically. Everything else in the grid is here to show what was tried *and found weaker, not what SimCLR actually uses at training time.*

Source: Lecture 06, slide 55 (data: Chen et al., 2020a, their Figure 4).

Why crop alone is weak: the colour shortcut. Two random crops of one photo almost always have the *same overall colours*. A crop moves a window over the picture; it does not repaint it. So with crops alone the network can win by *matching colour histograms* — before it has learned a single edge or object part. This is the shortcut theme of Chapter 8 again.

Colour distortion (random brightness, contrast, saturation and hue, plus the occasional switch to greyscale) destroys that clue. Two views of the same image may now have *different* colours, so the only reliable way to match them is the content itself: the shape of the scene, the object's parts, where they sit relative to each other.

So crop + colour is not "two weak tricks stacked". Each augmentation removes one cheap solution, until only the expensive, semantic solution is left. Chapter 11's discussion of augmentation engineering scales up this same logic.

9.7.2 The algorithm, walk by walk

Listing 9.1 restates SimCLR's Algorithm 1 in the same PyTorch-like style as the lecture's four build slides (56–59).

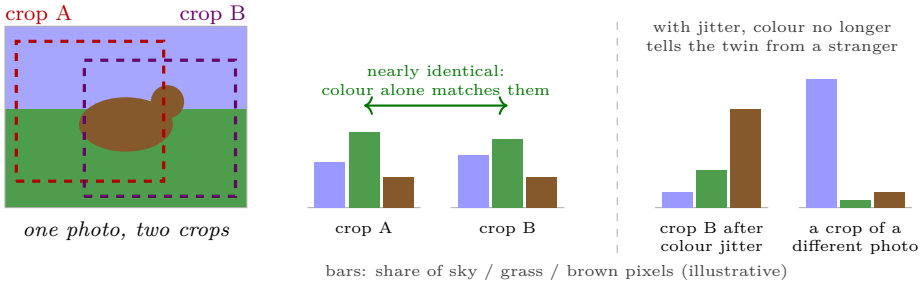


Figure 9.22. The colour shortcut, drawn. Two crops of one photo (red and purple boxes) have almost the same colour histogram, so an encoder can match them by colour statistics alone and never look at the dog. After colour jitter, the second crop's histogram (third panel) no longer resembles the first, and it is no more similar to it than a crop of a different photo is (fourth panel). The only reliable clue left is the content: the dog's shape and its parts.

How to read it: the first two histograms. When positives can be matched by something this cheap, the loss is satisfied long before any semantic feature is needed. Colour distortion exists to take that option away.

```
# input: batch x of N images; augmentations T; encoder f; projection g
for k in 1..N:                                  # build 1: positive pair
    t1, t2 = sample(T), sample(T)
    v[2k-1] = t1(x[k])                          # first view
    v[2k]   = t2(x[k])                          # second view

for i in 1..2N:                                 # build 2: project to contrastive space
    h[i] = f(v[i])                              # representation (kept afterwards)
    z[i] = g(h[i])                              # projection (used only in the loss)

for i, j in pairs(2N):                          # build 3: InfoNCE per ordered pair
    s[i,j] = dot(z[i], z[j]) / (norm(z[i]) * norm(z[j]))     # cosine similarity
    loss[i,j] = -log(exp(s[i,j]/tau) / sum(exp(s[i,k]/tau) for k != i))

L = sum(loss[2k-1,2k] + loss[2k,2k-1] for k in 1..N) / (2N)  # build 4
update f, g by SGD/Adam on L
return f                                        # discard g at transfer time
```

Listing 9.1. SimCLR — Algorithm 1 of Chen et al. (2020a) in PyTorch-like pseudocode, broken into the same four builds the lecture walks through on slides 56–59.

Step by step — SimCLR's forward pass, build by build

Step 1. Make two views of everything. For each image $\boldsymbol{x}_k$ in the batch, sample two augmentations and produce $\tilde{\boldsymbol{x}}_{2k-1}, \tilde{\boldsymbol{x}}_{2k}$ — a batch of N images becomes $2N$ views.

Step 2. Encode, then project. Push every view through the shared encoder to get $\boldsymbol{h}_i = f(\tilde{\boldsymbol{x}}_i)$, then through the shared projection head to get $\boldsymbol{z}_i = g(\boldsymbol{h}_i)$. The loss only ever looks at $\boldsymbol{z}$.

Step 3. Score every pair. For every ordered pair (i, j) among the $2N$ views, compute cosine similarity s_{ij} and the InfoNCE term $-\log\left(e^{s_{ij}/\tau} / \sum_{k \neq i} e^{s_{ik}/\tau}\right)$.

Step 4. Average and update. Sum the loss over the $2N$ ordered positive pairs, average, and take one gradient step on f and g together.

Step 5. At transfer time, throw g away. Only f ships; $\boldsymbol{h}$, not $\boldsymbol{z}$, is the representation every downstream probe uses.

The lecture builds SimCLR in four steps (slides 56–59):

1. **Two views.** From each image x_k, make two random augmentations. With a batch of N images you now have $2N$ views.
2. **Encode, then project.** Each view goes through the encoder f to give $\boldsymbol{h}$, then through a small projection head g to give $\boldsymbol{z}$. The loss only ever looks at $\boldsymbol{z}$ — but $\boldsymbol{h}$ is what you keep afterwards.
3. **A classification per view.** For view i, the question is: "out of the other $2N-1$ views, which one is my partner?" Score that with cross-entropy.
4. **Average over all $2N$ views.** Every view takes a turn as the one asking. Everything in the batch that is not its partner automatically serves as a negative — nothing extra has to be stored.

The label programmability of this loss is worth stressing: there is no explicit negative set; the choice of batch *is* the negative set. Equivalently, the denominator of each per-pair loss contains $2N-1$ terms: the positive twin, plus the $2N-2$ in-batch negatives. This is what makes the batch size N a load-bearing hyperparameter (Section 9.5.2).

Worked Example — One batch of size $N = 4$, walked by hand

Take a toy batch with $N = 4$ images, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4$, producing $2N = 8$ views $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_8$ in the order $(t(\boldsymbol{x}_1), t'(\boldsymbol{x}_1), t(\boldsymbol{x}_2), t'(\boldsymbol{x}_2), t(\boldsymbol{x}_3), t'(\boldsymbol{x}_3), t(\boldsymbol{x}_4), t'(\boldsymbol{x}_4))$ — in the listing's indexing, pairs are $(1,2)$, $(3,4)$, $(5,6)$, $(7,8)$. Suppose the encoder and projection produce, after ℓ_2 normalization, embeddings $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_8$ whose cosine similarities *with* $\boldsymbol{z}_1$ *as query* are

$$s(\boldsymbol{z}_1, \boldsymbol{z}_2), \ldots, s(\boldsymbol{z}_1, \boldsymbol{z}_8) \;=\; 0.88,\ 0.51,\ 0.46,\ 0.69,\ 0.63,\ 0.28,\ 0.34,$$

so $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ agree (similarity 0.88), there is one *hard negative* $\boldsymbol{z}_5$ (0.69) — a view that resembles $\boldsymbol{x}_1$ despite coming from a different image — and the remaining four are progressively easier.

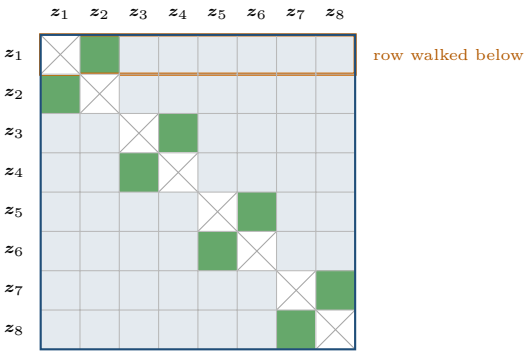
Build 3 of the listing: compute the per-pair InfoNCE loss for the ordered pair $(1, 2)$ at, say, $\tau = 0.5$. The unnormalized logits for query $\boldsymbol{z}_1$ are the similarities divided by τ: (1.76, 1.02, 0.92, 1.38, 1.26, 0.56, 0.68) for candidates $\boldsymbol{z}_2, \ldots, \boldsymbol{z}_8$. Exponentiating: (5.81, 2.77, 2.51, 3.97, 3.53, 1.75, 1.97); the denominator is their sum, 22.31. The positive probability is $5.81/22.31 \approx 0.26$ and $\ell(1,2) = -\log 0.26 \approx 1.35$. The reversed term $\ell(2,1)$

uses $\boldsymbol{z}_2$'s row of the similarity matrix, which in a real encoder need not be exactly mirrored; here it yields a similar value, ≈ 1.3.

Build 4: the batch loss averages $\ell(2k-1, 2k) + \ell(2k, 2k-1)$ over $k = 1, \ldots, 4$ — eight ordered terms in all. Two things to notice. First, the *denominator has* $2N - 1 = 7$ *terms*: one positive plus $2N - 2 = 6$ negatives (the "sum to $2N$" in the paper's Algorithm 1 runs over all indices and uses the indicator to remove the query). Second, if $\boldsymbol{x}_3$ happens to be another picture of the same object class as $\boldsymbol{x}_1$, the high similarity $s(\boldsymbol{z}_1, \boldsymbol{z}_5) = 0.69$ is precisely the *false negative* effect of Section 9.6: the loss assigns the largest push of any negative to the very embedding it should, semantically, be pulling closer. Figure 9.23 draws the whole $2N \times 2N$ matrix this batch produces, and then zooms into the one row computed above.

The whole 8×8 similarity matrix

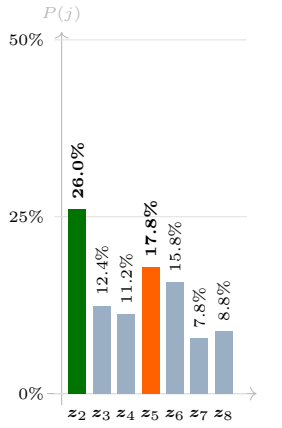
diagonal excluded, positives marked, negatives shaded



Green cells are the four ground-truth pairs; white diagonal cells never enter the loss (a view is not its own negative). Every remaining cell is an in-batch negative.

Row 1: the softmax $\boldsymbol{z}_1$ actually sees

$\tau = 0.5$, *candidates* $\boldsymbol{z}_2, \ldots, \boldsymbol{z}_8$



$\boldsymbol{z}_2$ (positive, green) wins with 26.0% of the softmax mass, but $\boldsymbol{z}_5$ (orange) — the hard negative, a different image that looks like $\boldsymbol{x}_1$ — takes 17.8%, more than any true negative.

Figure 9.23. The $N = 4$ SimCLR batch, walked by hand. *Left:* the full $2N \times 2N$=8×8 cosine-similarity matrix. Every row has exactly one green (positive) cell and the diagonal is always excluded — this is the entire structure InfoNCE optimizes, batch after batch. *Right:* zooming into the outlined row (query $\boldsymbol{z}_1$), the softmax at $\tau = 0.5$ over the seven candidates $\boldsymbol{z}_2, \ldots, \boldsymbol{z}_8$, reproducing $5.81/22.31 \approx 0.260$ for the positive from the worked example.

How to read it: the orange bar. It is a negative *by the batch's construction — $\boldsymbol{z}_5$ comes from a different source image, $\boldsymbol{x}_3$ — yet it captures more than two-thirds as much probability as the true positive (*17.8% *against* 26.0%*), and so the largest push of any negative. That single bar is the whole false-negative problem Section 9.6 names.*

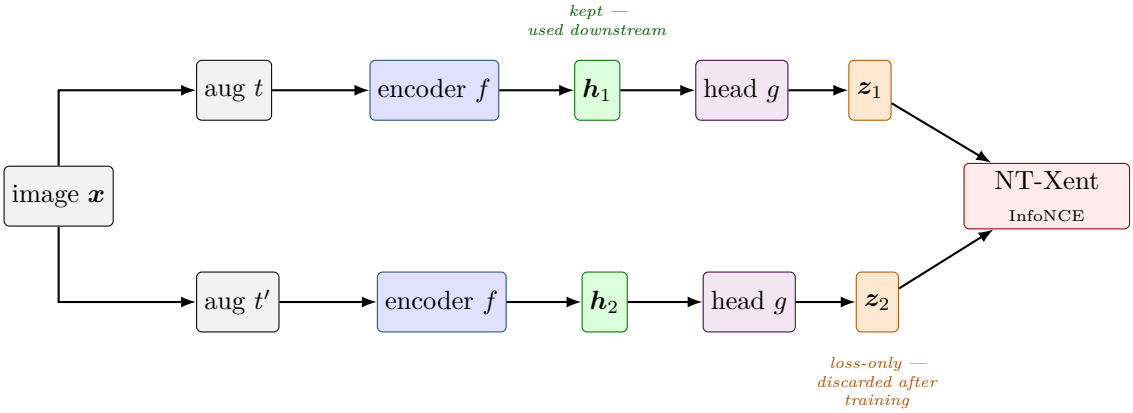


Figure 9.24. SimCLR's full pipeline, with the fate of every intermediate marked. Two augmentations of one image pass through the shared encoder f to produce $\boldsymbol{h}_1, \boldsymbol{h}_2$ (green) — *this* is the representation kept and used downstream. Only then does a shared projection head g map $\boldsymbol{h}$ to $\boldsymbol{z}$ (orange), and only $\boldsymbol{z}$ ever reaches the loss. Because the loss's invariance pressure lands on $\boldsymbol{z}$, not $\boldsymbol{h}$, the backbone is never directly punished for discarding information — colour, exact position — that a downstream task might still need.

How to read it: trace one branch left to right and watch the colour change. Green ($\boldsymbol{h}$) survives training; orange ($\boldsymbol{z}$) does not. Everything the loss touches is orange — that is the whole argument for why SimCLR discards its own projection head at transfer time.

9.7.3 The projection head: information is kept in $\boldsymbol{h}$, not $\boldsymbol{z}$

Slide 61 shows the projection-head ablation: a linear head reaches about 60% linear-probe top-1, a non-linear MLP slightly more, and no head about 50%. Why?

The key point: the loss acts on $\boldsymbol{z}$, not on $\boldsymbol{h}$. The loss forces $\boldsymbol{z}$ to ignore the augmentations — colour, position, scale. If it acted directly on $\boldsymbol{h}$, $\boldsymbol{h}$ would have to throw all that information away. The projection head takes the hit instead:

- g learns to strip out augmentation-related variation, so $\boldsymbol{z}$ is clean;
- $\boldsymbol{h}$ *keeps* that information, because g did the discarding.

After training, g is thrown away and $\boldsymbol{h}$ is used downstream. Slide 61 shows the same thing with arrows: "Representation" sits between f and g; "Maximize agreement" sits above g.

Why does a *non-linear* head do better?

- **A linear head is limited.** $g(\boldsymbol{h}) = W\boldsymbol{h}$ can only drop whole, fixed directions of $\boldsymbol{h}$ (by projecting to fewer dimensions). Everything else the loss demands of $\boldsymbol{z} = W\boldsymbol{h}$ is a linear constraint on $\boldsymbol{h}$ itself.
- **A small MLP can.** In $g(\boldsymbol{h}) = W^{(2)}\sigma(W^{(1)}\boldsymbol{h})$ the ReLU σ can zero out the directions of $\boldsymbol{h}$ that hold nuisance information (colour profile, exact crop position), and can do so differently for different inputs. Those details stop at the head and never reach $\boldsymbol{z}$.

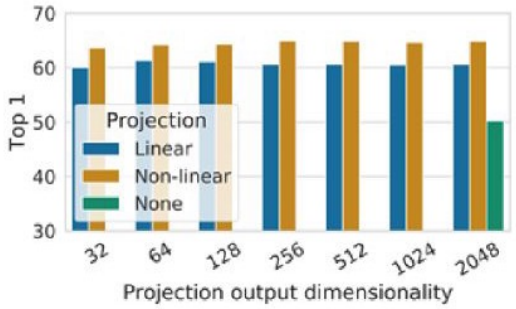


Figure 9.25. The projection-head ablation (slide 61): linear-probe top-1 on ImageNet as a function of the projection head's output dimensionality, for three head types. "None" (green, output dimension forced to 2048) trails both linear (blue) and non-linear (orange) heads by roughly 10–15 points, and the non-linear head is consistently a little ahead of the linear one across every dimensionality tested.

How to read it: the green bar at 2048 — the one case where no head exists and the loss acts directly on $\boldsymbol{h}$*. That is the single data point that proves the head is not a convenience; removing it costs real accuracy.*

Source: Lecture 06, slide 61 (data: Chen et al., 2020a).

The result is a division of labour. The loss demands "ignore colour and position", and the head is where that demand gets satisfied — not the representation $\boldsymbol{h}$. The encoder never has to *delete* the nuisance details; it only has to hand them to a head that will discard them. Figure 9.26 draws what that bottleneck buys geometrically.

Simplified intuition vs. more precise view

Simple story. g is a buffer; it protects f from the invariance pressure of the contrastive loss.

Precise story. Optimizing $\boldsymbol{z}$ for invariance yields a bottleneck $T_{\text{bottleneck}}(\boldsymbol{x})$ that removes transformation-specific components. Placing this bottleneck downstream of the representation lets f keep those components intact — they are simply not measured by the loss. The projection-head gain is robust across output dimensions 32–2048, with the non-linear MLP consistently ahead of a linear map.

Everyday analogy — translating for a customs form

Imagine filling out a customs declaration that only accepts a description in a restricted vocabulary — category and approximate value, nothing else. You still remember every detail of what is actually in the suitcase; you simply wrote a deliberately impoverished summary *for this one form.* The projection head is that form: it is trained to produce a summary $\boldsymbol{z}$ good enough to pass the "same image?" test and nothing more, while the backbone $\boldsymbol{h}$ keeps the full, undegraded memory of what it saw.

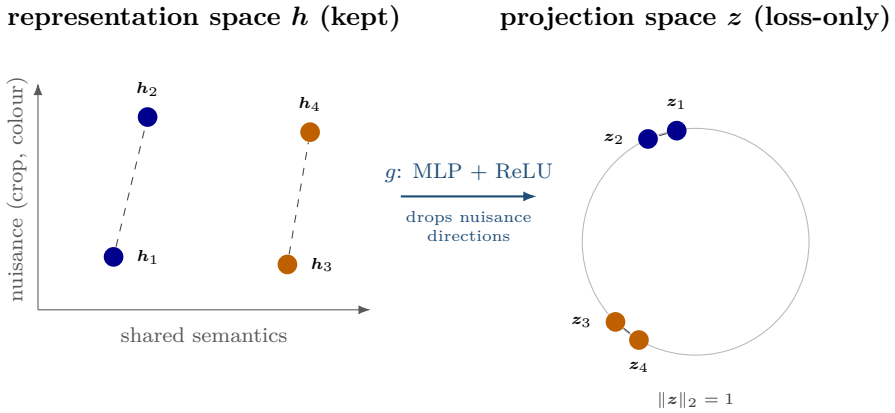


Figure 9.26. What the projection head throws away, geometrically. *Left:* in the encoder's representation space $\boldsymbol{h}$ (the part SimCLR keeps), the two views of one image — blue pair $\boldsymbol{h}_1, \boldsymbol{h}_2$, orange pair $\boldsymbol{h}_3, \boldsymbol{h}_4$, the dashed segments join same-image views — sit far apart, because augmentation moves a point mostly along the nuisance axis while different images separate along the shared-semantics axis. *Right:* the head g maps $\boldsymbol{h}$ onto the hypersphere where InfoNCE acts; its ReLU gates zero exactly the nuisance directions, so views of one image collapse to nearly the same point ($\boldsymbol{z}_1 \approx \boldsymbol{z}_2$) while the two images stay far apart. The loss's invariance pressure is spent crossing g, which is why every direction of $\boldsymbol{h}$ can stay intact for downstream use.

How to read it: the four dashed segments and what happens to them. On the left each segment is long; on the right each has shrunk to a point. The distance that disappeared between the two panels is precisely the information g was trained to discard — and $\boldsymbol{h}$ was never asked to discard.

Where the analogy breaks. A traveller chooses on purpose what to leave off a customs form. The projection head does not decide by design which details to discard — it discovers, through gradient descent, whichever information happens to be irrelevant to matching augmented pairs, and that is only a proxy for what some future downstream task will actually need.

A metaphor that survived several rounds of study sessions on this slide makes the division of labor vivid.

Learning note from study sessions

Think of training an art appraiser who must certify that two photographs depict the same painting, where one photograph has been color-distorted and tightly cropped. The rule you impose is: do your careful looking "in your head" (the backbone's representation $\boldsymbol{h}$, allowed to notice color and composition), then write on a scratchpad (the head's output $\boldsymbol{z}$) a short, deliberately color-free summary used only for the matching test. When training ends, the scratchpad is thrown away; what transfers is the eye, not the note.

The scratchpad-throwaway step is not an implementation afterthought — it is exactly SimCLR's protocol. At transfer time, g is simply not called; the frozen backbone's output $\boldsymbol{h} = f(\boldsymbol{x})$ is the representation handed to linear probes and downstream heads. Keeping g

would hand the downstream task a coordinate system deliberately engineered to be blind to color and scale.

Research Frontier

Chen et al. (2020b) (SimCLRv2) extend the recipe in three directions: deeper/wider ResNets (up to 4×), deeper projection heads (kept partly in the transfer network), and a memory bank à la MoCo. The result — big self-supervised models fine-tuned on a few labels are strong semi-supervised learners — foreshadows the self-training chapter (Chapter 12). The projection-head-plus-composed-augmentation backbone is unchanged.

Paper Analysis: SimCLR (Chen, Kornblith, Norouzi, Hinton, 2020)

Paper: *A Simple Framework for Contrastive Learning of Visual Representations* (Chen et al., 2020a).

Problem. Contrastive methods had grown complicated (memory banks, special-purpose architectures, many moving parts). SimCLR asks: how far do the simplest possible ingredients — composed augmentations, in-batch negatives, projection head — go?

Prior work. CPC (van den Oord et al., 2018b) provided InfoNCE and the MI view; PIRL, NPID-like memory banks, and CPCv2 populated the performance frontier; deep metric learning supplied the losses.

Insight. Three ingredients carry nearly all the signal: (i) composing crop-plus-color-distortion; (ii) a nonlinear projection head whose output, not whose input, is trained to be invariant; (iii) large batches providing many in-batch negatives, with temperatures tuned near 0.1.

Method. Listing 9.1: B images, two views each, ResNet encoder, MLP projection, NT-Xent loss, Adam/LARS at batch sizes up to 8192 on TPU pods.

Experiments. Linear-probe ImageNet with frozen ResNet-50; few-label fine-tuning on 1%/10% of ImageNet; transfer to 12 downstream datasets. The slide 60 table shows the 1%/10%-label regime: SimCLR (4×) achieves Top-5 85.8 (1% labels) and 92.6 (10% labels), ahead of prior self-supervised baselines (InstDisc 39.2/77.4, PIRL 57.2/83.8, CPCv2 77.9/91.2).

Results. ResNet-50 linear-probe top-1 ≈ 69.3 (slide 54 teaser plot), rising with width (2× and 4× models).

Limitations. Batch size really matters (Section 9.5.2); compute is heavy; the framework is vision-specific in its augmentation family though not in its loss; augmentation semantics can clash with downstream tasks (Section 9.6).

Later work. SimCLRv2 (Chen et al., 2020b) scales up; negative-free successors (Chapter 10) ask whether the negatives were ever the critical part.

What to remember. Augmentations must be *composed*; put the invariance burden on a head you then discard; batch size buys MI headroom; the whole loop fits in nine lines of pseudocode.

Method	Architecture	Label fraction 1%	10%
		Top 5	
Supervised baseline	ResNet-50	48.4	80.4
Methods using other label-propagation:			
Pseudo-label	ResNet-50	51.6	82.4
VAT+Entropy Min.	ResNet-50	47.0	83.4
UDA (w. RandAug)	ResNet-50	-	88.5
FixMatch (w. RandAug)	ResNet-50	-	89.1
S4L (Rot+VAT+En. M.)	ResNet-50 (4×)	-	91.2
Methods using representation learning only:			
InstDisc	ResNet-50	39.2	77.4
BigBiGAN	RevNet-50 (4×)	55.2	78.8
PIRL	ResNet-50	57.2	83.8
CPC v2	ResNet-161(*)	77.9	91.2
SimCLR (ours)	ResNet-50	75.5	87.8
SimCLR (ours)	ResNet-50 (2×)	83.0	91.2
SimCLR (ours)	ResNet-50 (4×)	**85.8**	**92.6**

Table 7. ImageNet accuracy of models trained with few labels.

Figure 9.27. The few-label fine-tuning table the paperbox above quotes (SimCLR paper, Table 7): ImageNet top-5 accuracy after fine-tuning on only 1% or 10% of labels. The boxed rows are SimCLR at three widths; the "representation learning only" block above it is the prior self-supervised frontier (InstDisc, BigBiGAN, PIRL, CPCv2) that SimCLR (4×) clears by a wide margin, and it even edges past several label-propagation methods that see labels directly during training.

How to read it: the "representation learning only" block versus the boxed SimCLR rows just below it — same evaluation protocol, same label budget, a jump of about 8 points with 1% of the labels.

Source: Lecture 06, slide 60 (data: Chen et al., 2020a, their Table 7).

What changed — InfoNCE / CPC → SimCLR

Before. CPC demonstrated InfoNCE works, wrapped in an autoregressive context network built for sequential and patch-based prediction — powerful, but architecturally specific and not obviously the right recipe for a single static image.

What changed. Strip the autoregressive machinery entirely. Replace it with two plain augmented views, add a disposable projection head, and scale the batch instead of engineering the architecture. The paper's contribution is almost entirely *which simple ingredients matter and how much*, established by systematic ablation rather than a new mechanism.

What it cost. Negatives now come only from the current batch, so getting enough of them means making the batch enormous — up to 8192 images, requiring TPU pods most labs did not have. That single cost is exactly what MoCo (next) is built to remove.

Key takeaway

SimCLR's result is almost anticlimactic and that is the point: no memory bank, no pretext task, no bespoke architecture — just composed augmentations, a throwaway projection head, and enough in-batch negatives. Every one of those three ingredients is load-bearing (the ablations remove each in turn and accuracy drops), and none of them is exotic. The one price tag left on the table is batch size, which is exactly where MoCo starts.

9.8 MoCo: Momentum Contrast and the Dictionary as a Queue

MoCo (He et al., 2020) attacks SimCLR's central hardware constraint: the batch size. (Historically MoCo appeared a few months *before* SimCLR; the lecture presents them in this order because SimCLR is the simpler recipe.)

The tension is this: more negatives K is good for the MI bound, but with in-batch negatives, more negatives means a bigger batch, and a huge batch (SimCLR's solution) requires TPUs. The older alternative, a memory bank, holds many negatives cheaply, but its entries were computed by many different, older versions of the encoder, so the keys are inconsistent with one another and with the current query.

MoCo instead treats contrastive learning as *dictionary look-up*: an encoder $\boldsymbol{f}_q$ turns the query view into $\boldsymbol{q}$; a *key encoder* $\boldsymbol{f}_k$ turns other views into keys $\boldsymbol{k}$ — a mini-batch supplies some keys, and a long *queue* of older keys supplies the rest. InfoNCE is a dictionary look-up: the positive key should match, the rest not.

The names are new but the roles are not. The **query** is the anchor. A **key** is any candidate's embedding: one key is the positive (the other view of the query's image), and all the rest are negatives. So MoCo's loss is the same InfoNCE lineup as before; only the place the negatives are stored changes.

In plain words

Instead of asking a huge batch to supply every negative on the spot, keep a running list of recently-seen keys in a queue, like a "recently returned books" shelf at a library. New keys get pushed onto the queue by a version of the encoder that only ever changes slowly (a moving average of the real encoder), so entries already on the shelf do not go stale before they get used for comparison.

Model at a glance: MoCo (Momentum Contrast) He, Fan, Wu, Xie & Girshick, 2020

In one line	A query encoder trained by gradient descent and a key encoder trained only by exponential moving average, with negatives read from a FIFO queue of past keys instead of the current batch.

Problem it solves	SimCLR needs a huge batch to get enough negatives. MoCo wants many negatives *and* an affordable batch size at the same time.
Key intuition	Decouple two things SimCLR ties together: how many negatives you use, and how large your batch is. A slowly-updated key encoder makes it safe to compare the query against keys computed several steps ago.
Inputs	Two augmented views per image: one becomes the query, one becomes a key.
Architecture	Two encoders with identical structure but different update rules: f_q by gradient, f_k by EMA (Equation (9.7)).
Loss	InfoNCE, Equation (9.4), with the positive key from the current batch and K negative keys read from the queue.
Training	$m = 0.999$, queue size $K = 65536$, temperature $\tau = 0.07$, batch size as small as 256.
What you keep	The query encoder f_q; f_k, the queue, and the projection machinery are all training-time-only scaffolding.
Strengths	Many, consistent negatives on a single machine; the EMA idea it introduces becomes the standard teacher mechanism for BYOL and DINO (Chapters 10–11); SimSiam later showed it can even be dropped.
Main failure modes	Sensitive to m at both extremes (Section 9.8.1); more moving parts than SimCLR — two encoders, a queue — for comparable final accuracy.
Leads to	MoCo v2 (adds SimCLR's projection head and stronger augmentation) and MoCo v3 (ports the recipe to Vision Transformers); the EMA teacher itself leads directly into Chapters 10–11.

Everyday analogy — a library that restocks slowly on purpose

Picture a reference library where the query desk changes its filing rules every single day (the query encoder, updated every step), but the shelves themselves are only ever reorganized by a librarian who moves things a tiny bit at a time (the momentum key encoder). Because the shelves change slowly and smoothly, a book you compared against yesterday is still filed closely enough to today's rules to be a fair comparison, even though it was not re-filed today.

Where the analogy breaks. A real library keeps every book forever. MoCo's queue is finite and forgets: once K new keys have been pushed in, the oldest are discarded, so "the shelf" is really a sliding window over the last K/B batches, not a permanent archive.

9.8.1 The EMA encoder

How should $\boldsymbol{f}_k$ relate to $\boldsymbol{f}_q$? Copying $\boldsymbol{f}_q$ verbatim (as in PIRL's memory bank) reintroduces staleness. MoCo updates $\boldsymbol{f}_k$ as an exponential moving average:

$$\boldsymbol{\theta}_k \leftarrow m\,\boldsymbol{\theta}_k + (1-m)\,\boldsymbol{\theta}_q, \tag{9.7}$$

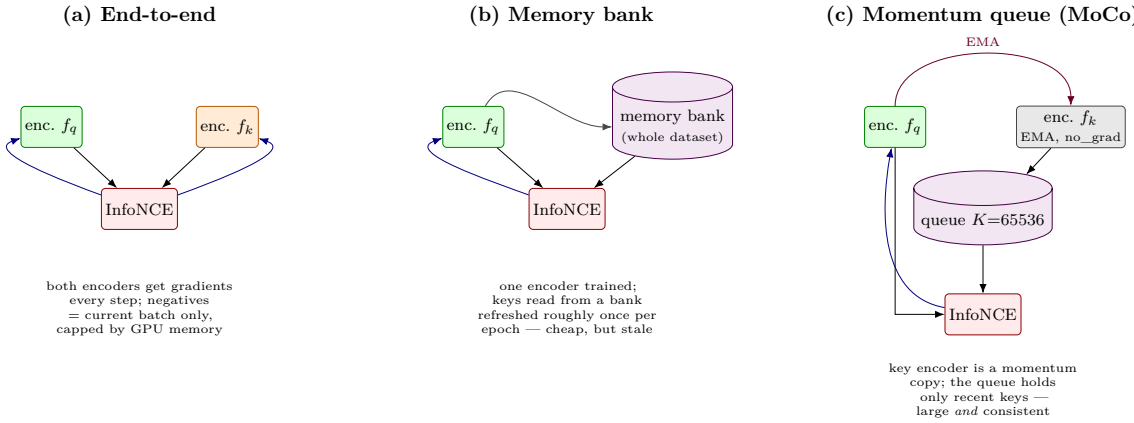


Figure 9.28. The three ways to get negatives, side by side (after the comparison figure in the MoCo paper). *(a)* End-to-end: simplest, but every negative must fit in the current batch, and both encoders move together every step. *(b)* Memory bank (PIRL's approach, Section 9.1.2): negatives come from a large stored dictionary, so K can be big, but the bank refreshes slowly and its entries lag far behind the current encoder. *(c)* MoCo: a momentum-updated key encoder keeps a queue's entries only a few steps stale, combining (a)'s consistency with (b)'s scale.

How to read it: the colour of the key-side box in each panel. Green — same as the query encoder, gradient-trained — in (a); grey, static dictionary in (b); orange-grey with a purple EMA arrow looping back from the query encoder in (c). That loop is the one new idea MoCo adds.

with m close to 1 (e.g. 0.999). Only $\boldsymbol{\theta}_q$ receives gradients; the queue keys are computed under sg — the red "no_grad" slash on slide 64. The instructor's marginal note on slide 65 — "this will prove very useful! usually now called exponential moving average" — is correct: the same EMA teacher now underpins BYOL and DINO (Chapters 10–11).

What the momentum m controls. Look at the two extreme settings.

$m = 0$**: the key encoder is an exact copy of the query encoder, updated every step.** Sounds sensible — but the queue still holds keys made by *older* versions of the encoder. If the encoder changes quickly, old and new keys live in different coordinate systems, and the loss ends up comparing incompatible numbers. Training becomes unstable.

$m = 1$**: the key encoder never changes.** It stays at its random starting weights forever, so the keys are meaningless, and the query encoder learns to tell apart random projections. Wasted effort.

$m = 0.999$ **(what MoCo uses):** the key encoder moves, but only 0.1% of the way towards the query encoder per step. It learns, yet stays consistent enough across the queue that old and new keys remain comparable.

So: m **close to 1, but not equal to 1.** MoCo uses $m = 0.999$. Two numbers make that concrete:

- **Memory of the key encoder:** about $1/(1-m) = 1000$ steps. It is a smoothed average of roughly the last thousand query encoders.
- **Age of the oldest key in the queue:** $K/B = 65536/256 = 256$ batches. Over that time the key encoder has moved only about a fifth of the way ($1 - 0.999^{256} \approx 0.23$) towards the query encoder.

Unrolling the update T steps back shows exactly what the key encoder is:

$$\boldsymbol{\theta}_k^{(t)} = m^T \boldsymbol{\theta}_k^{(t-T)} + (1-m) \sum_{i=0}^{T-1} m^i \, \boldsymbol{\theta}_q^{(t-i)},$$

a weighted average of past query encoders, with the weights shrinking by a factor m for every step into the past. Keys are still slightly stale — but the staleness is *small and smooth*, not large and jumpy as in PIRL's memory bank, where a stored feature could be a whole epoch old. That single property is what MoCo buys that PIRL could not.

A thought experiment that makes m concrete. Suppose that at step 0 the query encoder suddenly learns something new, and its weights jump from 0 to 1 and then stay there (one weight standing in for all of them). Unrolling the EMA, the key encoder after t steps has moved $1 - m^t$ of the way. Figure 9.29 plots that for three values of m, with the life span of one queue entry (256 steps) marked. The question to ask of each curve is: *while one key sits in the queue, how much does the encoder that made it change?*

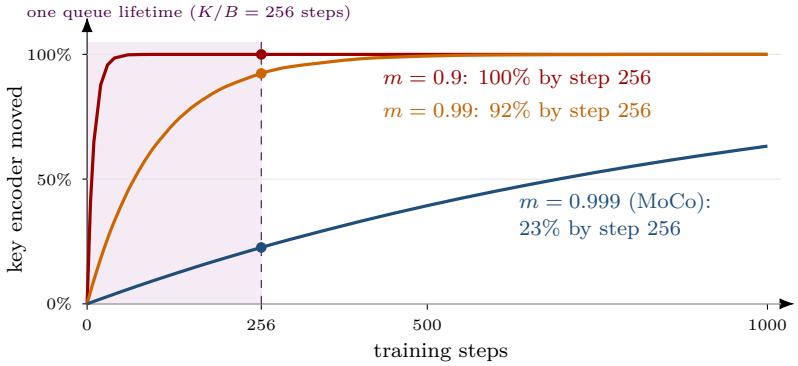


Figure 9.29. How far the key encoder moves after a sudden change in the query encoder, $1 - m^t$, for three momentum values. The shaded band is the life span of one key in MoCo's queue ($65536/256 = 256$ steps). With $m = 0.9$ or 0.99 the key encoder has essentially finished changing before the oldest keys leave the queue, so the queue holds keys from what are effectively different encoders. With $m = 0.999$ it has moved only 23%, so every key in the queue was made by a nearly identical encoder.

How to read it: where each curve crosses the dashed line. That height is "how inconsistent the oldest and newest keys in the queue are". MoCo picks m so that this height stays small.

The same arithmetic explains why the EMA also *smooths*: a query encoder that jitters up and down from one step to the next gets averaged over roughly $1/(1-m) = 1000$ steps, so

the jitter cancels and only the steady trend reaches the keys. Figure 9.30 shows the other half of the mechanism, the queue itself.

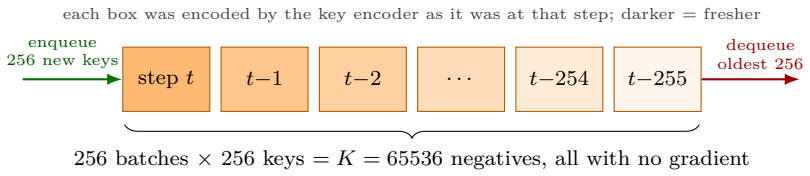


Figure 9.30. MoCo's queue as a conveyor belt. Every training step pushes the current batch's keys in on the left and drops the oldest batch off the right. The query is compared against all 65536 keys on the belt, but only the current batch had to be encoded this step, which is why the negative count is decoupled from the batch size.

How to read it: the colour fade from left to right. The oldest box was made 255 steps ago; Figure 9.29 is what guarantees that, with $m = 0.999$, it still speaks the same language as the newest one.

9.8.2 The pseudocode listing

Listing 9.2 reprints the lecture's six-slide build of the official pseudocode with minimal cosmetic changes.

```
# f_q, f_k : query and key encoders (conv nets)
# queue    : K encoded keys (C x K), init random
# m, t     : momentum (e.g. 0.999), temperature

for x in loader:
    x_q, x_k = aug(x), aug(x)                    # two random views     [slide 66]
    q = f_q(x_q)                                 # queries, N x C
    k = f_k(x_k)                                 # keys under no_grad
    k = k.detach()                               # stop gradient        [slide 67]

    l_pos = bmm(q.view(N,1,C), k.view(N,C,1))    # (N,1) positive logits
    l_neg = mm(q.view(N,C), queue.view(C,K))     # (N,K) negative logits
    logits = cat([l_pos, l_neg], dim=1)          # (N, 1+K)             [slide 68]
    labels = zeros(N)                            # positive = index 0
    loss = CrossEntropyLoss(logits / t, labels)  #                      [slide 69]

    loss.backward()
    update(f_q.params)
    f_k.params = m*f_k.params + (1-m)*f_q.params  # EMA                 [slide 70]

    enqueue(queue, k); dequeue(queue)            # FIFO                 [slide 71]
```

Listing 9.2. MoCo — PyTorch-like pseudocode from the MoCo paper's Algorithm 1, walked line-by-line on the lecture's slides 66–71.

Step by step — MoCo's forward pass, one training step

Step 1. Make two views. One image, two augmentations: one view becomes the query input, the other the key input.

Step 2. Encode with two different encoders. $q = f_q(x_q)$ with gradients tracked; $k = f_k(x_k)$ computed under no-gradient and then detached.

Step 3. Score against the positive and the whole queue. One positive logit (q against its own key k) plus K negative logits (q against every key currently in the queue).

Step 4. Cross-entropy with target class 0. The positive logit always sits in column 0, so the $(1 + K)$-way loss is one call to an ordinary cross-entropy.

Step 5. Update f_q by gradient, f_k by EMA. Backpropagate only into f_q; then slide f_k's weights a small step toward f_q's.

Step 6. Refresh the queue. Push this step's keys in, drop the oldest ones out — the dictionary keeps moving without ever being recomputed from scratch.

Line-by-line, mirroring the build slides: two views are drawn (66); the key encoder is run under `detach` (67); the loss compares the query against the positive key and against the whole queue (68–69); the query encoder gets SGD gradients while the key encoder is EMA-updated (70); the FIFO queue is updated (71) so old keys leave and fresh keys enter.

Two lines in the pseudocode matter more than they look.

`k.detach()` — "the queue is data, not parameters". Every key in the queue was produced by the key encoder at some *earlier* step. If gradients were allowed to flow into those keys, backpropagation would need the activations from every one of those old forward passes — thousands of them, under old weights. Those activations were thrown away long ago to save memory, so the gradient is not merely expensive; it is impossible to compute. Detaching the keys says: treat the queue as fixed data, and only train the query encoder.

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k^{(t-\Delta)}} = \sum_{j=1}^{K} \frac{\partial \mathcal{L}}{\partial \boldsymbol{k}_j} \cdot \frac{\partial \boldsymbol{k}_j}{\partial \boldsymbol{\theta}_k^{(t-\Delta)}} \qquad \text{(the gradient we refuse to compute)}$$

`labels = zeros(N)` — the loss is just classification. Each row of logits is laid out as `[positive, negative`$_1$`, ..., negative`$_K$`]`. The positive is always in position 0, so the "correct class" is 0 for every sample, and InfoNCE becomes one ordinary `CrossEntropyLoss` call. Contrastive learning is literally a $(1 + K)$-way classification problem in which the right answer is always "the first one".

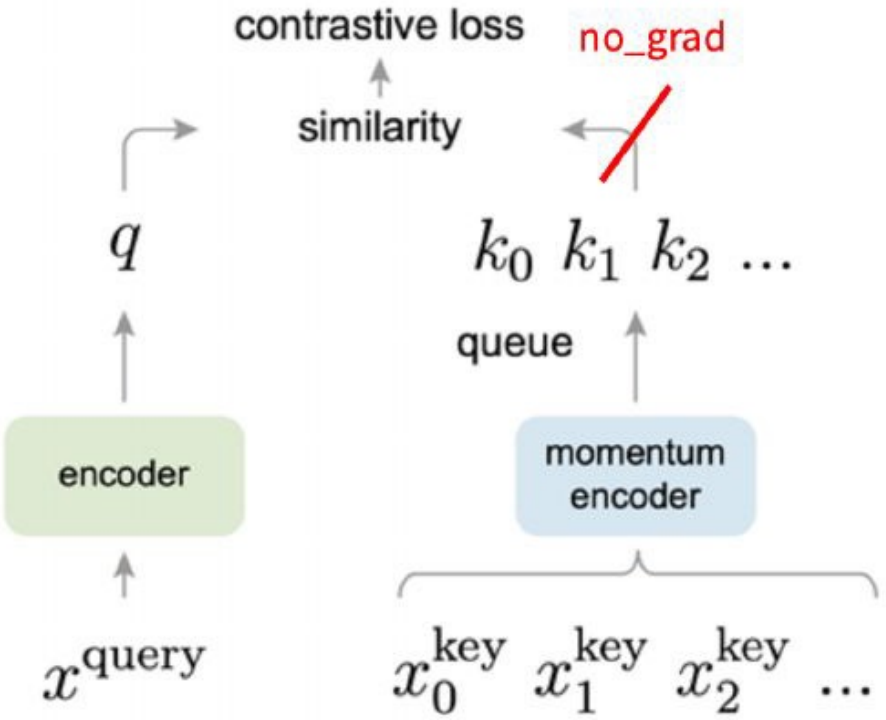


Figure 9.31. MoCo's dictionary-as-queue: the query encoder $\boldsymbol{f}_q$ is trained with gradients; the key encoder $\boldsymbol{f}_k$ is a momentum-EMA copy producing keys stored in a FIFO queue. The InfoNCE loss reads positives from the current key batch and negatives from the queue.

How to read it: the red "no_grad" slash between the momentum encoder and the queue. That single mark is Listing 9.2's `k.detach()` *— the reason the queue is affordable at all (see the discussion just below the listing).*

Source: Lecture 06, slide 64.

Paper Analysis: MoCo (He, Fan, Wu, Xie, Girshick, 2020)

Paper: *Momentum Contrast for Unsupervised Visual Representation Learning* (He et al., 2020).

Problem. Contrastive learning needs many negatives, but large batches are compute-hungry and small-batch memory banks are stale. Can one have many negatives that are also *consistent*?

Prior work. NPID (memory bank), PIRL (memory bank with NCE), end-to-end in-batch negatives.

Insight. Decouple the key encoder from the query encoder via an EMA update, and decouple the negative set size from the batch size via a queue. The EMA update cheaply tracks the query encoder's changing representation without retraining, so the queue's keys are approximately consistent.

Method. Equation (9.7) with $m = 0.999$, queue size $K = 65536$, temperature $\tau = 0.07$; Listing 9.2.

Experiments. Linear-probe ImageNet and PASCAL VOC transfer; PIRL's own Table 2 (slide 43) lists MoCo at 60.6 top-1.

	SimCLR	MoCo
Negatives	In-batch ($2(N-1)$ per query)	Queue ($K = 65536$)
Key/query encoder	Shared	EMA copy ($m = 0.999$)
Compute	TPU pods, batch up to 8192	Single node, batch 256
Temperature	$\tau \approx 0.1$	$\tau = 0.07$
Augmentations	Composed crop + color	Standard crop + color
Memory mechanism	Batch is the dictionary	Queue is the dictionary

Table 9.4. Two canonical solutions to the same problem: consistent-many-negatives. SimCLR pays in hardware; MoCo pays in engineering.

What changed — SimCLR → MoCo

Before. SimCLR's negatives are exactly the rest of the current batch. More negatives means a bigger batch, and a bigger batch means more GPU memory and, at the largest settings, hardware most labs did not have.

What changed. Decouple the negative pool from the batch entirely. Keep a queue of past keys, refreshed by a momentum-averaged copy of the encoder rather than the encoder itself, so the queue stays large *and* close enough to consistent for the softmax to remain meaningful.

What it cost. A second encoder — no extra gradient cost, since it is never backpropagated through, but extra memory to hold its weights and the queue — a new hyperparameter m with its own failure modes at both extremes, and FIFO bookkeeping that SimCLR never needed.

Key takeaway

MoCo answers the exact question an in-batch method like SimCLR raises (the two papers appeared within months of each other, MoCo first): what if the negatives did not have to live in the current batch? The EMA key encoder is the enabling idea — slow enough that a queue of old keys stays trustworthy, fast enough that it still tracks the query encoder's progress. That one mechanism outlives MoCo itself: every EMA teacher in Chapters 10 and 11, including DINO's, is a direct descendant of Equation (9.7).

9.9 Putting the Pieces Together: SimCLR vs. MoCo

The two frameworks take different positions on three axes:

It is tempting to read the table's last two rows as saying SimCLR and MoCo are two different *memory systems* and that the actual encoder training is shared. This is almost, but not quite, true. The two make different choices about *which keys are in the comparison set at all*: SimCLR's negatives are exactly the other samples currently being optimized, so every gradient step immediately freshens every negative; MoCo's negatives are mostly history, so

the freshest entries in the queue are one step old and the oldest are K/B steps old. SimCLR therefore never worries about staleness and pays in batch size; MoCo never worries about batch size and pays in staleness management (the EMA) and bookkeeping (the queue). The two are best read as two points on one axis — how much consistency do you trade for scale? — not as two different algorithms.

A genuinely surprising later finding: both reduce in accuracy if the projection head or the augmentation composition is removed — the details matter more than the loss form. This realization seeds Chapter 10's negative-free methods: if the pipeline rather than the loss carries the semantics, perhaps negatives are not needed at all.

Key takeaway

SimCLR and MoCo are not rival algorithms; they are two answers to one resource question — consistent negatives, cheaply — and both lose accuracy in the same way if their supporting cast (projection head, composed augmentations) is removed. That shared fragility is the clue Chapter 10 follows: if the pipeline around the loss matters this much, maybe the negatives inside the loss are not the essential ingredient at all.

9.10 Why Does Contrastive Learning Work? (Interpretation, Not Settled Theory)

Simplified intuition vs. more precise view

Simple story. InfoNCE maximizes mutual information between two views (Section 9.5); mutual information selects semantic features.

Precise story. InfoNCE provably optimizes a *lower bound* on MI (9.6), but the bound is loose (Poole et al., 2019) and bounded above by $\log(K+1)$. Moreover, MI alone is not a sufficient criterion for semantic features (Tian et al., 2020) — one can create high-MI but semantically useless representations by choosing pathological views. The MI story is *an* interpretation, not a theorem about the downstream utility of learned features.

9.10.1 Empirical picture: alignment and uniformity

A different, measurable lens on the same phenomena decomposes contrastive behaviour on the hypersphere into two complementary quality scores.

Alignment is the expected distance between positive pairs $\mathbb{E}\|f(\boldsymbol{x}) - f(\boldsymbol{x}^+)\|^2$: smaller is better, and InfoNCE's per-pair pressure directly minimizes it.

Uniformity is a proxy for how evenly the embeddings of all samples cover the sphere — operationalized as the log-average pairwise Gaussian potential $\log \mathbb{E}_{x,y}\, e^{-2\|f(\boldsymbol{x})-f(\boldsymbol{y})\|^2}$: more uniform scatter (even coverage of the sphere) gives lower (better) uniformity scores. The empirical finding reported in the literature, consistent with the slides' kNN and linear-probe

results, is that good contrastive learners optimise both — positives are pulled together *and* the cloud of all embeddings spreads on the sphere.

Worked Example — Scoring the three panels of Figure 9.5

Go back to the toy picture at the start of the loss ladder: four images, two views each, on a unit circle. Computing both scores for each panel (alignment averages over the four positive pairs; uniformity averages over all 28 pairs of the eight points; lower is better for both):

panel	alignment ↓	uniformity ↓	verdict
(a) before training	3.75	−2.31	spread out, but twins are far apart
(b) pull only (collapse)	0.00	0.00	twins agree, but so does everyone
(c) pull + push	0.03	−1.92	good on both

Each metric on its own picks the wrong panel. Alignment alone calls the collapsed panel (b) perfect. Uniformity alone slightly prefers the untrained scatter (a), because in (c) each image's two views sit close together, and the uniformity score counts that closeness against it. Only the pair of scores together singles out (c). This is the measurable version of "pull positives together, push everything else apart", and it is why the two are always reported side by side.

Research Frontier

Chapter 15 returns to this debate at the research level. There, InfoNCE bounds (van den Oord et al., 2018b; Poole et al., 2019), augmentation-graph spectral arguments, and collapse analyses compete as explanations for contrastive behaviour. The honest position of this book: no single one is currently settled as *the* theory of why contrastive learning works.

9.10.2 Comparison to the previous paradigm

Chapter 8's pretext tasks (RotNet, jigsaw, colorization) train by maximizing the *predictive information about the transformation*: the network is rewarded for encoding the rotation angle. Contrastive learning trains by maximizing the *shared information across views*: the network is rewarded for *ignoring* rotation. PIRL was the first method to philosophically sit between the two: it *uses* a pretext transformation but *discards* its predictive target. Empirically, on semantic transfer benchmarks and at moderate compute, invariance wins — hence the rest of Part II.

Key takeaway

Take the mutual-information story as a useful mental model, not a proof of why contrastive learning transfers well: it correctly predicts some effects (more negatives helps, up to a point) and is silent or wrong about others (which information matters, not just how

much). The alignment/uniformity lens is a second, complementary measurement — pull positives together, spread everything else out — that describes what a good contrastive space looks like without claiming to explain why it generalizes. Chapter 15 revisits the open question.

9.11 Chapter summary

Why. Two augmented views of one image are, almost by definition, the same underlying thing seen twice. Training an encoder to recognize that — against a crowd of unrelated distractors — needs no human label at all, and if the crowd is large and varied enough, the only reliable way to win the game is to encode real semantic structure.

How. Every method in this chapter shares one picture (Figure 9.4): two branches, a shared encoder, a loss on the pair. What changes, rung by rung, is how many negatives that loss can see and where they come from. The loss ladder (Section 9.3) turns a single hinge on one negative into a $(K+1)$-way softmax over many; SimCLR affords many negatives with a huge batch, MoCo with a queue and a momentum encoder. A temperature (Section 9.4) decides how much of that softmax's gradient any one negative receives.

How you know. Linear-probe and kNN accuracy on frozen features, exactly as in Chapter 8 — the training loss itself is not evidence, since an encoder that matches views through a low-level shortcut can reach a small, well-behaved InfoNCE value while learning little that transfers, and class collision (Section 9.6) is invisible in the loss.

One sentence worth keeping past the equations: contrastive learning replaces a human's "these are the same class" with an augmentation pipeline's "these are the same photo," and the entire chapter is the story of what that substitution costs (false negatives) and how far compute can push back against the cost (bigger batches, queues, better temperatures).

Key takeaway

If this chapter collapses to one idea, it is this: a contrastive loss is a classification problem in disguise, and everything interesting about it — temperature, batch size, queues, false negatives — is a fact about how that disguised classifier is built, not about some separate "contrastive learning" principle on its own. Chapter 10 asks what happens when the classifier's negatives are removed altogether.

Method	Year	What it fixed, and what it cost
Siamese network	1993	Fixed: gave verification a trainable, symmetric distance. Cost: needs a human-labeled pair for every example.
Triplet loss (FaceNet)	2015	Fixed: replaced an absolute distance threshold with a relative margin. Cost: one negative per update, mostly wasted on already-satisfied triplets.
N-pair loss	2016	Fixed: all negatives in one softmax instead of one hinge. Cost: fixed, unscaled temperature; more compute per step.
InfoNCE / CPC	2018	Fixed: explicit temperature, a $(K+1)$-way classification framing, and a mutual-information interpretation. Cost: saturates at $\log(K+1)$; blind to false negatives.
SimCLR	2020	Fixed: showed composed augmentations + a disposable projection head + in-batch negatives beat elaborate predecessors. Cost: needs a very large batch (up to 8192) for enough negatives.
MoCo	2020	Fixed: decoupled negative count from batch size with a queue and a momentum encoder. Cost: a second encoder, a new momentum hyperparameter, queue bookkeeping.

Table 9.5. The lineage of this chapter in one table. Every row fixes a specific weakness of the row above it, and every fix has a price — Table 9.2 and Table 9.4 give the mechanism-level detail behind each row.

Term	In one line
Positive pair	Two views manufactured from the same underlying sample.
Negative pair	Two views the pipeline treats as unrelated.
Anchor	The reference view a triplet or InfoNCE loss is written from.
Margin α	The minimum required gap between positive and negative distances in a hinge loss.
Temperature τ	The softmax's sharpness dial; small τ concentrates gradient on the hardest negative.
Projection head	A disposable network on top of the representation, trained to absorb the loss's invariance pressure so the representation itself need not.
Memory bank	A dictionary of previously computed features, reused as negatives; updated slowly, so it goes stale.
Momentum encoder	An encoder whose weights are an exponential moving average of another encoder's, never updated by gradient directly.
Queue	A fixed-size, first-in-first-out buffer of past keys, MoCo's negative source.
False negative / class collision	A same-class pair the loss mistakenly treats as a negative, because it has no access to labels.
Alignment / uniformity	Two measurable properties of a good contrastive space: positives close together, the whole cloud spread evenly.

Table 9.6. Vocabulary introduced in this chapter.

9.12 Exercises

Check Your Understanding

1. **(Level 1)** Give the defining property of a positive pair in contrastive learning.
2. **(Level 1)** In one sentence, what does the temperature τ control in InfoNCE?
3. **(Level 2)** Why does the triplet loss need a margin α? What can go wrong without one?
4. **(Level 2)** Name two differences between N-pair and InfoNCE losses.
5. **(Level 2)** What is a memory bank, and what problem does it solve for PIRL?
6. **(Level 2)** Why does SimCLR "throw away" the projection head g at transfer time?
7. **(Level 3)** Write the MoCo EMA update rule for the key encoder and explain what a value of m *too far from 1* would do.
8. **(Level 3)** Explain class collision and name one mitigation.
9. **(Level 3)** In SimCLR, the batch is $\{x_1, \ldots, x_B\}$, each seen under two augmentations. How many negative logits does one InfoNCE denominator contain?
10. **(Level 3)** Slide 51 places a second cat among the negatives of a cat query. Trace what the InfoNCE gradient does to that negative's embedding over the course of training, and explain why the harm grows as training progresses rather than shrinking.
11. **(Level 3)** In Listing 9.2, why is the target label for every row of the cross-entropy `0`? What structural choice in the two preceding lines makes that correct?
12. **(Level 4)** Step through the proof of $I \geq \log(K+1) - \mathcal{L}_{\text{InfoNCE}}$: what are the three main moves? Where does the critic enter?
13. **(Level 4)** Why can a very small τ hurt training even though it sharpens the softmax? Refer to the scalar example in Section 9.4.
14. **(Level 4)** Compare PIRL's memory bank with MoCo's queue: which mechanism controls key staleness, and how?
15. **(Level 5)** The bound (9.6) suggests "larger K is better." Give two reasons this is a heuristic and not a law.
16. **(Level 5)** Sketch how you would detect shortcut learning in a contrastive pipeline. What kind of augmentation family would defend against it?

Solution sketches

- **1.** Two views of the *same* underlying sample — the pairing comes from the data augmentation pipeline, not from labels.
- **2.** It rescales logits on the hypersphere before the softmax: small τ sharpens, large τ flattens.
- **3.** The margin converts the triplet ordering into a constraint with a real gap. Without it ($\alpha = 0$), collapsing every embedding to one point gives zero loss, so nothing stops the trivial solution.

- **4.** E.g., N-pair operates in inner-product space with $\tau = 1$ fixed; InfoNCE is normalized by temperature and is the canonical K-way cross-entropy interpretation tied to MI bounds.
- **5.** A dictionary of previously computed features reused as negatives, so the network does not have to re-encode the whole dataset every step.
- **6.** The loss acts on $\boldsymbol{z} = g(\boldsymbol{h})$, training $\boldsymbol{z}$ to be invariant to transformations; $\boldsymbol{h}$ retains information g discards and is the transferable representation.
- **7.** $\boldsymbol{\theta}_k \leftarrow m\boldsymbol{\theta}_k + (1-m)\boldsymbol{\theta}_q$; m far below 1 (say 0.9) makes the key encoder track the fast-moving query encoder, so queue keys become inconsistent and contrastive discrimination degrades.
- **8.** Two in-batch samples share a semantic class but are treated as negatives for each other; mitigations: moderate τ, larger batch / queue (more true negatives), class-aware sampling.
- **9.** $2(B-1)$ negatives: all $2B$ representations minus the query and its positive.
- **10.** Early on, the two cats are fairly similar (both encode "cat-like" features), so the false negative is a *hard* negative — high s_j, hence high softmax weight $P(j)$ in the gradient $\partial\mathcal{L}/\partial\ell_j = P(j)$. Training therefore pushes it away with nearly the strongest repulsion in the batch. As the encoder improves at separating them, the false negative remains among the most similar negatives (they still share class semantics), so it stays near the top of the hardness ranking and keeps attracting gradient; the push-apart pressure does not decay to zero the way it does for genuinely easy negatives.
- **11.** `cat([l_pos, l_neg], dim=1)` places the positive logit in column 0 of every row; `labels = zeros(N)` then declares class 0 correct for every query. The $(1+K)$-way InfoNCE is exactly `CrossEntropyLoss`$(\text{logits}/\tau, \mathbf{0})$.
- **12.** (i) Identify the optimal critic as a log density ratio of positive-pair conditional to noise marginal; (ii) substitute into the loss and split off the chance-level term $\log(K+1)$ and the expected log-ratio; (iii) recognize the remaining expectation as $-I(\boldsymbol{x}; \boldsymbol{z}^+)$ and re-arrange.
- **13.** With $\tau = 0.1$, a similarity difference of 0.4 becomes a logit difference of 4, so one negative dominates the gradient; others receive almost no push. Medium τ spreads gradients across more negatives.
- **14.** PIRL's memory bank is updated only occasionally (a stored feature can be up to an epoch old); MoCo's queue keys are only a few batches old and encoded by an EMA of the query encoder, bounding the drift.
- **15.** The bound is loose (Poole et al., 2019); MI is not per-se semantic (Tian et al., 2020); large K can also hurt via class collision.
- **16.** Track a probe of an augmentation-nuisance variable (e.g. dominant color); if a linear probe predicts it perfectly the network may shortcut via color. Defense: compose augmentations that destroy the shortcut.

Concept Map — how this chapter connects

Backward. The representations of this chapter inherit Chapter 8's augmentation toolkit (crop, jitter, rotation) and the invariance-vs-covariance debate (PIRL begins where RotNet ended); the evaluation protocols from Chapter 8 (linear probe, VOC07 transfer) are exactly the metrics used here. The kNN and clustering tools of Chapter 3 are how slide 43's tables are read. Mutual information and softmax are the Chapter 2 prerequisites the MI-bound section builds on.

Sideways. The loss ladder of Section 9.3 is shared with deep metric learning; CPC (van den Oord et al., 2018b) is the cross-modal, sequence-side ancestor of everything here.

Forward. Chapter 10 asks whether negatives are needed at all (SimSiam, BYOL, Barlow Twins — teased on slide 72 of Lecture 06) and how collapse is avoided without them. Chapter 11 shows the MoCo-style EMA teacher reappearing inside DINO and its descendants; the data-engine discussion there explains how contrastive recipes scale to billions of images. The closing interpretation questions (MI bounds vs. alignment/uniformity) are taken up at research depth in Chapter 15.

CHAPTER 10

Beyond Negatives: the Collapse Problem and Non-Contrastive Learning

"SimSiam can learn meaningful representations with none of the following: (i) negative sample pairs, (ii) large batches, (iii) momentum encoders."
— Chen & He, *Exploring Simple Siamese Representation Learning*, CVPR 2021

In this chapter

10.1 The collapse problem, stated precisely .. 482 • A theorem in one line • How InfoNCE avoids collapse by construction • Diagnosing collapse in the field
10.2 Three families of collapse avoidance 486
10.3 SimSiam: stop-gradient plus a predictor, and nothing else ... 487 • Setup and the lecture's collapse curves • The collapse experiment, read closely • Architecture and the symmetric loss • Why stop-gradient plus predictor works: the verified story and the interpretation
10.4 BYOL: SimSiam plus EMA — and still no negatives 498
10.5 Barlow Twins: redundancy reduction on the cross-correlation matrix 500 • From identity cross-correlation to a loss
10.6 VICReg: variance, invariance, covariance — an explicit triad 506
10.7 Why does any of this work? Two sufficient conditions 508
10.8 Head-to-head comparison 508
10.9 Research frontier: collapse as a black box 509
10.10 Summary 511

Chapter roadmap

Before you start. You need three things from earlier chapters. From Chapter 9: the joint-embedding setup (two augmented views of one image, one encoder, a loss on the two embeddings) and the InfoNCE loss with its negatives. From Chapter 8: what an

augmentation family is and why we ask features to ignore it. From Chapter 2: cosine similarity, variance and covariance, and what an exponential moving average does. If any of those is hazy, read Section 9.3.4 and Chapter 2's statistics section first.

The idea in one sentence. If you train two branches only to *agree*, the cheapest way to agree is to output the same vector for every image, so every negative-free method needs some other force that makes "always answer the same thing" a bad answer.

What we cover.

- *Collapse*, named and drawn (Section 10.1). We show a collapsed embedding space next to a healthy one and put the loss value on both, so you can see why the loss is delighted by the broken model.
- *The three known counterforces* (Section 10.2): push different images apart (negatives), make the two branches structurally different (asymmetry), or add a term that measures collapse directly (redundancy reduction). Every method in this chapter is one of these three.
- *SimSiam* (Section 10.3): the smallest recipe that works. One shared encoder, one predictor, one `detach`.
- *BYOL* (Section 10.4): the same thing with a slow copy of the network as the target.
- *Barlow Twins* and *VICReg* (Sections 10.5–10.6): stop being clever about the architecture and just write the anti-collapse condition into the loss as statistics you can print out.
- *SwAV* (Section 10.2): the cluster-flavoured answer, which keeps a form of repulsion but between *groups* instead of between images.
- A comparison table (Section 10.8) that says, for each method, what breaks if you remove its key part.

Afterwards you will be able to (i) recognize collapse from a training log in under a minute and name the three symptoms; (ii) say which anti-collapse mechanism any given joint-embedding method uses, and what it costs; (iii) compute a Barlow Twins cross-correlation matrix and its loss by hand; and (iv) pick a negative-free method for a given batch-size and code-base budget and defend the choice.

Chapter 9 built joint-embedding learning on a specific pair of forces: an *attraction* term that pulls the embeddings of two augmented views of one image together, and a *repulsion* term that pushes embeddings of different images apart. InfoNCE implements repulsion with in-batch negatives; MoCo with a queue of momentum-encoded keys. This machinery is effective but heavy: large batches, memory banks, queues, and careful temperature tuning all exist essentially to source enough negatives. In 2020–2022 a series of papers asked a disarmingly simple question: *can we delete repulsion entirely?* The answer turned out to be yes — but only once we understand precisely what repulsion was protecting us from. This chapter names that enemy (*collapse*, Section 10.1), develops the three families of collapse-avoidance mechanisms (negatives; asymmetric architectures with stop-gradient; explicit redundancy-reduction terms),

and works through one representative of each family in enough detail to re-implement it: SimSiam (Chen and He, 2021) with its sibling BYOL (Grill et al., 2020), then Barlow Twins (Zbontar et al., 2021) and VICReg (Bardes et al., 2022).

10.1 The collapse problem, stated precisely

We are in the joint-embedding setting of Chapter 9. An augmentation family $\mathcal{T}$ (random crops, flips, color jitter, as catalogued in Chapter 8) produces two *views* $\boldsymbol{x}_1 = t_1(\boldsymbol{x})$ and $\boldsymbol{x}_2 = t_2(\boldsymbol{x})$ of the same training image $\boldsymbol{x}$, with $t_1, t_2 \sim \mathcal{T}$. An encoder $f_{\boldsymbol{\theta}}$ maps each view to an embedding $\boldsymbol{z}_i = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \in \mathbb{R}^{d_z}$. The shared attraction goal of every method in this chapter is *invariance*: agree across views,

$$\min_{\boldsymbol{\theta}} \ \mathbb{E}_{\boldsymbol{x}\sim p,\, t_1,t_2\sim\mathcal{T}}\big[\, D(\boldsymbol{z}_1, \boldsymbol{z}_2)\,\big], \tag{10.1}$$

where the expectation is over the data distribution p and the augmentations, and D is a discrepancy between embeddings (negative cosine similarity in SimSiam, squared Euclidean error in Barlow Twins' diagonal term, cross-entropy in DINO, and so on).

10.1.1 A theorem in one line

Suppose D measures only pairwise alignment — it is a function of the two embeddings that is minimized whenever they agree (all of the above qualify). Then there exists a trivial global minimizer: any *constant function* $f_{\boldsymbol{\theta}^\star}(\boldsymbol{x}) = \boldsymbol{c}$ for all inputs $\boldsymbol{x}$, where $\boldsymbol{c} \in \mathbb{R}^{d_z}$ is an arbitrary fixed vector. Every pair of views embeds to the same point, the discrepancy is at its optimum, and (10.1) is minimized — while the representation carries *zero information about the input.*

Key Concept — Collapse

Collapse is the failure mode of joint-embedding learning in which the encoder maps (nearly) every input to the same embedding. The training loss reaches its theoretical optimum — attraction is perfect — yet the representation has constant output, hence zero mutual information with the data: $I(X; Z) = 0$. **Complete collapse** means literally constant output; **dimensional collapse** means the outputs occupy a low-dimensional subspace of $\mathbb{R}^{d_z}$ even though they are not constant.

In plain words

Ask a network only "make your two answers match" and it will find the laziest way to obey: answer the same thing every time. Two photographs of a zebra match perfectly if the network replies "7" to both. So do two photographs of a teapot. The loss is looking for agreement, it got perfect agreement, and it has no way of noticing that the answers stopped depending on the pictures. That is collapse, and it is not a bug in the code — it is the exact solution the loss asked for.

New words used from here on

Embedding space. The d_z-dimensional space $\mathbb{R}^{d_z}$ that the encoder's outputs live in. Drawing it as a flat page, as in Figure 10.1, is a two-dimensional cartoon of something usually 128- to 8192-dimensional. **Constant function.** A network whose output does not change when the input changes. **Loss floor.** The smallest value a loss can ever take. For negative cosine similarity that is -1; for cross-entropy it is 0. A run that reaches the floor has not necessarily succeeded — it has only found *some* minimizer, and this chapter is about the useless ones.

Figure 10.1 is the picture the rest of this chapter exists to explain. It puts a collapsed encoder next to a healthy one and writes the training loss under each. The collapsed one scores better.

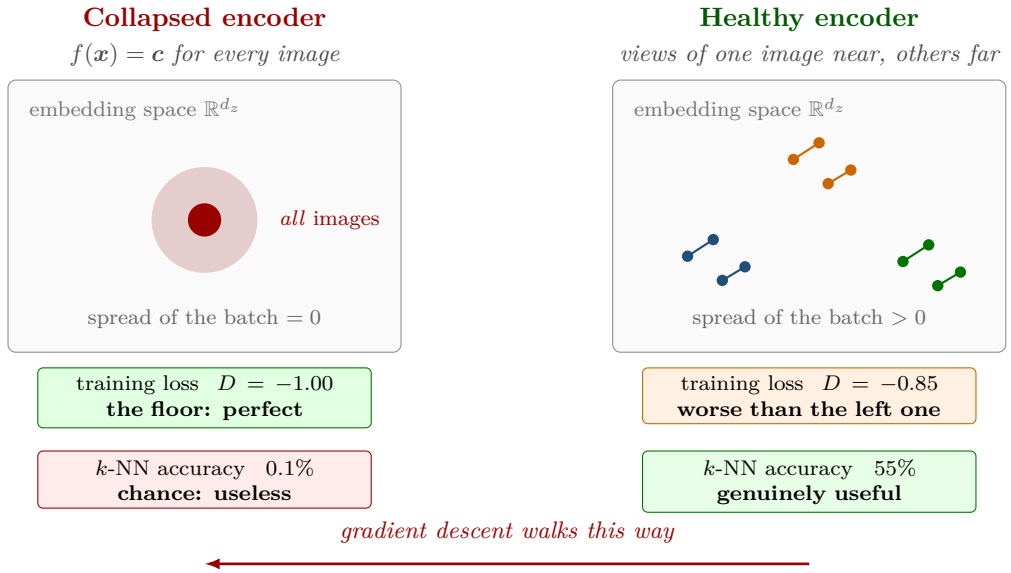


Figure 10.1. Collapse, drawn. *Left:* a collapsed encoder sends every image to the same point $\boldsymbol{c}$; the two views of an image are on top of each other, so their negative cosine similarity is exactly -1, the best value the loss can ever take. *Right:* a healthy encoder keeps the two views of one image close but different images apart; because two independent crops of a real photograph never match perfectly, its loss *cannot* reach -1 and plateaus around -0.85. The loss column therefore ranks the useless model first. Only the k-NN column, which the loss never sees, ranks them correctly. (Values are the ones the lecture's collapse experiment reports for SimSiam on ImageNet-1k at 100 epochs; see Figure 10.3. Chance on ImageNet-1k is 0.1%.)

How to read it: Compare the two green boxes before anything else. The model on the left wins on the number you are optimizing and loses on the number you care about. Everything else in this chapter is machinery for deleting the left-hand model from the search space.

Why collapse is so easy to fall into.

- **It is simple.** The network has millions of parameters; a constant output needs only a handful of numbers to describe.
- **It is a wide, flat valley.** Gradient descent is not *looking* for it, but gradients flow into it whenever nothing in the loss pushes back.
- **Shrinking looks like progress.** Pulling all embeddings towards one common point reduces the pairwise differences just as surely as learning good features would. Grill et al. (2020) documented this "shrinking cone" effect for BYOL.

10.1.2 How InfoNCE avoids collapse by construction

Recall the InfoNCE loss from Chapter 9, written for a query $\boldsymbol{x}$ with positive view $\boldsymbol{x}^+$ and K negatives $\boldsymbol{x}_k^-$:

$$\mathcal{L}_{\text{NCE}} = -\log \frac{\exp\big(\operatorname{sim}(\boldsymbol{z}, \boldsymbol{z}^+)/\tau\big)}{\exp\big(\operatorname{sim}(\boldsymbol{z}, \boldsymbol{z}^+)/\tau\big) + \sum_{k=1}^{K} \exp\big(\operatorname{sim}(\boldsymbol{z}, \boldsymbol{z}_k^-)/\tau\big)}, \tag{10.2}$$

where $\operatorname{sim}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^\top \boldsymbol{v} / (\|\boldsymbol{u}\| \, \|\boldsymbol{v}\|)$ is cosine similarity and $\tau > 0$ the temperature.

What happens under collapse? If every vector is the same, $\boldsymbol{z} = \boldsymbol{z}^+ = \boldsymbol{z}_k^- = \boldsymbol{c}$, then every similarity is 1, every logit is $1/\tau$, and the softmax in (10.2) is uniform over the $K{+}1$ candidates. The loss is $\mathcal{L}_{\text{NCE}} = \log(K+1)$ — no better than guessing at random. Collapse is *penalised*, not rewarded: the negatives in the denominator are as similar as they can possibly be. Near collapse, the gradient is dominated by the repulsion terms, which push different samples apart.

So contrastive learning is *alignment plus built-in repulsion*, and it cannot collapse completely (it can still suffer the milder dimensional collapse). The price is Chapter 9's engineering burden. Negatives must be *many* (SimCLR's batches of thousands; the mutual-information bound is capped at $\log(K{+}1)$, which Chapter 9 writes with pool size $N = K + 1$) and *well-spread* (false negatives, memory consistency).

It also matters *what* collapse means informationally. If Z is constant, then $I(X; Z) = 0$: a downstream linear probe can do no better than majority guessing, and a k-NN classifier in Z finds every training point at zero distance, so it too can only guess. Collapse is not a subtle degradation of features; it is the total loss of the representation's reason to exist.

Common Confusion

"Low training loss = good representation." True in supervised learning, false here. Slide 13 of Lecture 07 states it bluntly: "the loss does not directly model the downstream task." Under collapse the SimSiam training loss dives to its optimum (-1) within a few epochs while the probe accuracy stays at chance. Training loss in joint-embedding SSL measures only how well the network satisfies (10.1), which a constant function satisfies perfectly. Always watch a downstream signal — k-NN accuracy on frozen features is the cheap, low-variance choice that Chapter 11 discusses — alongside the training loss.

10.1.3 Diagnosing collapse in the field

Collapse is easy to *define* and surprisingly easy to *miss* in a running job: the training loss keeps improving throughout, so none of the usual supervision cues fire. The lecture drills a symptom triad that is worth committing to memory, because it converts collapse from a pathology you read about into one you can detect at the console (Lecture 07, slides 13–14 and the diagnosis exercises attached to them):

1. **Training loss at its floor.** For SimSiam-style negative-cosine losses, convergence to -1 within the first few epochs. Nothing in natural data is *that* easy to align; a floor-level loss early in training is suspicious, not encouraging.
2. **Probe accuracy pinned at chance.** k-NN on frozen features flat at the random-guess rate for the whole run, no matter how the loss moves. The loss and the probe departing in opposite directions is the single most reliable collapse signal.
3. **Zero-spread embeddings.** A PCA (or simply a per-dimension standard deviation) of a held-out batch of embeddings collapses to a point, or to a needle along one axis in the dimensional-collapse variant. A cheap online statistic suffices: track Std of each dimension of the normalized embeddings across the batch; if the typical value approaches zero, collapse is underway.

When the three warning signs appear, check in this order.

1. **Is the stop-gradient really detaching the target branch?** One missing `detach` silently turns SimSiam into a symmetric, attraction-only network that collapses within a few iterations.
2. **Is the predictor really on one side only?** It must sit on one side of each loss term, not be accidentally applied to both.
3. **Are the two views still different?** Weaker augmentation does not make training safer — it makes the shortcut easier, because the branches can agree without looking at the input.

Notice that none of these is hyperparameter tuning. Collapse bugs in this family are almost always *wiring* bugs.

Key takeaway

> A loss that only rewards agreement has a perfect, useless solution: output the same vector for every input. It is not rare, it is not subtle, and the training loss will never warn you about it, because reaching that solution *is* minimizing the loss. Watch a downstream number (k-NN on frozen features) and the per-dimension spread of the embeddings, never the loss alone.

10.2 Three families of collapse avoidance

Once collapse is understood as "attraction has no counterforce," the design space organizes itself by *where the counterforce comes from.* Three families cover essentially all successful joint-embedding methods.

(a) Contrastive repulsion with negatives. Extend the loss with a term that is explicitly maximized by collapse and minimized by spreading different samples apart. SimCLR (Chen et al., 2020a) does this with in-batch negatives, MoCo (He et al., 2020) with a queue, PIRL (Misra and van der Maaten, 2020) with a memory bank, and SwAV (Caron et al., 2020) with contrasting cluster assignments. This family was Chapter 9; here we only carry the observation that repulsion is *one* working answer, not the only one.

(b) Asymmetric architectures with decoupled gradient paths. Keep the attraction-only loss of (10.1) but break the symmetry that lets the encoder slide into the trivial basin: predict one branch's output from the other through a small extra network, and forbid gradients to flow back through the target branch (stop-gradient), optionally making the target branch a slowly-updated exponential moving average (EMA) of the source branch. BYOL (Grill et al., 2020) uses predictor *and* EMA; SimSiam (Chen and He, 2021) strips the EMA away and keeps predictor *plus* stop-gradient; DINO (Caron et al., 2021) (Chapter 11) uses EMA and stop-gradient with centering and sharpening instead of a predictor.

(c) Explicit decorrelation or expansion terms. Add a term to the loss that directly encodes what collapse violates — non-degenerate second-order statistics of the embedding batch: unit variance per dimension (VICReg (Bardes et al., 2022)), decorrelated dimensions converging to an identity cross-correlation matrix (Barlow Twins (Zbontar et al., 2021)), whitened embeddings, or uniform spread on the sphere (the KoLeo nearest-neighbour-distance term of DINOv2, Chapter 11, which itself descends from the "spreading vectors" regularizer of Sablayrolles et al. (2019)). Here collapse is not avoided implicitly; it is *defined away*, term by term.

Figure 10.2 draws all three counterforces in one picture, because the rest of the chapter is bookkeeping on top of it. Read it as three different answers to one question: *what stops the encoder from shrinking everything to a point?*

The rest of the chapter develops (b) through SimSiam and BYOL, (c) through Barlow Twins and VICReg, and the cluster-flavoured corner of (a) through SwAV. It works a small numerical example for the Barlow Twins cross-correlation objective, and closes by comparing all the methods on their mechanics rather than their marketing.

Key takeaway

There is no method that avoids collapse by accident. Every working joint-embedding recipe contains exactly one of: explicit repulsion, architectural asymmetry, or an explicit

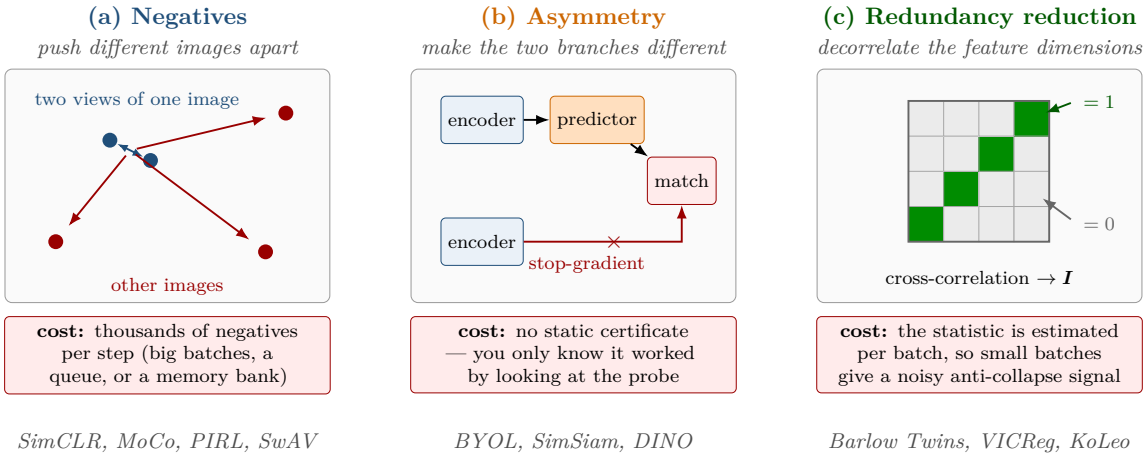


Figure 10.2. The three known ways to stop a joint-embedding encoder from collapsing. *(a)* Add a term to the loss that grows when different images sit close, so the constant solution becomes the *worst* solution instead of the best. *(b)* Keep the agreement-only loss, but make one branch structurally unlike the other — a predictor on one side, a stop-gradient on the other — so the two sides cannot simply drift into each other. *(c)* Measure collapse directly as a statistic of the embedding batch (are the dimensions correlated? do they have any spread?) and add that measurement to the loss.

How to read it: Look at the bottom row of each panel first: that is the bill. Negatives cost batch size, asymmetry costs diagnosability, and redundancy reduction costs a reliable batch-level estimate. There is no free option — the chapter is about which bill you would rather pay.

statistical anti-collapse term. When you meet a new method, your first question should be "which of the three is this, and where is it in the code?"

10.3 SimSiam: stop-gradient plus a predictor, and nothing else

What is it? SimSiam is the smallest joint-embedding method that works. One encoder, used twice. One small extra network on one side, called the *predictor*. One line of code that cuts the gradient on the other side. No negatives, no queue, no second copy of the weights, no batch of thousands. Two views of an image go in, and the training signal is simply "make the predicted vector point the same way as the other branch's vector."

Why was it introduced? Because everyone believed the wrong thing. BYOL had already shown that negatives could be deleted, but it kept a slow, averaged second copy of the network (a *momentum encoder*), and the community read that copy as the thing holding training up. Removing it in early ablations lost accuracy, which seemed to confirm the reading. SimSiam's experiment removed it properly — same predictor, same stop-gradient, no momentum encoder, batch size 256 on one machine — and training stayed healthy. That demoted the momentum encoder from *necessity* to *optimization helper* and left exactly two ingredients standing.

Model at a glance: SimSiam Chen & He, 2021

In one line	A Siamese network trained only to agree with itself, kept from collapsing by a predictor on one branch and a stop-gradient on the other.
Problem it solves	Contrastive learning needs thousands of negatives per step. Remove them and the encoder collapses. SimSiam removes them and does not collapse.
Key intuition	Never let the target move toward the prediction. Freeze it, and make the other side *predict* it rather than copy it.
Inputs	Two independently augmented views $\boldsymbol{x}_1, \boldsymbol{x}_2$ of the same image (random crop, flip, colour jitter, blur — Chapter 8's standard recipe).
Architecture	One shared encoder f = backbone (e.g. ResNet-50) + projection MLP. One shared predictor h = a smaller MLP, applied on one branch per loss term.
Loss	Symmetrized negative cosine similarity, (10.4), with sg on whichever branch supplies the target.
Training	Plain SGD, batch size 256, 100–800 ImageNet epochs. The predictor keeps a fixed (non-decayed) learning rate; everything else uses the usual cosine schedule.
What you keep	The backbone. The projection MLP and the predictor are thrown away after pretraining.
Strengths	Simplest code in the chapter; no negatives, no queue, no EMA; works at small batch size on one machine.
Main failure modes	Delete the `detach` and it collapses in a few iterations. Delete or weaken the predictor and it near-collapses too. Nothing in the loss warns you.
Leads to	BYOL read backwards (Section 10.4) and, with a distributional target instead of a vector target, DINO (Chapter 11).

In plain words

Two copies of the same network look at two crops of one photo. One copy's answer is put through a small extra network that has to *guess* what the other copy said. The other copy's answer is frozen: it is used as the target, but it is not allowed to move toward the guess. Because the target does not budge, the pair cannot meet in the middle at some meaningless constant, and the only way to guess well is to actually encode what is in the photo.

10.3.1 Setup and the lecture's collapse curves

SimSiam (Chen and He, 2021) is a Siamese (twin-branch) network, in the line this book traced back to Becker and Hinton (1992) and Bromley et al. (1993). It adds just two things: an extra *predictor* head on one side, and a stop-gradient on the other.

In 2021 this was a real surprise. BYOL had just shown that negatives could be dropped, but almost everyone credited its *momentum encoder* — the slowly drifting EMA teacher looked like the part holding everything up, and removing it seemed to cost accuracy. SimSiam deleted the EMA too, trained at batch size 256 on a single machine, and still worked. The

momentum encoder went from *necessary* to *helpful*, and the minimal anti-collapse recipe shrank to two asymmetric parts.

Concretely: the shared encoder f is a backbone (e.g. ResNet-50) plus a projection MLP; the predictor h is a second, smaller MLP. Given a batch of images $\boldsymbol{x}$, sample two views per image and compute

$$\boldsymbol{x}_1, \boldsymbol{x}_2 = \mathrm{aug}(\boldsymbol{x}), \mathrm{aug}(\boldsymbol{x}), \quad \boldsymbol{z}_i = f(\boldsymbol{x}_i), \quad \boldsymbol{p}_i = h(\boldsymbol{z}_i), \quad i \in \{1, 2\}, \tag{10.3}$$

where $\boldsymbol{z}_i \in \mathbb{R}^{d_z}$ are the projections and $\boldsymbol{p}_i$ the predictions. The objective matches each prediction to the *other* branch's projection with negative cosine similarity:

$$\mathcal{L} = \tfrac{1}{2}\, D(\boldsymbol{p}_1, \mathrm{sg}(\boldsymbol{z}_2)) + \tfrac{1}{2}\, D(\boldsymbol{p}_2, \mathrm{sg}(\boldsymbol{z}_1)), \qquad D(\boldsymbol{p}, \boldsymbol{z}) = -\frac{\boldsymbol{p}}{\|\boldsymbol{p}\|_2} \cdot \frac{\boldsymbol{z}}{\|\boldsymbol{z}\|_2}, \tag{10.4}$$

where $\mathrm{sg}(\cdot)$ is the stop-gradient operator (identity in the forward pass, zero gradient in the backward pass; notation table). The loss ranges over $[-1, 1]$; -1 is maximal view agreement. Listing 10.1 shows the full training loop in the PyTorch-like form the lecture walks through line by line (slides 15–17).

10.3.2 The collapse experiment, read closely

The natural baseline is the same architecture with sg deleted (and, in the extreme, without h): a symmetric attraction-only loss. Before SimSiam, a reasonable bet was that such a network must collapse and that asymmetric tricks would only slow the failure. The paper ran the experiment; the lecture makes it the visual centerpiece of the chapter (Lecture 07, slides 13–14), and it deserves a close reading, because both of its curves carry signposts that recur throughout the book.

Figure 10.3 redraws the two panels: training loss and frozen-feature k-NN accuracy over 100 ImageNet epochs, with and without stop-gradient. Read the *orange* traces (no stop-gradient) first.

- **The loss crashes to** -1.0 — the lowest value the negative-cosine loss can take — within a few epochs. That speed is the clue. Matching two augmented views of a natural image is hard; a loss that "solves" it almost at once has not solved it at all. The encoder has found the constant function $f(\boldsymbol{x}) = \boldsymbol{c}$, for which every pair of views has cosine similarity exactly 1.
- **The k-NN accuracy stays flat at chance** from the first epoch to the last, confirming it.

Together the two orange curves show the trap: the number you optimise reports total success while the number you care about reports total failure — for a hundred epochs, with no warning in between.

Now the *blue* traces (stop-gradient in place). Two things to see:

- **The loss falls slowly — and never reaches -1.** It drifts to about -0.85 by epoch 100 and is still clearly above the floor. This is the detail newcomers miss: a *healthy* run stops short of perfect agreement. Matching two different crops, colour jitters and blurs of one image is genuinely hard, and the gap between the plateau and the floor is, quite literally, the price of the representation still carrying information. (BYOL and DINO show the same pattern.)
- **The k-NN accuracy climbs steadily,** passing 50% as semantic structure forms.

Hold the two pairs next to each other and the chapter's thesis writes itself: the fastest way to minimize the loss is to collapse, and *only* the stop-gradient stands between gradient descent and that shortcut. The experiment is why the SimSiam paper's authors describe stop-gradient as the pivotal operation of the whole method, and why every subsequent negative-free method has been evaluated, at least in part, by asking "where is your Figure 10.3?"

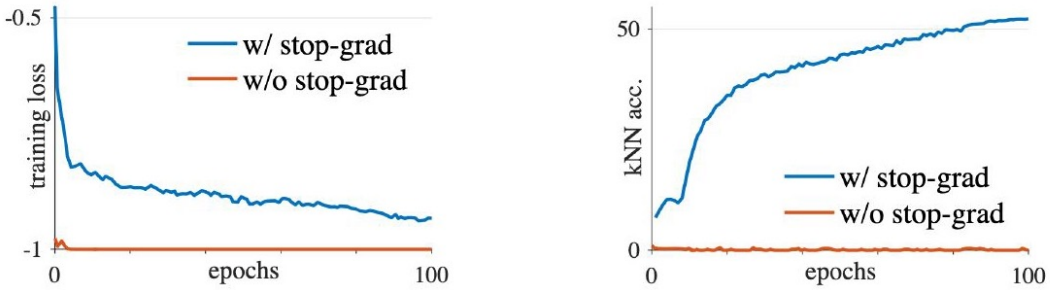


Figure 10.3. The collapse experiment as the lecture shows it (source experiment: Chen and He, 2021). *Left:* training loss of (10.4) over 100 ImageNet epochs, with (blue) and without (orange) stop-gradient. Without stop-gradient the loss drops to -1 in the first epoch and stays pinned there — a collapsed constant solution. *Right:* k-NN accuracy of the frozen features. With stop-gradient it climbs past 50%; without, it is flat at chance for the whole run. Low training loss does not imply good features.

How to read it: Cover the right panel with your hand. Nothing in the left panel tells you which run is broken — the orange one simply looks like it converged faster. That is the entire problem.

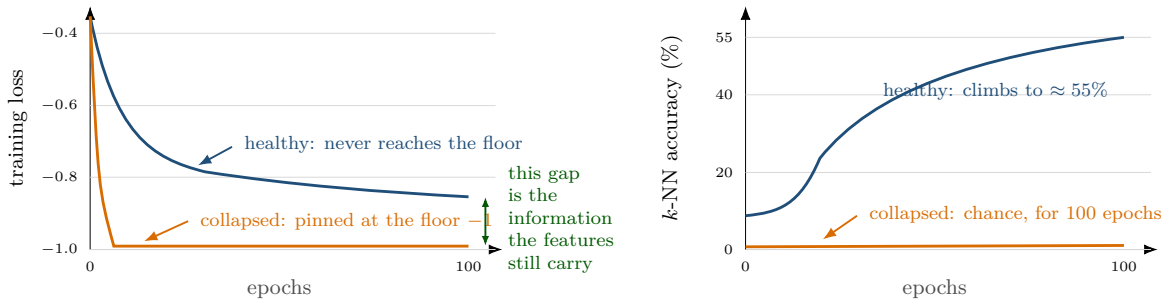
Source: Lecture 07, slide 13.

Figure 10.4 labels the same two panels with what each region *means*, since the plot itself is silent about it.

10.3.3 Architecture and the symmetric loss

Step by step — one training step of SimSiam

Step 1. Two views. Draw two augmentations t_1, t_2 from the family $\mathcal{T}$ and apply them to the same image: $\boldsymbol{x}_1 = t_1(\boldsymbol{x})$, $\boldsymbol{x}_2 = t_2(\boldsymbol{x})$. They are crops of one photograph, so they share content but not pixels.



The two panels disagree, and only one of them is in the loss. The orange run wins on the left and loses on the right. If you monitor only the left, you will ship the orange model.

Figure 10.4. The collapse experiment, annotated. Same two curves as Figure 10.3, with the parts that matter named. *Left:* the orange (no stop-gradient) trace hits the objective's floor -1 almost immediately, which for a negative cosine loss means every pair of views has similarity exactly 1 — the constant encoder. The blue trace stops around -0.85; the green bracket marks the residual gap, which is not a training failure but the price of the representation still depending on the input. *Right:* k-NN accuracy on frozen features, the number nobody optimizes and everybody should watch.

How to read it: Follow the orange trace across both panels. It is the best run in the left panel and the worst run in the right panel, and nothing in between them reconciles the two. That contradiction is what "collapse" looks like from the console.

Step 2. Encode both. Push each view through the *same* encoder f (backbone + projection MLP), giving $\boldsymbol{z}_1 = f(\boldsymbol{x}_1)$ and $\boldsymbol{z}_2 = f(\boldsymbol{x}_2)$. Same weights, two forward passes.

Step 3. Predict across. Push each projection through the *same* predictor h, giving $\boldsymbol{p}_1 = h(\boldsymbol{z}_1)$ and $\boldsymbol{p}_2 = h(\boldsymbol{z}_2)$. The predictor's job is to guess what the *other* branch will say.

Step 4. Freeze the targets. Detach $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ from the computation graph. Their numbers are still used; their gradients are thrown away. This is the single line that keeps the method alive.

Step 5. Score both directions. Compute $D(\boldsymbol{p}_1, \mathrm{sg}(\boldsymbol{z}_2))$ and $D(\boldsymbol{p}_2, \mathrm{sg}(\boldsymbol{z}_1))$, each the negative cosine similarity of two unit vectors, and average them. Averaging both directions is what "symmetrized" means: neither view is permanently the student.

Step 6. Update. Backpropagate and take one SGD step on f and h together. Gradient reaches f only through the $\boldsymbol{p}$ side of each term.

Figure 10.5 draws the result, and Figure 10.6 shows the lecture's own side-by-side of SimCLR and SimSiam, which is the fastest way to see what was deleted.

The two asymmetries are the whole method.

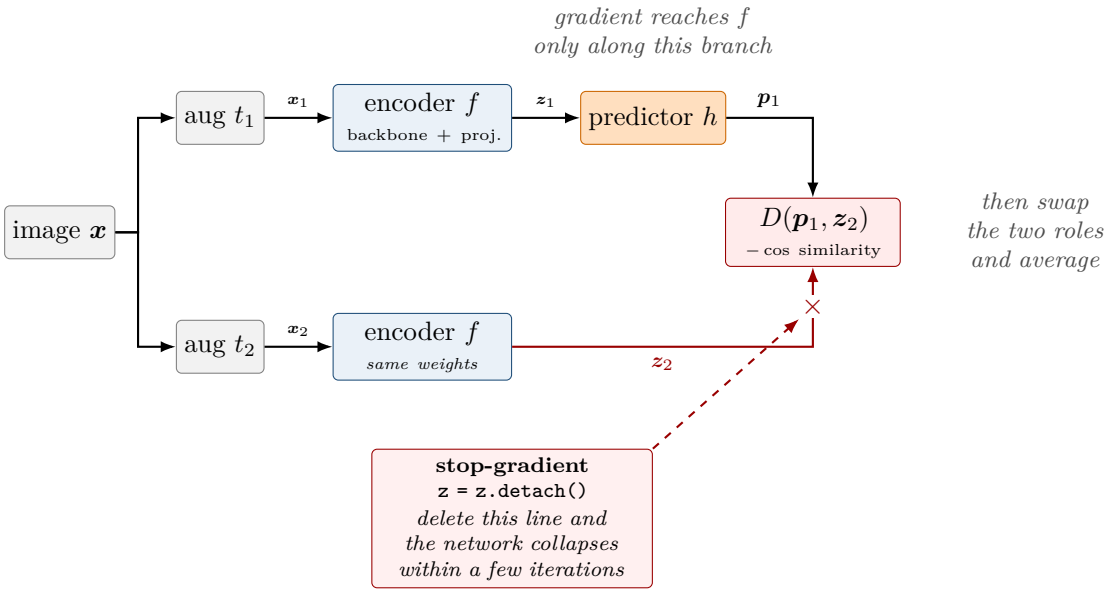


Figure 10.5. SimSiam, drawn. One encoder f is applied to both views; one predictor h is applied to one of them; the other view's projection enters the loss through a stop-gradient (the red edge and the $\times$). The full objective (10.4) runs this diagram twice with the branches swapped and averages the two scores. Everything that prevents collapse is visible here: the orange box (asymmetry in *computation*) and the red edge (asymmetry in *gradient flow*).

How to read it: Trace backwards from the loss. The gradient can walk left along the black arrows into h and then into the top copy of f; it cannot walk along the red arrow at all. That one-way street is the method.

Asymmetry 1: the predictor. The left branch computes $h(f(\boldsymbol{x}_1))$; the right branch stops at $f(\boldsymbol{x}_2)$. So the left branch is asked to *predict* the right branch's output, not to copy it.

Asymmetry 2: the gradient. Only the branch that holds the prediction gets trained. The other branch's output is treated as a fixed target: sg passes the value through unchanged on the way forward, and blocks the gradient on the way back.

The loss is then computed both ways round (swap which view is which) to use each batch twice. Compare with SimCLR: its two branches are identical, and the only thing stopping collapse is the negatives in the denominator. SimSiam has no negatives; its protection comes from making the two branches do *different* jobs.

One network, two roles. Notice that there is only *one* encoder f and *one* predictor h — both branches use the same weights. So the asymmetry that prevents collapse is not two different networks. It is two different *roles* the same network plays in each half of the loss:

- In $D(\boldsymbol{p}_1, \mathrm{sg}(\boldsymbol{z}_2))$, view 1 is the *student* (it learns) and view 2 is the *teacher* (its output is a frozen target).
- In the swapped term, the roles reverse.

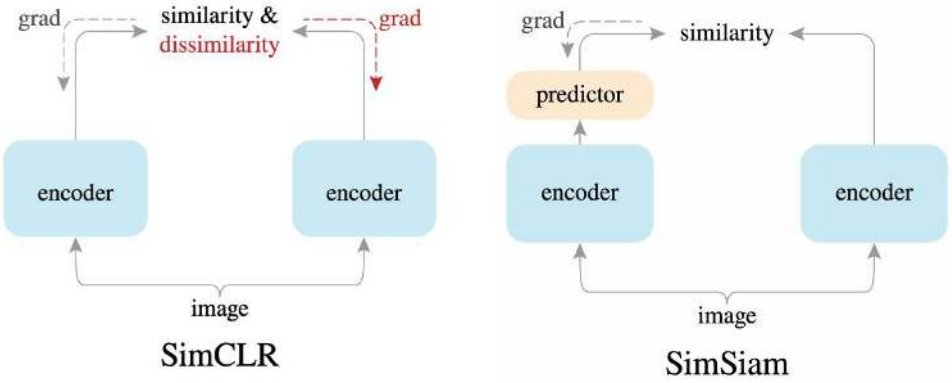


Figure 10.6. SimSiam architecture, redrawn from Lecture 07 slides 12 and 14. Two augmentations of one image pass through a shared encoder f (backbone + projection MLP). A prediction MLP h transforms the projection on one side; cosine similarity is maximized against the *other* side's projection through a stop-gradient ($\times$). The loss is then symmetrized by swapping the branches.

This student/teacher split with gradients flowing one way only will come back as BYOL's online/target networks later in this chapter, and as DINO in Chapter 11. SimSiam is the simplest possible version: everything removed except the asymmetry itself.

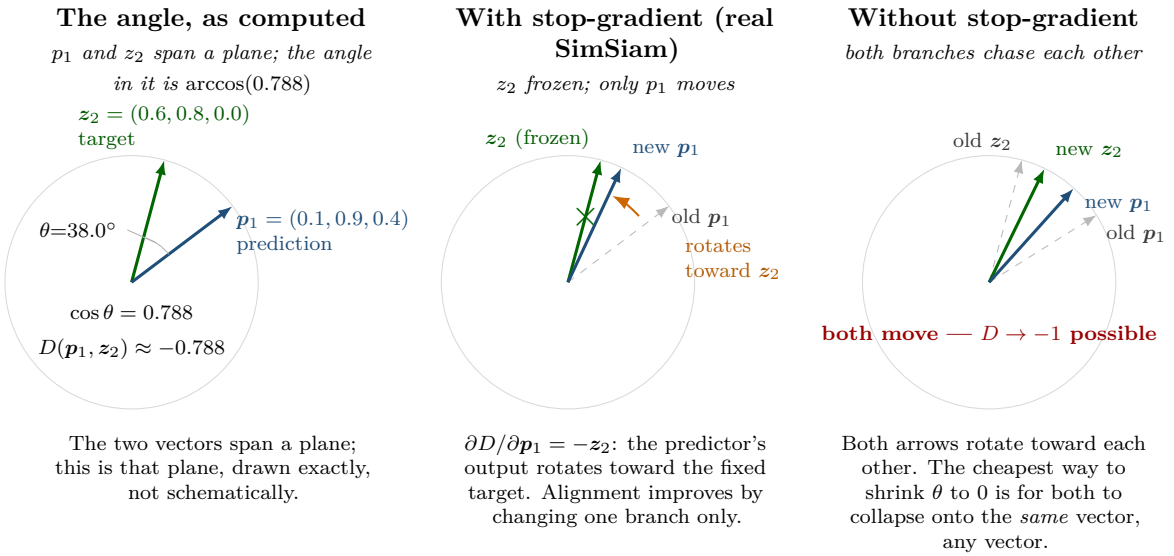
Learning note from study sessions — the dance-partners picture

In study sessions, the stop-grad/predictor pair finally became intuitive through a partner-dance reading. Ask two dancers (the two branches) to stay perfectly synchronized, mirroring each other in real time, and the globally easiest solution is for both to simply stand still: flawless agreement, zero content. That is collapse. Now change the rules: one partner — the *lead* — freezes for the duration of a phrase (stop-gradient), and the other must not merely copy them but *anticipate* their movement through a prediction step (the predictor). Now standing still is no longer cheap: the frozen partner's pose encodes real information about the input view, and the following partner must internalize enough of that structure — through a mapping it has to learn — to predict where the pose will land. Every phrase the partners swap roles, which is the symmetrized loss. Two properties do the work together: the freeze deletes the lazy mutual-drift shortcut, and the anticipation requirement makes posing meaningfully — input-dependently — the only stable strategy. Neither rule alone suffices on its own.

Worked Example — One step of SimSiam, numerically

Take a batch of one image produced as two embeddings (already in 3-D projection space for readability). Let $\boldsymbol{z}_2 = (0.6,\ 0.8,\ 0.0)^\top$ (the target) and $\boldsymbol{p}_1 = (0.1,\ 0.9,\ 0.4)^\top$ (the prediction). The target is unit length, $\|\boldsymbol{z}_2\| = 1$, and the prediction nearly so, $\|\boldsymbol{p}_1\| = \sqrt{0.01 + 0.81 + 0.16} \approx 0.990$, so $D(\boldsymbol{p}_1, \boldsymbol{z}_2) = -(0.1 \cdot 0.6 + 0.9 \cdot 0.8 + 0.4 \cdot 0)/0.990 \approx$

-0.788. Now watch what gradient descent can do. *(i) With stop-gradient:* the gradient $\partial D/\partial \boldsymbol{p}_1 \approx -\boldsymbol{z}_2 = (-0.6, -0.8, 0)$ (ignoring the small normalisation correction) flows into $h(z_1)$ only; $\boldsymbol{z}_2$ is held fixed. The predictor moves its output toward the target — alignment improves by *changing the left branch. (ii) Without stop-gradient:* the loss also differentiates through $\boldsymbol{z}_2$, handing the encoder the gradient $\partial D/\partial \boldsymbol{z}_2 \approx -\boldsymbol{p}_1/0.990$, which moves the target *toward the prediction.* Each branch now gets gradient that makes the *loss* smaller without making either embedding more informative — chasing a moving target. Repeated, this mutual chasing has a degenerate shortcut: drive both embeddings to the same vector, any vector, at which point $D = -1$ exactly and the loss reports success on an information-free representation. Figure 10.7 draws this angle and both possible updates.



The two vectors span a plane; this is that plane, drawn exactly, not schematically.

$\partial D/\partial \boldsymbol{p}_1 = -\boldsymbol{z}_2$: the predictor's output rotates toward the fixed target. Alignment improves by changing one branch only.

Both arrows rotate toward each other. The cheapest way to shrink θ to 0 is for both to collapse onto the *same* vector, any vector.

Figure 10.7. One SimSiam update, drawn in the plane spanned by $\boldsymbol{p}_1$ and $\boldsymbol{z}_2$. *Left:* the current angle between prediction and target is 38.0°, so $\cos\theta = 0.788$ and $D = -\cos\theta \approx -0.788$. *Centre:* with the stop-gradient SimSiam actually uses, $\boldsymbol{z}_2$ is treated as a constant ($\times$ marks the blocked gradient) and only $\boldsymbol{p}_1$ rotates toward it. *Right:* remove the stop-gradient and both vectors rotate toward each other — nothing stops them meeting at an arbitrary point where $D = -1$ and the loss is minimized with zero information transmitted.

How to read it: the $\times$ in the centre panel. That one blocked gradient is the entire mechanism keeping SimSiam from collapsing: it converts a two-sided chase (right panel) into a one-sided approach to a fixed target (centre panel).

```
# f: backbone + projection MLP
# h: prediction MLP
for x in loader:                # a minibatch of images
    x1, x2 = aug(x), aug(x)     # two augmented views
    z1, z2 = f(x1), f(x2)       # projections
    p1, p2 = h(z1), h(z2)       # predictions
    # symmetric loss: each branch can act as student or teacher
    L = D(p1, z2)/2 + D(p2, z1)/2
    L.backward()
    update(f, h)                # SGD

def D(p, z):                    # negative cosine similarity
    z = z.detach()              # stop gradient: teacher never
                                # propagates gradient
    p = normalize(p)            # l2-normalize
    z = normalize(z)
    return -(p*z).sum(dim=1).mean()
```

Listing 10.1. SimSiam pseudocode (PyTorch-like), the form walked line-by-line on Lecture 07 slides 15–17 (Listing 10.1). The three highlighted steps on the slides are: forward passes through encoder and predictor; the symmetric loss in which each branch alternately plays student and teacher; and the `detach` inside `D` that makes the teacher gradient-free.

The whole trick is one line. In Listing 10.1, all the asymmetry lives in `z = z.detach()` inside the similarity function. `detach` returns the same numbers but cut off from the computation graph, so `L.backward()` reaches f and h *only* through the $\boldsymbol{p}$ arguments.

The rest is routine:

- the other two lines of `D` normalise both vectors to unit length and average the negative cosine over the batch ((10.4));
- the minus sign turns "maximise agreement" into something to minimise;
- everything else is an ordinary training loop.

If you re-implement SimSiam, that one line is everything. Delete it and you get the orange curve of Figure 10.3: a symmetric, attraction-only network that finds the constant solution within a few iterations. No other line of the file knows that collapse exists.

10.3.4 Why stop-gradient plus predictor works: the verified story and the interpretation

What follows separates what the experiments establish from what is *interpretation*. Experimentally (Chen and He, 2021): removing stop-gradient collapses training (loss -1, chance accuracy) in every configuration tried; keeping the stop-gradient but removing the predictor also near-collapses, and so does a fixed random predictor; dropping the symmetrization costs a few points; and the method works with batch size 256 and without any momentum encoder. So the empirically necessary ingredients are exactly the two asymmetries of Section 10.3.3.

Simplified intuition vs. more precise view

Simplified story. Stop-gradient prevents collapse because the "teacher" side cannot run away to meet the student, so the encoder is forced to keep producing input-dependent outputs.

More precise view. Stop-gradient *alone* does not prevent collapse — the constant solution $\boldsymbol{z}_i = \boldsymbol{c}$, $\boldsymbol{p}_i = \boldsymbol{c}$ still achieves loss -1 with stop-gradient active. What changes is the gradient *path*: with the moving-target shortcut removed, the trivial basin is no longer directly downhill everywhere, and the predictor creates a genuine learning problem — predict one augmentation's representation from another — whose solution requires input-dependent features. The SimSiam paper's own interpretation frames training as an *alternating, EM-like optimization*: fix the "target" embeddings implied by the current encoder (the stop-gradient step), then optimize the current encoder and predictor to match them (the gradient step), resembling k-means' alternation between assignment and update rather than a single monolithic minimization. Under this view the predictor is a learned optimizer of a symmetry-breaking objective: it absorbs the view-dependent part of the mapping, so the encoder itself can specialize in the view-invariant part. Both the EM analogy and the optimizer reading are hypotheses consistent with the evidence, not theorems; later analyses have probed the predictor's role further (e.g. showing that under a strong-linear predictor the training dynamics can be driven toward local predictability while the loss still appears non-collapsed). Treat the mechanism as an open question, with the recipe — asymmetric predictor, stop-gradient, symmetrized loss — as the robust core.

The EM-like view, worked out. The paper's alternating-optimization reading rewards being written down explicitly, because it converts stop-gradient from a trick into a step of a familiar algorithm. Define, over the whole dataset of N images, the objective

$$\mathcal{E}(\boldsymbol{\theta}, \boldsymbol{\eta}) = \frac{1}{2N} \sum_{i=1}^{N} \mathbb{E}_{v_1, v_2 \sim \mathcal{T}} \left\| h\big(f_{\boldsymbol{\theta}}(v_1(\boldsymbol{x}_i))\big) - \boldsymbol{\eta}_i \right\|_2^2, \tag{10.5}$$

where $\boldsymbol{\eta}_i \in \mathbb{R}^{d_z}$ is a *target vector* treated as a free variable per image — the idealized "assigned" representation of image $\boldsymbol{x}_i$, one target no matter which augmentation is drawn. Now imagine minimizing $\mathcal{E}$ by alternation, in the style of k-means or EM. *Assignment step:* holding $\boldsymbol{\theta}$ fixed, the problem separates over images, and the best target is the mean representation over augmentations,

$$\boldsymbol{\eta}_i^{\text{new}} = \mathbb{E}_{v \sim \mathcal{T}}\big[f_{\boldsymbol{\theta}}(v(\boldsymbol{x}_i))\big]. \tag{10.6}$$

Why this might work: training as taking turns. One way to understand SimSiam is as two alternating jobs.

1. **Decide what each image's representation *should* be.** Ideally, the average representation over all possible augmentations of that image. SimSiam cheaply approximates it with just *one* other augmented view, $\boldsymbol{z}_2$, and freezes it with sg.
2. **Train the network to produce that.** Gradient descent on Equation (10.4).

The predictor h then has a clear job: one other view is a noisy guess at the ideal average, and h learns to bridge the gap from "this view" to "where the average should be", so the encoder itself does not have to. Collapse is slowed because step 1 always uses yesterday's frozen encoder.

This is only an analogy — using a single sample breaks the guarantees that make the textbook version provably work. But it correctly predicts one odd detail of SimSiam: the predictor's learning rate is kept *fixed* rather than decayed, so that it keeps tracking step 1 closely.

Common Confusion

"Stop-gradient blocks the collapse gradient, so it is the whole mechanism." No: a constant representation is still a global optimum of (10.4) with stop-gradient, and several papers reproduce near-collapse when the predictor is removed even with stop-gradient intact. The correct statement is that the two asymmetries act *jointly*: stop-gradient removes the trivial gradient path, and the predictor's input-conditional mapping makes non-constant solutions (locally) easier to find. "It's like momentum contrast" is equally wrong — SimSiam's headline finding is precisely that no EMA encoder and no large queue are needed.

Failure Modes

- **Stop-gradient removed:** loss dives to -1 within a few epochs; features are constant; k-NN accuracy $\approx$ chance (the slide 13 phenomenon, reproduced by Chen and He, 2021).
- **Predictor removed or h = identity:** accuracy drops to near-zero in SimSiam's ablations (BYOL without a predictor also fails badly in its own ablation); the predictor is load-bearing here in the strongest empirical sense.
- **Careless fixed learning rate for h:** the paper's EM-like view suggests h should be optimized "more" than f; a fixed (non-decayed) learning rate for the predictor is a standard implementation detail — trainable but easy to get wrong.
- **Over-aggressive augmentations:** if the two views share little semantic content, the prediction problem becomes ill-posed and the method inherits augmentation-sensitivity failure modes from Chapter 8 (e.g. color drop destroying color-critical information).
- **Single-branch shortcuts at small scale:** with very small datasets the symmetric loss can be partially satisfied by encoding view-specific noise; monitor k-NN accuracy, not the loss.

Paper Analysis: Chen & He 2021, *Exploring Simple Siamese Representation Learning*

Problem: does joint-embedding SSL fundamentally require negatives, large batches, or momentum encoders? **Prior work:** SimCLR (in-batch negatives), MoCo (queue + EMA), BYOL (predictor + EMA), SwAV (clustering targets). **Insight:** with a stop-gradient and an asymmetric predictor, none of the three pillars are necessary; collapse is avoidable by architecture and gradient surgery alone. **Method/objective:** equations (10.3) and (10.4): siamese encoder f, predicted branch $h \circ f$, symmetrized negative-cosine loss, stop-gradient on the target side. **Experiments/results:** ImageNet pretraining with a ResNet-50, batch size 256, standard augmentations, 100–800 epochs; ablations isolate each component (stop-grad, predictor, symmetrization) as sketched in Figure 10.3; final linear-probe accuracy is competitive with SimCLR/MoCo/BYOL (about 68% top-1 at 100 epochs and 71.3% at 800 in the paper's comparison). **Limitations:** no convergence guarantee; the EM view is provided as an intuition, not a proof. At 100 epochs it was actually the best of the compared methods; with longer training BYOL and SwAV pull ahead. **What to remember:** negatives are not *necessary*; the necessary ingredient is some mechanism that makes collapse not-a-solution — here, decoupled gradient paths plus a prediction asymmetry.

10.4 BYOL: SimSiam plus EMA — and still no negatives

BYOL (*Bootstrap Your Own Latent*, Grill et al., 2020) predates SimSiam by months and supplied the template SimSiam simplified. The two-branch layout is the same — online network with a predictor, target network without one, negative-cosine (or normalized squared-Euclidean) agreement, stop-gradient on the target. The single structural addition is that the target network's parameters $\boldsymbol{\phi}$ are *not* the online network's parameters at all: they are an exponential moving average of them,

$$\boldsymbol{\phi} \leftarrow \boldsymbol{m}\,\boldsymbol{\phi} + (1-\boldsymbol{m})\,\boldsymbol{\theta}, \qquad \boldsymbol{m} \in [0,1) \text{ close to } 1, \tag{10.7}$$

where $\boldsymbol{\theta}$ are the online (student) parameters and $\boldsymbol{m}$ is the EMA coefficient — e.g. 0.996, raised towards 1 during training.

This is the same momentum trick as MoCo's key encoder (Chapter 9) and DINO's teacher (Chapter 11). Only the purpose changes: in MoCo it kept the keys consistent; here it makes the target *a slowly drifting copy of the student*.

Figure 10.8 draws the wiring. Put it next to Figure 10.5 and you will find exactly one difference: the bottom encoder is no longer f itself but its slow EMA copy $f_{\boldsymbol{\phi}}$. In the lecture's comparison table (Lecture 07, slide 33), BYOL with a ResNet-50 reaches 74.4% linear and 64.8% k-NN ImageNet accuracy — with no negatives anywhere.

Two points for the chapter's map of methods:

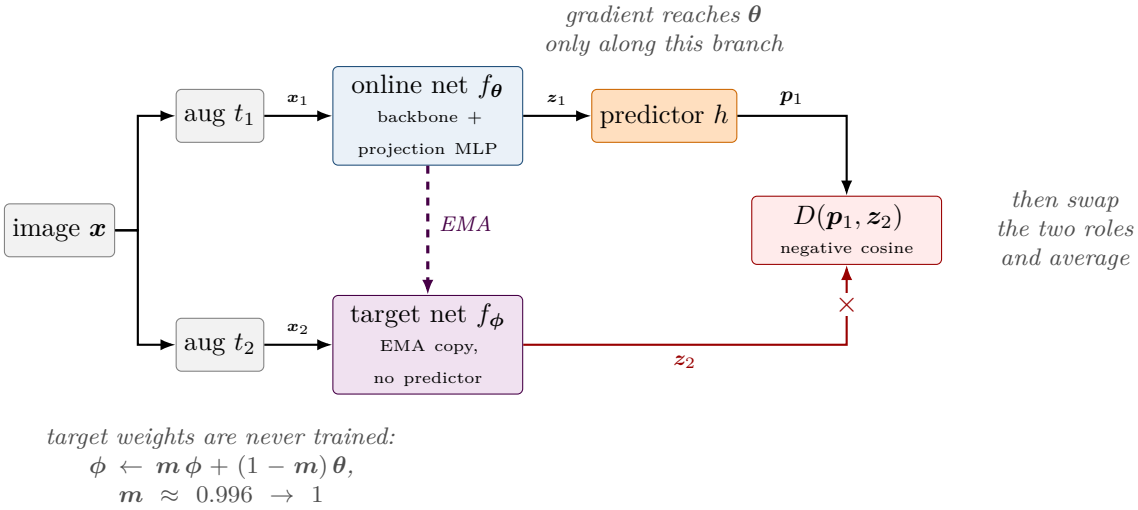


Figure 10.8. BYOL, drawn in the same layout as SimSiam (Figure 10.5) so the difference stands out. The online network (top) is SimSiam's branch verbatim: encoder $f_{\boldsymbol{\theta}}$ plus predictor h, trained by backpropagation. The target network (bottom, violet) is a *second set of weights* $f_{\boldsymbol{\phi}}$ with no predictor and no gradient path at all: the $\times$ cuts the red edge, and the dashed violet arrow shows how $\boldsymbol{\phi}$ changes — only through the EMA update $\boldsymbol{\phi} \leftarrow \boldsymbol{m\phi} + (1-\boldsymbol{m})\boldsymbol{\theta}$ of (10.7); m starts near 0.996 and rises toward 1, so the target is a slowly-lagging ghost of the student. Where SimSiam holds the target still *within* a step by detaching it, BYOL additionally holds it still *across* steps by averaging. Symmetrization (swap roles, average) is identical.

How to read it: Follow the violet arrow, not the boxes. Every other method in this chapter moves its target by gradient descent; BYOL is the one whose target is a lagged copy of the student — a second, independent source of asymmetry on top of the predictor.

- **Who removed what.** BYOL had already shown that negatives are unnecessary. SimSiam showed that the EMA is unnecessary too, leaving *predictor + stop-gradient* as the smallest recipe known at the time.
- **The parts overlap.** Both methods fail without a predictor, but follow-up analyses of BYOL-style training find that the EMA makes the method more tolerant of how the predictor is trained, so the EMA seems to share part of the predictor's stabilising job.

A reasonable guess (a hypothesis, not a proof): EMA and predictor do the *same* job — *asynchrony*. Each makes the two branches aim at the same target at slightly different times or from slightly different inputs, so "agree perfectly, right now" can no longer be achieved by a constant function.

Common Confusion

"BYOL fails without BatchNorm because BN leaks batch statistics as a hidden contrastive signal." Versions of this claim circulated after BYOL's release (the so-called *BN-as-negatives* hypothesis). The published evidence is more nuanced: BYOL with group norm plus weight standardization still trains to meaningful accuracy without

any batch statistics, while some BN-free configurations fail. BatchNorm interacts with self-supervision in subtle ways, but "BYOL secretly uses negatives" is not a supported reading of the ablation literature, and SimSiam's results make it moot: SimSiam avoids collapse with or without batch-statistic leakage because the predictor and stop-gradient carry the load.

10.5 Barlow Twins: redundancy reduction on the cross-correlation matrix

Family (c) replaces implicit stabilization with an explicit statistical constraint. To see why anyone reached for a third mechanism after family (b) worked, recall what the asymmetry methods cost conceptually: the reason they avoid collapse is entangled with their optimization dynamics, so it cannot be checked on a static snapshot of the model. Barlow Twins (Zbontar et al., 2021) takes the opposite stance — write down, as an objective, a direct *measurement* of the property collapse violates. The lecture seeds this family with a single teaser slide (Lecture 06, slide 72) showing the objective; we expand it here.

The intuition is division of labor. Imagine training a team of security guards to describe suspects. The contrastive approach of Chapter 9 shows the guards photos of *different* suspects and demands that their descriptions differ — which works, but each training step must wheel in thousands of comparison suspects. Barlow Twins shows only photos of *the same* suspect under different lighting, and issues two rules: (1) your several descriptions must agree with one another, and (2) no two of you may cover the same detail — if one guard commits to height, another takes build, another clothing. "Collapse" for this team would be every guard chanting the memorized sentence; rule (2) forbids it without any other suspect ever entering the room. The guards are embedding dimensions, rule (1) is the invariance term, and rule (2) is the redundancy-reduction term of the loss we now derive.

10.5.1 From identity cross-correlation to a loss

Two distorted views of each image in a batch pass through twin networks (shared weights, no stop-gradient, no predictor), producing embedding matrices $\boldsymbol{Z}^A, \boldsymbol{Z}^B \in \mathbb{R}^{B \times d_z}$, where B is the batch size. The *empirical cross-correlation matrix* between the two view-embeddings, with entries indexed by embedding dimensions i and j and computed over the batch index b, is

$$\mathcal{C}_{ij} = \frac{\sum_{b=1}^{B} z^A_{b,i}\, z^B_{b,j}}{\sqrt{\sum_{b=1}^{B} (z^A_{b,i})^2}\,\sqrt{\sum_{b=1}^{B} (z^B_{b,j})^2}} \in [-1, 1]. \tag{10.8}$$

What should $\mathcal{C}$ look like if training succeeded? Two desiderata from earlier chapters answer this. *Invariance* (Chapter 8): the two views describe the same image, so dimension i of view A should correlate perfectly with dimension i of view B: $\mathcal{C}_{ii} \to 1$. *Sufficiency / decorrelation*:

the embedding dimensions should carry different, non-redundant pieces of information; cross-dimension correlations should vanish: $\mathcal{C}_{ij} \to 0$ for $i \neq j$. Together they say $\mathcal{C} \to \boldsymbol{I}$, the identity — the target matrix drawn next to the empirical one on the slide. The loss is simply the squared distance to that target, split into the two terms:

$$\mathcal{L}_{\mathrm{BT}} = \underbrace{\sum_{i=1}^{d_z} (1 - \mathcal{C}_{ii})^2}_{\text{invariance term}} + \lambda \underbrace{\sum_{i=1}^{d_z} \sum_{j \neq i} \mathcal{C}_{ij}^2}_{\text{redundancy-reduction term}}, \tag{10.9}$$

where $\lambda > 0$ balances the two (the paper uses 5×10^{-3}). Figure 10.9 draws both matrices as heatmaps for a toy $d_z = 8$: the empirical $\mathcal{C}$ of a partially-trained batch next to the identity target the loss is pulling it toward.

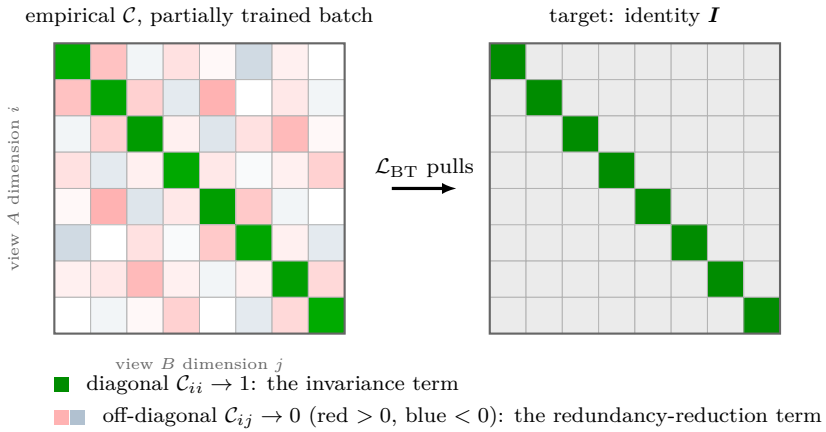


Figure 10.9. The Barlow Twins objective as two heatmaps (toy $d_z = 8$; the real matrix is 8192×8192). *Left:* the empirical cross-correlation matrix $\mathcal{C}$ of (10.8) for a batch partway through training. Diagonal cells (green) measure how strongly the same dimension agrees across the two views — they should all be 1. Off-diagonal cells measure how much dimension i of view A leaks into dimension j of view B — red for positive, blue for negative correlation, saturation proportional to $|\mathcal{C}_{ij}|$ — and should all be 0. *Right:* the identity target. The loss (10.9) is the squared distance between the two grids, with the diagonal errors broken out and the off-diagonal sum downweighted by λ. The strongly saturated off-diagonal pairs here (e.g. $\mathcal{C}_{25} = 0.5$) are what dimensional collapse inflates everywhere: a low-rank embedding bills nearly every off-diagonal cell at ± 1 (cf. (10.11)).

How to read it: The right grid is not a metaphor, it is the literal target of the loss. Any cell on the left that differs from its counterpart on the right is being charged: dark-enough diagonal cells are under-charged, any colored off-diagonal cell is over-charged.

What exactly is $\mathcal{C}_{ij}$? The slide's short formula hides a normalisation. Equation (10.8) is the ordinary *Pearson correlation* between two columns:

- dimension i of view A's embeddings, and
- dimension j of view B's embeddings,

measured across the B images of the batch. (The batch plays the role of "samples"; the two dimensions are the two "variables".)

First standardise each column over the batch, so it has mean 0 and variance 1: subtract the batch mean $\mu_i^A = \frac{1}{B}\sum_b z_{b,i}^A$ and divide by the batch standard deviation $\sigma_i^A = \sqrt{\frac{1}{B}\sum_b (z_{b,i}^A - \mu_i^A)^2}$ (likewise for view B). Then (10.8) becomes a single inner product,

$$\mathcal{C}_{ij} = \frac{1}{B}\sum_{b=1}^{B} z_{b,i}^A \, z_{b,j}^B, \tag{10.10}$$

i.e. the (i,j) entry of $\frac{1}{B}(\boldsymbol{Z}^A)^\top \boldsymbol{Z}^B$, with the normalisation already applied to the data. Two practical consequences:

1. **It is easy to implement.** Subtract the batch mean, divide by the batch standard deviation (plus a small epsilon for safety), do one matrix multiply. The slides' "tiny normalisation stage after the networks" is exactly this — not an extra learned layer.
2. **It explains the hunger for large batches.** $\mathcal{C}$ is a $d_z \times d_z$ table of correlations, and each entry is estimated from only B samples. With a small batch the estimates are noisy and biased, and no value of λ fixes that. At the paper's width ($d_z = 8192$) one $\mathcal{C}$ holds about 67 million correlations. Large batches are a statistical necessity, not a matter of taste.

How the loss stops collapse. The loss (10.9) has two jobs: the diagonal pulls the two views of an image together, and the off-diagonal *prevents collapse*. Check it on the two kinds of collapse.

- **Complete collapse** — every image gets the same vector, $z_{b,i}^A = c_i$ and $z_{b,j}^B = c_j$ for all b. Subtracting the batch mean leaves all zeros ($c_i - c_i = 0$), and the standard deviation in the denominator of (10.8) is zero. The correlation is not "bad" — it is *undefined*. In practice the small safety epsilon turns those columns into zeros, and the invariance term charges $\sum_i (1-0)^2 = d_z$ — as if no dimension agreed at all.
- **Dimensional collapse** — the sneakier kind. Embeddings are squeezed onto a line or a thin subspace, so the dimensions move together. Now most off-diagonal entries are close to ± 1, and the redundancy term builds up a penalty on the scale of

$$\lambda \sum_i \sum_{j \neq i} \mathcal{C}_{ij}^2 \;\approx\; \lambda \, d_z(d_z - 1), \tag{10.11}$$

  which at the paper's embedding width $d_z = 8192$ and $\lambda = 5 \cdot 10^{-3}$ is on the order of 10^5 — an overwhelming gradient forcing the dimensions apart.

This is the sense in which family (c) "defines collapse away": chapter 9's negatives made collapse *expensive per sample*; (10.9) makes it *expensive per dimension-pair*, quadratically so.

The same calculation exposes where the objective can be fooled. Squared correlations are second-order statistics: two dimensions with a deterministic nonlinear relationship (say $z_2 = z_1^2$ after centering) can have $\mathcal{C}_{12} \approx 0$ while carrying exactly the same information — decorrelation is not yet independence. And because $\mathcal{C}$ is built from the batch, it constrains only *batchwise* statistics; the objective has nothing to say about rare inputs. Both gaps motivate the refinements later in the chapter (VICReg's per-branch variance floor) and are the price of an anti-collapse certificate you can compute in closed form.

Worked Example — Computing $\mathcal{C}$ on a 2-sample, 3-dimension batch

Let $B = 2$, $d_z = 3$, with view embeddings

$$\boldsymbol{Z}^A = \begin{pmatrix} 1 & 2 & 0 \\ -1 & 2 & 1 \end{pmatrix}, \qquad \boldsymbol{Z}^B = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 2 & 1 \end{pmatrix}.$$

Each column of (10.8) uses the batch as the correlation "sample dimension." (To keep the arithmetic readable we apply (10.8) literally, *without* the batch-mean subtraction a real implementation does first; with only $B = 2$ samples, centering would leave column 2 of $\boldsymbol{Z}^A$, which is constant, undefined.) For $(i, j) = (1, 1)$: numerator $= (1)(1) + (-1)(-1) = 2$; norms $\sqrt{1^2 + (-1)^2} = \sqrt{2}$ each; so $\mathcal{C}_{11} = 2/2 = 1$. Similarly $\mathcal{C}_{22}$: numerator $= (2)(0) + (2)(2) = 4$; norms $\sqrt{4+4} = 2\sqrt{2}$ and $\sqrt{0+4} = 2$; $\mathcal{C}_{22} = 4/(4\sqrt{2}) \approx 0.7071$. And $\mathcal{C}_{33}$: numerator $= (0)(1) + (1)(1) = 1$; norms $\sqrt{0+1} = 1$ and $\sqrt{1+1} = \sqrt{2}$; $\mathcal{C}_{33} = 1/\sqrt{2} \approx 0.7071$. Off-diagonal, e.g. $\mathcal{C}_{12}$: numerator $= (1)(0) + (-1)(2) = -2$; norms $\sqrt{2}$ and 2; $\mathcal{C}_{12} = -2/(2\sqrt{2}) \approx -0.7071$. Or $\mathcal{C}_{23}$: numerator $= (2)(1) + (2)(1) = 4$; norms $2\sqrt{2}$ and $\sqrt{2}$; $\mathcal{C}_{23} = 4/4 = 1$. Assembling all nine,

$$\mathcal{C} \approx \begin{pmatrix} 1 & -0.7071 & 0 \\ 0 & 0.7071 & 1 \\ -0.7071 & 1 & 0.7071 \end{pmatrix}.$$

Note that $\mathcal{C}$ is not symmetric: it is a *cross-view* correlation, so $\mathcal{C}_{ij}$ compares dimension i of view A against dimension j of view B, and $\mathcal{C}_{12} \approx -0.7071$ while $\mathcal{C}_{21} = 0$. Reading off the loss: invariance term $(1-1)^2 + (1-0.7071)^2 + (1-0.7071)^2 \approx 0 + 0.0858 + 0.0858 = 0.1716$; redundancy term (six off-diagonal squares) $\approx 0.5 + 0 + 0 + 1 + 0.5 + 1 = 3.0$. With $\lambda = 5 \times 10^{-3}$, $\mathcal{L}_{\mathrm{BT}} \approx 0.1716 + 0.015 = 0.1866$. Training proceeds by adjusting the twin networks so that the diagonal creeps toward 1 while the off-diagonal shrinks — dimension 2 is the weak spot in this batch (its views agree only up to 0.7071, as do dimension 3's), and its content has crossed channels: dimension 2 of view A covaries *perfectly* with dimension 3 of view B ($\mathcal{C}_{23} = 1$, mirrored by $\mathcal{C}_{32} = 1$), so the twins agree on that information but have parked it in different coordinates — a leak the redundancy term is still billing at full price. Figure 10.10 draws the full grid computed above.

The code realization is as compact as the mathematics. Standardize each branch along the batch axis, compute $\mathcal{C}$ as one scaled matmul $\frac{1}{B}(\boldsymbol{Z}^A)^\top \boldsymbol{Z}^B$, form the identity target once,

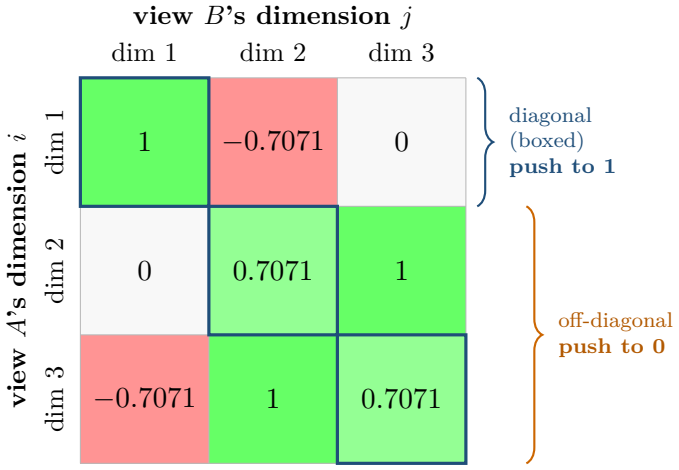


Figure 10.10. The full cross-correlation matrix $\mathcal{C}$ from the worked example, as a heat map. Cell (i, j) compares dimension i of view A with dimension j of view B; green shades are positive, red shades negative, intensity proportional to $|\mathcal{C}_{ij}|$. The three boxed cells are the diagonal, which $\mathcal{L}_{\text{BT}}$ pulls toward 1; every other cell is pulled toward 0. Note the matrix is *not* symmetric — $\mathcal{C}_{23} = 1$ but $\mathcal{C}_{32} = 1$ too here only by coincidence of this small batch, while $\mathcal{C}_{12} = -0.7071 \neq \mathcal{C}_{21} = 0$.

How to read it: cell $(2, 3)$, off the diagonal but coloured as dark as the diagonal cells. That is dimension 2 of view A agreeing almost perfectly with dimension 3 of view B — exactly the "leak" the worked example's prose describes: real shared information, parked in the wrong coordinate, still billed at full price by the off-diagonal penalty.

and take squared errors on the diagonal and off-diagonal separately — ten lines of PyTorch in total, with no queue, no momentum update, no `detach`, and no pairwise comparisons between different images anywhere in the inner loop. It is the only method in this chapter whose anti-collapse mechanism is a property you can evaluate by hand on a saved checkpoint.

Reading the Barlow Twins target, entry by entry. The loss pushes the cross-correlation matrix $\mathcal{C}$ towards the identity matrix. Look at what that asks for.

Diagonal entries $\to 1$. Feature i from view 1 should be perfectly correlated with feature i from view 2. In plain words: whatever feature i measures, it should measure the same thing on both crops. And because $\mathcal{C}$ is a *correlation*, hitting exactly 1 also rules out the cheat of shrinking every feature to zero.

Off-diagonal entries $\to 0$. Feature i should carry *no* information about feature j. Every feature has to find its own job.

Everyday analogy — a team of specialists

Think of each feature dimension as a guard assigned to watch an image. The diagonal rule says every guard must report the same thing about both crops — reliable witnesses. The off-diagonal rule says no two guards may be watching the same thing — no duplicated effort. A team that obeys both rules covers as much of the image as it has members.

Where the analogy breaks. "No correlation" is weaker than "no shared information": two features can be uncorrelated yet still dependent in a non-linear way. Barlow Twins only enforces the linear version.

This also explains the paper's scaling result: accuracy keeps improving as the embedding gets wider (up to $d_z = 8192$ in the reported sweep). Every extra dimension is one more specialist. And the split form of the loss is a useful diagnostic: if the diagonal drifts below 1, the features disagree across views; if off-diagonal entries grow, features are starting to copy each other.

A useful reading of (10.9): it is the information-bottleneck principle implemented with second-order statistics instead of information terms. The diagonal term maximizes what the embedding preserves about the augmentation-invariant content; the off-diagonal term minimizes the redundancy between preserved components. The name honours the neuroscientist Horace Barlow's redundancy-reduction principle for sensory coding.

Failure Modes

- λ **too small:** the off-diagonal term loses force and the solution drifts toward a rank-deficient embedding (many $\mathcal{C}_{ij} \to 1$), i.e. partial/dimensional collapse; typical symptom is good-looking training loss with degraded probe accuracy.
- λ **too large:** decorrelation dominates, the model scrambles to orthogonalize dimensions before aligning views, and invariance suffers.
- **Small batch size:** $\mathcal{C}$ is a batch-level statistic, so gradient estimates become noisy and correlation estimates biased at small B; Barlow Twins is markedly more batch-sensitive than SimSiam/BYOL (batches of thousands are standard in the paper).
- **Too few embedding dimensions:** decorrelation requires capacity in Z; with small d_z the model cannot be both invariant and decorrelated, and the two terms fight.
- **Augmentation shortcuts:** as with all family-(c)/(b) methods, semantics-destroying augmentations transfer here unchanged (Chapter 8).

Paper Analysis: Zbontar et al. 2021, *Barlow Twins: Self-Supervised Learning via Redundancy Reduction*

Problem: obtain collapse-free joint-embedding learning without negatives, asymmetry, stop-gradient, or momentum encoders. **Prior work:** contrastive methods; BYOL; information-bottleneck-style objectives; the redundancy-reduction view of representation learning. **Insight:** drive the cross-correlation matrix between view embeddings to the identity — invariance and decorrelation are jointly sufficient to rule out collapse. **Method/objective:** twin shared-weight networks; per-batch cross-correlation matrix (10.8); loss (10.9) with off-diagonal weight λ. **Experiments/results:** ImageNet linear-probe and transfer experiments competitive with contrastive and asymmetric methods for ResNet-50 (the Lecture-07 slide 33 table carries "BarlowT" at 73.2 linear / 66.0 k-NN); ablations over λ, batch size, and embedding dimension. **Limitations:** batch-size sensitivity; the squared-correlation objective acts only on second-order statistics, so it cannot detect higher-order redundancy. **What to remember:** collapse is preventable by *measuring* it — embedding covariance pathology — and subtracting it from the objective.

10.6 VICReg: variance, invariance, covariance — an explicit triad

VICReg (Bardes et al., 2022) factorizes the Barlow-Twins intuition into three separately interpretable terms, applied to each branch's own embedding batch. Given view-embedding batches $\boldsymbol{z}$ and $\boldsymbol{z}'$ (twin encoders, no asymmetry):

$$\mathcal{L}_{\text{VICReg}} = \lambda\, s(\boldsymbol{z}, \boldsymbol{z}') + \mu\,[v(\boldsymbol{z}) + v(\boldsymbol{z}')] + \nu\,[c(\boldsymbol{z}) + c(\boldsymbol{z}')], \tag{10.12}$$

where

$$\begin{aligned} s(\boldsymbol{z}, \boldsymbol{z}') &= \tfrac{1}{B}\textstyle\sum_b \|\boldsymbol{z}_b - \boldsymbol{z}'_b\|_2^2 && \text{(invariance term)},\\ v(\boldsymbol{z}) &= \tfrac{1}{d_z}\textstyle\sum_j \max\left(0,\ \gamma - \mathrm{Std}(\boldsymbol{z}_{:,j})\right) && \text{(variance term)},\\ c(\boldsymbol{z}) &= \tfrac{1}{d_z}\textstyle\sum_{i\neq j} \left[\mathrm{Cov}(\boldsymbol{z})_{ij}\right]^2 && \text{(covariance term)}, \end{aligned}$$

and λ, μ, ν are weights, γ a target standard deviation (typically $\gamma = 1$), Std the per-dimension batch standard deviation, and $\mathrm{Cov}(\boldsymbol{z})$ the batch covariance matrix of the embeddings. Figure 10.11 draws the three terms and what each one forbids.

Why each of the three terms is there.

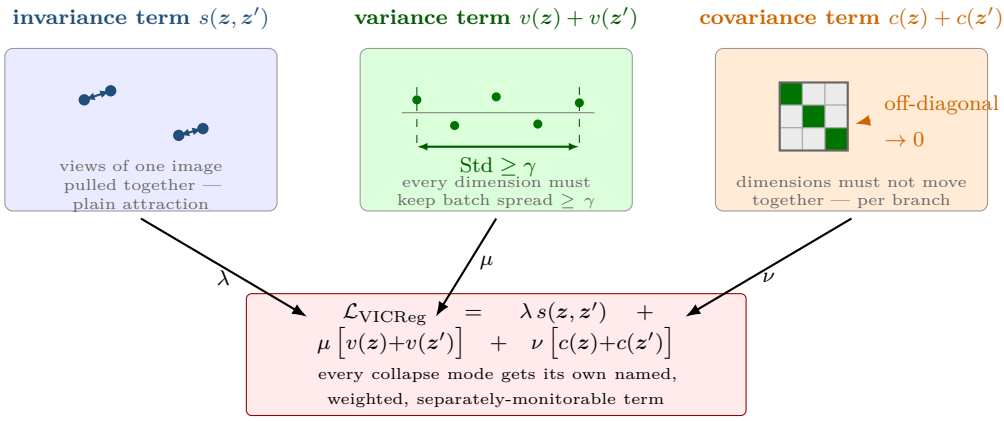


Figure 10.11. VICReg's triad, one panel per term of (10.12). *Invariance* (blue) is plain attraction between the two views of each image — on its own it is exactly the collapsing objective (10.1). *Variance* (green) puts a floor under each dimension's batch standard deviation: the dashed rails mark the hinge target γ, and a collapsed batch, which has zero spread, pays the term in full. *Covariance* (orange) squares the off-diagonal entries of each branch's own covariance matrix, so dimensions cannot share a direction — that is the dimensional-collapse loophole the variance term alone leaves open. Unlike Barlow Twins' single cross-view matrix, all three statistics are computed *per branch*, and each appears in the training log as a separately weighted number (λ, μ, ν), which is what makes VICReg the most diagnosable objective in the chapter.

How to read it: Covering any one panel tells you which collapse mode survives: without green, the constant encoder wins; without orange, the collapsed-to-a-line encoder wins; without blue, nothing aligns the views at all. The design is minimal in the strong sense — each term exists because the other two have a hole.

Term	Job
s (invariance)	pull the two views together. Alone it collapses — this is (10.1) again.
v (variance)	every dimension must keep at least γ of spread across the batch. A constant embedding cannot, so complete collapse is ruled out.
c (covariance)	decorrelate the dimensions. Without it, all dimensions could spread out along *one shared direction* (dimensional collapse). This is Barlow Twins' off-diagonal idea, applied within each branch instead of across branches.

VICReg is therefore the most explicit member of family (c): every way of collapsing gets its own named, weighted term that you can watch during training. In practice its accuracy is similar to Barlow Twins', and it is somewhat less sensitive to batch size — consistent with per-branch statistics being easier to estimate than a full cross-correlation.

Common Confusion

"VICReg prevents collapse with the variance term only." The variance term alone forces spread but not decorrelation; a collapsed-to-a-line embedding can satisfy it. The covariance term alone allows constant embeddings with zero covariance ("no covariance to

penalize"). All three terms are needed; this is deliberate design, not multiply-redundant safety.

10.7 Why does any of this work? Two sufficient conditions

Across the families, the same two ingredients recur.

(1) View agreement plus decorrelation forces information retention. The contrastive view from Chapter 9 put a mutual-information bound on InfoNCE ($I \geq \log(K{+}1) - \mathcal{L}_{\text{NCE}}$ in this book's convention — van den Oord et al., 2018b's N counts candidates; Chapter 9 translates) ; the negative-free view has its analogue in second-order language. If embeddings of the two views agree ($\mathcal{C}_{ii} \to 1$ or cosine similarity $\to 1$), then Z must encode whatever the two views share — which, under the augmentations actually used, includes semantics and excludes photometric noise. If additionally the embedding dimensions are decorrelated with unit variance (family c) or are simply *not constant* and held open by asynchrony (family b), then no information was thrown away in the process: decorrelation combined with invariance is a proxy for *uniformity on the hypersphere*, which was the good geometry that negatives secured in Chapter 9. In short, second-order sufficiency statistics (covariance structure) stand in for the sample-level repulsion term. The argument is empirical but matches the deep results that uniformity and alignment explain InfoNCE's quality, and it explains why all successful negative-free methods end up enforcing some decorrelation-like constraint whether they say so or not.

(2) Asynchrony beats repulsion as a stabilization mechanism. In family (b) methods, stability comes from making the two sides' targets mutually non-trivial to satisfy simultaneously — predictor versus projection (SimSiam), online versus slowly-averaged target (BYOL, DINO). The *uncertainty of the predictor* plays an important role: since h cannot perfectly predict the other view's embedding early in training, its errors act as a structured, annealing noise that prevents the system from settling into the exactly-collapsed fixed point. This is why remove-the-predictor ablations are catastrophic for SimSiam, and why the EMA lag in BYOL appears to share part of h's job. The EM-like reading from Section 10.3.4 rephrases it: each iteration solves for Z against a frozen target, so the dynamical system contracts toward input-dependent fixed points rather than diverging toward the constant one. None of this is theorem-grade; it is the compendium of mechanisms that repeatedly show up in ablations, and Chapter 15 revisits which of them constitute "the" explanation.

10.8 Head-to-head comparison

Table 10.1 collects the mechanics. It deliberately reports machinery rather than leaderboard positions — at matched training budgets, linear-probe results among all six methods sit within a few ImageNet points of each other (the Lecture-07 slide 33 spans SimCLR 69.1,

Table 10.1. Mechanics of six joint-embedding methods. "Negatives" = does the loss contain explicit repulsion between different images; "EMA" = target / key network updated as an exponential moving average; "predictor" = asymmetric MLP on one branch only; collapse mechanism = the ingredient whose removal collapses training (per each paper's ablations). Batch sensitivity is qualitative, from the papers' ablations.

Method	Negatives?	EMA teacher?	Predictor?	Batch size	Collapse avoided by
SimCLR (Chen et al., 2020a)	yes (in-batch)	no	no	thousands; very sensitive	repulsion in InfoNCE
MoCo (He et al., 2020)	yes (queue)	yes (key enc.)	no	insensitive (queue)	repulsion, consistent keys
BYOL (Grill et al., 2020)	no	yes	yes	moderate	predictor + EMA asynchrony
SimSiam (Chen and He, 2021)	no	no	yes	low (256 works)	stop-grad + predictor
Barlow Twins (Zbontar et al., 2021)	no	no	no	high; sensitive	off-diagonal decorrelation
VICReg (Bardes et al., 2022)	no	no	no	moderate	variance + covariance terms

MoCo v2 71.1, Barlow Twins 73.2, BYOL 74.4 on ResNet-50 linear accuracy — reproduced in Figure 10.12), so what distinguishes them is *what you must buy* to make them stable.

Which one should I use?

- **Small batches, want simple code** → SimSiam: the cheapest robust choice.
- **Large batches anyway** (e.g. image–text-scale loaders, Chapter 11's data engines) → Barlow Twins or VICReg. Their anti-collapse terms are easy to monitor: a run going bad shows up as the variance or off-diagonal statistics drifting *before* accuracy drops.
- **Already have an EMA teacher in the code** (for distillation, semi-supervised learning, or DINO, Chapters 11–12) → BYOL is nearly free.
- **Must rule collapse out by design** → negatives. They are the best-understood, most theoretically grounded mechanism — the fallback especially for small models, where family (b)'s asynchrony is hardest to tune.

10.9 Research frontier: collapse as a black box

Research Frontier

Why exactly does stop-gradient work? Despite its practical importance, collapse avoidance in asymmetric joint-embedding methods is not a closed theoretical question. Established by experiment: stop-gradient is indispensable for SimSiam, the predictor is indispensable for SimSiam and for BYOL, though BYOL's EMA makes it more forgiving of how the predictor is trained, and the constant solution remains a formal optimum of

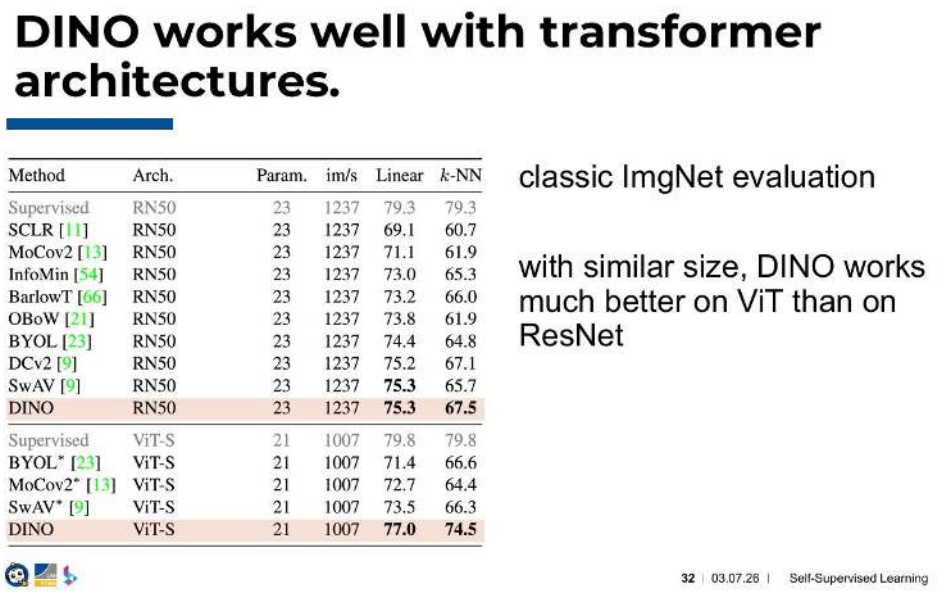


Figure 10.12. The DINO-era comparison table from which this chapter quotes its accuracy numbers. The upper block lists ResNet-50 models: a supervised baseline (79.3, greyed out) and two years' worth of self-supervised methods — SimCLR ("SCLR"), MoCo v2, InfoMin, Barlow Twins ("BarlowT"), OBoW, BYOL, DeepCluster v2 ("DCv2"), SwAV, and DINO — each reporting linear-probe and k-NN accuracy on frozen ImageNet features. Two readings matter for this chapter. First, the negative-free methods sit *inside* the same accuracy band as the negative-based ones (BYOL 74.4 and Barlow Twins 73.2 sit just above SimCLR's 69.1 and MoCo v2's 71.1): deleting negatives cost nothing in final accuracy, which is why the chapter compares mechanisms rather than leaderboards. Second, the lower block (ViT-S rows) is Chapter 11's hook: methods tuned for ResNets *drop* on ViTs (BYOL 74.4 → 71.4) while DINO *rises* (75.3 → 77.0) — the table the lecturer uses to argue that self-distillation pairs naturally with transformers. The starred ViT rows are DINO-paper re-implementations of methods whose original results were ResNet-only.

How to read it: Read the Linear column of the upper block top to bottom and watch how flat it is: about six points separate the simplest in-batch-negative method (SimCLR, 69.1) from the best (DINO, 75.3). All the differentiation lives in the machinery column, which is why Table 10.1 reports machinery instead.

Source: Lecture 07, slide 33 (table numbers from the DINO paper).

the symmetrized loss even when training does not find it. Interpretations offered — EM-like alternation (Chen and He, 2021), predictor-as-learned-optimizer, predictor-induced local predictability, power-iteration/eigenspace views of stop-gradient dynamics — each reproduce some ablations and fail others. Family (c)'s answer is rigorous but narrow: it proves that *if* the decorrelation statistics are enforced, collapse is excluded, and leaves the optimization dynamics unanalyzed. Open questions of direct relevance to practice: (i) necessary and sufficient conditions on the predictor h (width, depth, learning rate) for stability; (ii) whether small-model ViT training inherits the same basin geometry (Chapter

11's DINO suggests yes with centering/sharpening as the correspondingly-tuned family-(b/c) hybrid); (iii) a unified "collapse certificate" — a measurable statistic that predicts imminent collapse before probe accuracy degrades. Partial candidates exist (embedding-rank estimators, whitening-matrix condition numbers, predictor-loss forensics), but none is standard yet.

10.10 Summary

The chapter in one paragraph. A loss that only pulls views together can be minimised by a constant encoder — collapse. InfoNCE's negatives are *one* way to rule that out, not the only way. Three families work:

(a) **explicit repulsion** — negatives (Chapter 9);
(b) **asymmetry** — stop-gradient + predictor (SimSiam), plus an EMA teacher (BYOL);
(c) **statistics terms** — Barlow Twins' identity cross-correlation target, VICReg's variance–covariance–invariance triad.

Their final accuracies are remarkably close, so choose by your constraints: batch size, existing code, how easy failures are to diagnose, and whether you prefer machinery that must be tuned or loss terms whose failure shows up in the numbers. The next chapter meets collapse again, inside self-distillation: DINO solves it with yet another mix of ingredients and ends up teaching the Vision Transformer its best features.

Check Your Understanding

Check Your Understanding

1. **Level 1.** State the collapse solution of the SimSiam loss (10.4) and its value.
2. **Level 1.** In (10.2), what is $\mathcal{L}_{\text{NCE}}$ under collapse, and why is that *high*?
3. **Level 1.** Which two architectural ingredients does SimSiam require that a plain siamese network does not?
4. **Level 2.** Compute $D(\boldsymbol{p}, \boldsymbol{z})$ for $\boldsymbol{p} = (1, 0, 0)^\top$, $\boldsymbol{z} = (0, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})^\top$.
5. **Level 2.** In the Barlow Twins loss, which term fails under dimensional collapse and which under complete collapse? Explain.
6. **Level 2.** Explain why Barlow Twins is more batch-size-sensitive than SimSiam despite both avoiding negatives.
7. **Level 3.** Walk through one gradient step of SimSiam with and without sg on a 2-D embedding, showing which parameter gets which update in each case.
8. **Level 3.** Why is "stop-gradient prevents collapse" incomplete even though the slide 13 experiment shows it alone separating success from failure?

9. **Level 3.** Write down the EMA update of BYOL and explain, in your own words, how it might share part of the predictor's job.
10. **Level 4.** Show informally that if both views' embeddings agree and each embedding dimension has non-zero variance, then the encoder cannot be constant; what additional property is needed to exclude dimensional collapse?
11. **Level 4.** Barlow Twins minimizes $\|\mathcal{C} - \boldsymbol{I}\|_F^2$-style error; VICReg splits the same goal into three per-branch terms. Give one practical advantage of each design.
12. **Level 4.** Using the comparison table, pick a method for training a ResNet-50 on 8 GPUs with maximum feasible batch 512 and justify.
13. **Level 5.** Sketch an experiment that would test the EM-alternation interpretation of SimSiam against the predictor-as-optimizer interpretation.
14. **Level 5.** Propose a "collapse certificate" statistic computable from a training batch and argue why it should fire before probe accuracy drops.

Solution sketches.

- *1.* Constant encoder $\boldsymbol{z}_i = \boldsymbol{c}$, $\boldsymbol{p}_i = \boldsymbol{c}$; normalized cosine similarity is 1, so $D = -1$ and $\mathcal{L} = -1$.
- *2.* All logits equal $1/\tau$; softmax uniform over $K{+}1$; $\mathcal{L}_{\text{NCE}} = \log(K{+}1)$, the chance-level value: the positive is no more likely than any negative, so the loss cannot drop below it.
- *3.* A prediction MLP on one branch only and stop-gradient on the target branch.
- *4.* Both vectors unit norm; dot $= 0$; $D = 0$, halfway between the best value (-1) and the worst $(+1)$; the gradient still pulls $\boldsymbol{p}$ toward $\boldsymbol{z}$.
- *5.* Dimensional collapse inflates off-diagonal $\mathcal{C}_{ij}$, so the redundancy term fires; complete collapse makes the correlation matrix degenerate/undefined and typically destroys per-dimension variance that family (c) terms require.
- *6.* $\mathcal{C}$ is a second-order statistic estimated from the batch; small batches give noisy, biased correlation estimates — SimSiam's loss is per-sample and only needs two views per image.
- *7.* With sg: only the prediction branch gets gradient toward the frozen target. Without: both embeddings receive reflective gradients pulling them together along the loss surface, enabling the constant solution.
- *8.* Because collapsed solutions remain global optima of the stop-gradient loss; the predictor's input-conditional mapping is what makes non-constant solutions learnable.
- *9.* $\boldsymbol{\phi} \leftarrow \boldsymbol{m}\boldsymbol{\phi} + (1 - \boldsymbol{m})\boldsymbol{\theta}$; the slowly drifting target cannot instantaneously agree with the student, injecting the same asynchrony the predictor provides in SimSiam.
- *10.* A constant embedding contradicts unit variance; to exclude rank-deficient collapse one also needs cross-dimension decorrelation (or uniformity) — e.g. the off-diagonal / covariance terms.

- *11.* Barlow Twins: single hyperparameter, single interpretable matrix target. VICReg: per-branch statistics, more batch-robust, each failure mode's term independently monitorable.
- *12.* SimSiam: batch-insensitive, no queue/EMA machinery to tune, robust at 256–512 per its ablations; alternatives fail or need workarounds.
- *13.* E.g. train with the predictor frozen at random, or with a predictor refreshed/reset on a schedule: EM predicts stability tracks target-refresh dynamics; the optimizer view predicts instability when the predictor's optimization quality is perturbed.
- *14.* Effective rank / stable rank of the batch embedding matrix, or mean per-dimension variance; both hit degenerate values when collapse is incipient, before downstream metrics respond.

Concept Map — how this chapter connects

Builds on: Chapter 2 (cosine similarity, EMA, information theory); Chapter 8 (augmentation families, invariance vs. covariance); Chapter 9 (InfoNCE, SimCLR, MoCo, projection heads, uniformity–alignment view of contrastive quality).
This chapter: collapse as the central obstruction; the three mechanism families; SimSiam, BYOL, Barlow Twins, VICReg.
Leads to: Chapter 11 (DINO/iBOT/DINOv2 – self-distillation facing the same collapse problem with centering + sharpening + EMA; KoLeo as a uniformity-flavored family-(c) term), Chapter 12 (EMA teachers in self-training), Chapter 15 (competing theories of why joint-embedding SSL learns at all), and the comparisons of Chapter 16.

CHAPTER 11

Self-Distillation and the DINO Family

> "We thus also evaluate the quality of features with a simple weighted nearest neighbor classifier, as finetuning evaluations are sensitive to hyperparameters with large variance between runs."
>
> — Caron et al., *Emerging Properties in Self-Supervised Vision Transformers*, ICCV 2021

In this chapter

11.1 Self-distillation, and what this chapter leaves to Chapter 12 516
11.2 The joint-embedding family, diagrammed 517 • SimSiam: the minimal anti-collapse recipe • BYOL beside SimSiam: what the EMA teacher buys • Barlow Twins: collapse policed by a matrix
11.3 DINO: self-distillation with no labels 521 • Architecture and loss • Intuition: what DINO is really asking for • Two collapse modes, two counter-measures
11.4 Evaluating DINO: k-NN and emergent attention .... 528 • The result table: DINO prefers Transformers • Why k-NN was a novel evaluation • Emergent patch-level structure from an image-level loss • Ablations that matter: multi-crop
11.5 DINOv2: better data, two more losses ... 532 • The iBOT loss: masked-token DINO • The KoLeo regularizer: spread out the batch • Results: matching weak supervision without text
11.6 Scaling with data engines: DINOv3 and beyond 536 • The size progression, and why it forces a new chapter of the story • Curated data versus crawl data • The DINOv2 pipeline, stage by stage • The DINOv3 pool: the same engine, a hundred times bigger • Balancing the pool: hierarchical k-means (Vo et al.)
11.7 The BERT interlude: masked-token modelling for language ..... 544 • Setup and the two pre-training tasks • Bidirectional versus causal objectives • Results: frozen features versus finetuning

Chapter roadmap

Before you start. You need Chapter 10's collapse problem and the stop-gradient / EMA-teacher machinery, plus the vision transformer (Section 2.2.3) and softmax with temperature (Section 2.5.4). Those four pieces are literally all DINO is made of.

The idea in one sentence. Two copies of the same network look at two different crops of the same photo, and the slower copy's answer is used as the target for the faster copy — with no labels anywhere.

Why the word "distillation". Normally you distil a big trained teacher into a small student. Here there is no trained teacher: the teacher *is* the student, a few thousand gradient steps in the past. Nothing knowledgeable exists at the start, and yet the arrangement bootstraps. Why that works rather than collapsing into a constant is the central question of the chapter.

What we cover.

- DINO itself: multi-crop, the EMA teacher, and the two small operations — *centring* and *sharpening* — that stop the whole thing degenerating.
- The famous emergent result: attention maps that segment objects although nobody ever supplied a segmentation.
- iBOT and DINOv2: what happens when you bolt masked-patch prediction (Chapter 5) on top of a joint-embedding loss.
- DINOv3 and data engines: the shift from better *losses* to better *data*, at a scale where curation is the research.
- A BERT interlude, because masked-token modelling in language is the same idea and got there first.

Afterwards you will be able to explain exactly what centring and sharpening each prevent, and what happens if you remove either; read a DINO-family ablation table and know which component each row is testing; and say why these features dominate frozen-feature evaluation while MAE dominates fine-tuning.

Chapter 10 stripped negatives out of joint-embedding learning and found collapse waiting underneath every simplified objective. Each escape route there was architectural: stop-gradient, a predictor, an EMA target, or an explicit anti-collapse statistic. This chapter tells the next part of the story, in which the same escape routes reappear — reorganized under a single, older idea: *distillation*. Instead of pulling embeddings toward each other with cosine similarity, a *student* network is trained to match the output *distribution* of a second, gradient-free *teacher* network that sees a different view of the same image. There are no labels; the teacher is the label. This is *self-distillation*, and its flagship is DINO (Caron et al., 2021) — the paper that made self-supervised Vision Transformers competitive, produced object-segmentation-like attention maps for free, and then scaled, through DINOv2 (Oquab et al., 2024) and DINOv3 (Siméoni et al., 2025), into the argument that what limits self-supervised learning today is not objectives but *data engineering*.

Section 11.1 fixes terminology and bounds what this chapter covers (general knowledge distillation belongs to Chapter 12). Section 11.2 revisits SimSiam, BYOL, and Barlow Twins — owned by Chapter 10 — as diagrams, because one clean look at the three pipes is worth more than another page of prose. Section 11.3 develops DINO in full: architecture, centering,

sharpening, EMA, multi-crop, pseudocode, and the two collapse modes those pieces decorrelate. Section 11.4 covers evaluation and the emergent properties. Section 11.5 adds DINOv2's two extra losses. Section 11.6 is the scaling story: curated versus crawl data, the DINOv2 curation pipeline, DINOv3's three-part pool, and the hierarchical k-means balancing algorithm of Vo et al. (2024). Section 11.7 honours the lecture's BERT interlude as masked token modelling beyond vision — the conceptual parent of the iBOT patch loss.

11.1 Self-distillation, and what this chapter leaves to Chapter 12

Knowledge distillation (Hinton et al., 2015) trains a small *student* network to match the softened output distribution of a large, already-trained *teacher*, on the observation that the teacher's probabilities over wrong classes — its "dark knowledge" — carry more information than a one-hot label. Distillation is the classic tool for model compression: the lecture's example (Lecture 08, slide 16) is Gemini 1.5 Flash, trained by distillation from Gemini 1.5 Pro and running roughly three times faster per output character (Gemini Team, Google, 2024). That supervised, label-aware story — soft targets, ensembles as teachers, multi-teacher methods such as DUNE (Sariyildiz et al., 2025) — is developed in Chapter 12 and deliberately kept short here.

This chapter's subject is the special case where no labels, no task head, and no separately trained teacher exist. The teacher is a slowly-evolving copy of the student itself, and the "knowledge" being distilled is whatever view-invariant structure the system has discovered so far.

Key Concept — Self-distillation

Self-distillation is joint-embedding learning in which a student network g_{θ_s} is trained to match the output distribution of a teacher network g_{θ_t} applied to a different view of the same input, where the teacher (i) is never updated by gradient descent — it works under stop-gradient, and its parameters are an exponential moving average (EMA) of the student's — and (ii) sees only a subset of the views (in practice the global crops). The two views play, at each step, the roles that labels and predictions play in supervised distillation. DINO (Caron et al., 2021) named the recipe "distillation with no labels."

Why call it distillation? Three ingredients carry over exactly from classic distillation, as the lecture stresses (Lecture 08, slide 14):

	Hinton-style distillation	**DINO**
Target	full softmax distribution	full softmax distribution
No gradient to target	frozen teacher	stop-gradient
Smoothing	temperature	EMA over weights

Only the *source* of the teacher changes: here it is the student's own past. And that brings back Chapter 10's collapse problem in a new form — now it can happen through the *softmax outputs* instead of the raw embeddings. DINO's centering and sharpening (Section 11.3) are best seen as the two halves of the fix.

11.2 The joint-embedding family, diagrammed

Chapter 10 analysed collapse and the three mechanism families in prose and equations. Before adding DINO, we redraw the three representative pipelines as diagrams — per the lecture's own visual presentation (Lecture 07, slides 12–14 for SimSiam; slide 33 for the comparison table in which BYOL and Barlow Twins appear as rows) — so that the rest of the chapter can point at pictures instead of repeating derivations.

11.2.1 SimSiam: the minimal anti-collapse recipe

Figure 11.1 shows SimSiam (Chen and He, 2021):

1. two views of one image pass through a shared encoder f (backbone plus projection MLP);
2. a small predictor MLP h sits on *one* branch only;
3. each prediction is matched, by negative cosine, to the *other* branch's projection, with the gradient blocked on that side:

$$\mathcal{L} = \tfrac{1}{2}\, D(\boldsymbol{p}_1, \mathrm{sg}(\boldsymbol{z}_2)) + \tfrac{1}{2}\, D(\boldsymbol{p}_2, \mathrm{sg}(\boldsymbol{z}_1))\,.$$

Here $\boldsymbol{z}_i = f(\boldsymbol{x}_i)$ are the projections, $\boldsymbol{p}_i = h(\boldsymbol{z}_i)$ the predictions, D the negative cosine similarity of (10.4), and sg (stop-gradient) does nothing on the forward pass and blocks the gradient on the backward pass. So gradients flow into the branch with h, never into the target branch. In the lecture's reading (slides 16–17), each branch takes a turn as student, but the teacher side never backpropagates.

Why does Figure 11.1 matter? See Figure 11.2:

- **Stop-gradient removed:** training *looks* successful — the loss dives to its minimum, -1, within a few epochs — but k-NN accuracy on the frozen features stays at chance. The encoder has found the constant function: Chapter 10's collapse.
- **Stop-gradient (and predictor) in place:** the loss falls slowly, and k-NN accuracy climbs to about 50% over 100 epochs.

Slide 13's caption is worth quoting: "unlike supervised training, low loss is not always good since loss does not directly model the downstream task." In self-supervised training, the loss curve tells you about the objective, not about the representation.

Two practical failure modes are worth restating here because the SimSiam section of this lecture is where the course teaches them. First, *collapse sensitivity is architectural, not*

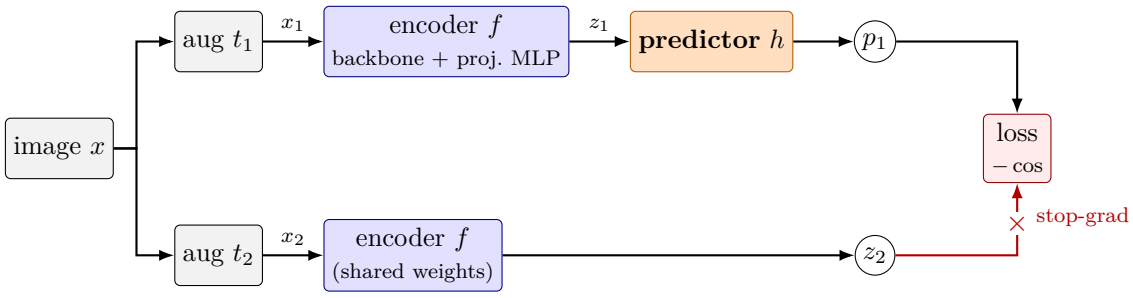


Figure 11.1. SimSiam twin-view pipeline (redrawn from Lecture 07, slides 12 and 14). The two asymmetries of Chapter 10 are marked: the *predictor* h (orange) exists on one branch only, and the target branch reaches the loss through *stop-gradient* ($\times$, red). The full loss additionally symmetrizes the diagram by swapping the branch roles.

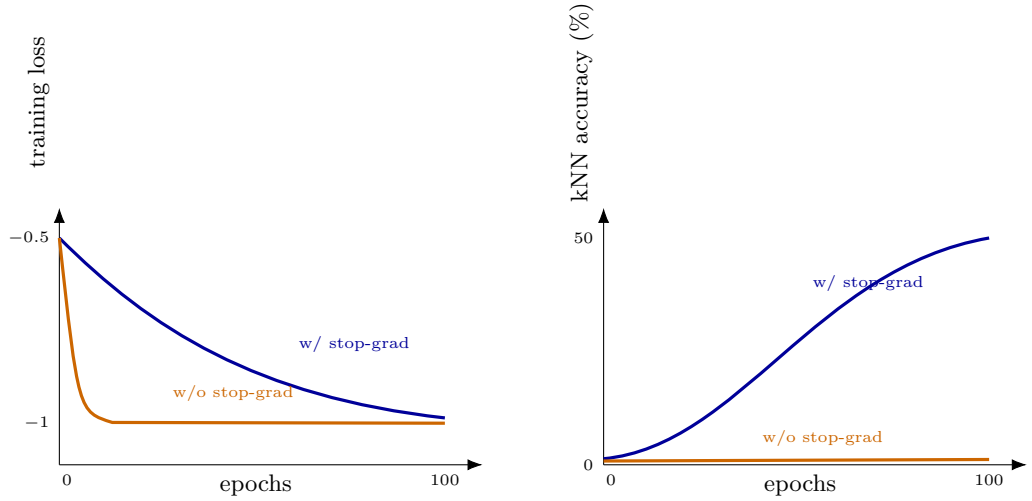


Figure 11.2. Collapse versus successful training, schematic after Lecture 07 slide 13 (source experiment: Chen and He, 2021). *Left:* with stop-gradient the training loss decays slowly from about -0.5 toward -0.9; without it the loss collapses to the trivial optimum -1 almost immediately. *Right:* frozen-feature k-NN accuracy rises steadily to roughly 50% with stop-gradient, and flatlines at chance without. The two diagnostics point in opposite directions exactly when collapse occurs.

statistical: nothing in the loss signals approaching collapse — you must watch a downstream probe. Second, SimSiam's celebrated batch-size robustness is one-directional: batch 256 works precisely because the method sources no negatives from the batch, whereas family-(a) in-batch methods lose negatives as the batch shrinks and family-(c) methods, relying on batch statistics, lose estimation accuracy — both degrade.

Failure Modes

SimSiam.

- **Stop-gradient removed:** loss $\to -1$ in a few epochs, features constant, kNN at chance (Figure 11.2).
- **Predictor removed or identity:** near-collapse in the paper's ablations even with stop-gradient intact — the two asymmetries are jointly load-bearing.
- **Batch-size interaction:** SimSiam is insensitive *downward* (batch 256 works, since no negatives are sourced from the batch), but this is exactly why silent regressions go unnoticed at small batch: the training loss looks fine while the probe degrades. Monitor kNN accuracy per epoch, never the loss.
- **Predictor learning rate:** the EM-like reading of Chapter 10 suggests h should track f's drift; standard implementations use a fixed (non-decayed) learning rate for the predictor.

11.2.2 BYOL beside SimSiam: what the EMA teacher buys

BYOL (Grill et al., 2020) is SimSiam's immediate parent. The branch layout, predictor, and stop-gradient are identical; the one structural difference is that the target network is not a shared-weight copy but an *exponential moving average* of the online network's weights,

$$\boldsymbol{\phi} \leftarrow m\,\boldsymbol{\phi} + (1-m)\,\boldsymbol{\theta}, \tag{11.1}$$

where $\boldsymbol{\theta}$ are the online parameters, $\boldsymbol{\phi}$ the target parameters, and $m \in [0, 1)$ a momentum coefficient close to 1 (e.g. 0.996, typically increased toward 1 over training). Figure 11.3 places the two side by side. The EMA appears to share part of the predictor's stabilizing role (Chapter 10), and it is the ingredient DINO will keep while discarding the predictor.

The failure signature flips with the ingredient. In SimSiam, remove the predictor and training breaks; in BYOL, the delicate part is the EMA itself.

Failure Modes

BYOL.

- **EMA momentum too small:** the target chases the online network, the asynchrony protecting against collapse vanishes, and training approaches the no-EMA regime — which is unstable for BYOL's backbone setups.
- **EMA momentum frozen near 1 too early:** the target stalls; the online network overfits to a stale target and progress stops.
- **Predictor tricks:** because the EMA can partly mask a poorly trained predictor, a broken predictor may show up only as a few points of lost accuracy rather than an obvious failure. Ablate the two stabilizers independently, not jointly.

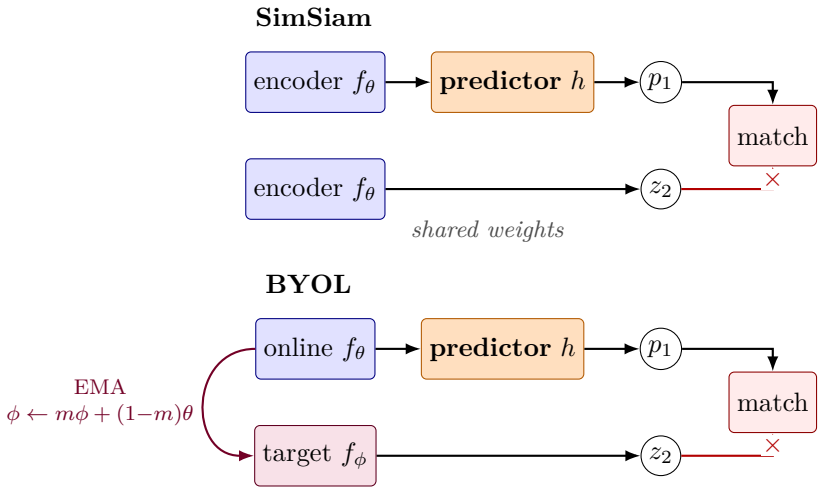


Figure 11.3. BYOL next to SimSiam (architectures per Lecture 07, slides 11–17; comparison row of slide 33). Same predictor, same stop-gradient, same matching loss; the single difference is the *target network*: shared weights in SimSiam, an EMA of the online weights in BYOL (purple).

- **Normalization folklore:** the claim that "BYOL secretly uses BatchNorm statistics as negatives" is not supported as stated (group-norm variants still train), but removing batch normalization does interact with stability; treat normalization choice as a stability hyperparameter, not a conspiracy.

11.2.3 Barlow Twins: collapse policed by a matrix

The third family needs no asymmetry. Barlow Twins (Zbontar et al., 2021) passes two views through the same encoder and pushes the *cross-correlation matrix* between the two batches of embeddings towards the identity:

- **diagonal** $\to 1$: the two views agree, dimension by dimension;
- **off-diagonal** $\to 0$: different dimensions carry different information, so no dimensional collapse.

Figure 11.4 shows the target; Chapter 10 works a 3×3 example by hand, loss terms included. We repeat the picture here because it is the clearest view of an "explicit anti-collapse term": the loss is the distance to a *drawn* target matrix, and dimensional collapse shows up directly as ink off the diagonal.

With the three pipes drawn, notice what none of them does: none has a *distributional* target. SimSiam and BYOL match vectors by cosine similarity; Barlow Twins matches a correlation matrix to the identity. DINO's move — next section — is to make the target a categorical distribution over K "prototype" dimensions, which is precisely the step that turns the whole construction into distillation.

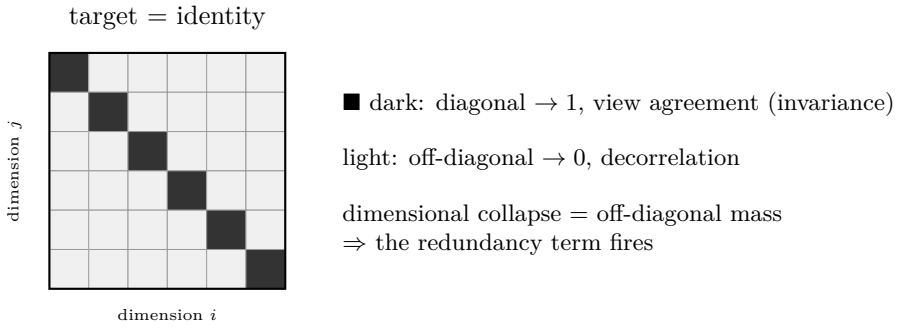


Figure 11.4. Barlow Twins objective drawn as a matrix target (teaser slide, Lecture 06 slide 72; method of Zbontar et al., 2021). The empirical cross-correlation matrix between the two views' embedding batches (entries defined in Chapter 10) is trained toward the identity: on-diagonal to 1, off-diagonal to 0. Collapse is not avoided implicitly; it is penalized entry by entry.

11.3 DINO: self-distillation with no labels

Model at a glance: DINO Caron, Touvron, Misra, Jégou, Mairal, Bojanowski & Joulin, 2021

In one line	A student network is trained to match the output of an EMA copy of itself, across different crops of the same image, with centring and sharpening applied to the teacher to prevent collapse.
Problem it solves	Getting joint-embedding quality without negative pairs, without large batches, and without a memory bank — and, as it turned out, getting object segmentation for free as a side effect.
Key intuition	The teacher is the student's own past. It changes slowly, so it is a *stable* target; it is not independently knowledgeable, so there is nothing to distil except consistency — and consistency across crops of one photo turns out to be a strong enough demand to produce semantics.
Architecture	A ViT (or ResNet) backbone plus a projection head ending in a softmax over K prototype dimensions. Student and teacher share the architecture; the teacher's weights are an EMA of the student's.
Multi-crop	Two large "global" crops and several small "local" crops. The teacher sees only the global crops; the student sees all of them, and must match the teacher's global answer from a local patch. This local-to-global demand is where much of the performance comes from.
Loss	Cross-entropy between the sharpened, centred teacher distribution and the student distribution, summed over crop pairs.
Centring	Subtract a running mean from the teacher's logits. Prevents any one prototype dimension from dominating every image.
Sharpening	Use a lower temperature for the teacher than the student. Prevents the teacher's output from flattening to uniform.
Why both	They fail in opposite directions. Centring alone drives the output towards uniform; sharpening alone drives it towards one dominant dimension. Each is the antidote to the other's failure, and removing either one collapses training.

Emergent property	The attention maps of the `[CLS]` token segment the foreground object without any segmentation supervision — the single most reproduced figure in the modern SSL literature.
Evaluation	The paper deliberately reports k-NN accuracy as well as linear probing, because fine-tuning numbers vary too much between runs to compare methods honestly (this chapter's epigraph).
Leads to	iBOT (add masked-patch prediction), DINOv2 (add curated data at scale), DINOv3 (add a data engine).

11.3.1 Architecture and loss

DINO (Caron et al., 2021) keeps the EMA teacher of BYOL and the stop-gradient of everything, deletes the predictor, deletes the negatives, and replaces the vector-matching loss by a cross-entropy between softmax distributions. Figure 11.6 shows the full pipeline, including multi-crop. The student $g_{\boldsymbol{\theta}_s}$ and teacher $g_{\boldsymbol{\theta}_t}$ share one architecture — a backbone (ResNet or ViT (Dosovitskiy et al., 2021)) followed by a projection head outputting K logits — and produce, for an input view $\boldsymbol{x}$, K-dimensional distributions

$$\boldsymbol{p}_s(\boldsymbol{x}) = \mathrm{softmax}\big(g_{\boldsymbol{\theta}_s}(\boldsymbol{x})/\tau_s\big), \qquad \boldsymbol{p}_t(\boldsymbol{x}) = \mathrm{softmax}\Big(\big(g_{\boldsymbol{\theta}_t}(\boldsymbol{x}) - \boldsymbol{c}\big)/\tau_t\Big), \tag{11.2}$$

where $\tau_s > 0$ and $\tau_t > 0$ are the student and teacher *temperatures* (smaller τ sharpens the distribution), and $\boldsymbol{c} \in \mathbb{R}^K$ is the *center*, an EMA estimate of the teacher's mean output that is subtracted before the teacher's softmax (*centering*). The loss between one teacher view and one student view is the cross-entropy $H(\boldsymbol{p}_t, \boldsymbol{p}_s)$, where $H(\boldsymbol{a}, \boldsymbol{b}) = -\sum_{k=1}^{K} a_k \log b_k$ with a_k, b_k the k-th components of the two distributions. Gradients update the student only; the teacher's parameters follow by EMA,

$$\boldsymbol{\theta}_t \leftarrow \lambda\, \boldsymbol{\theta}_t + (1-\lambda)\, \boldsymbol{\theta}_s, \tag{11.3}$$

where $\lambda \in [0, 1)$ is the teacher momentum (e.g. ramped from 0.996 toward 1 through training), and the center follows its own EMA,

$$\boldsymbol{c} \leftarrow m\, \boldsymbol{c} + (1-m)\, \frac{1}{B} \sum_{i=1}^{B} g_{\boldsymbol{\theta}_t}(\boldsymbol{x}_i), \tag{11.4}$$

where m is the centre momentum, B the batch size, and the average is over the teacher's raw outputs on the current batch.

Multi-crop. The two networks see different crops:

- the **teacher** sees only the two *global* crops (large, 224×224);
- the **student** sees those *plus* several *local* crops (small, 96×96 in DINO; 98×98 in DINOv2, whose patches are 14 pixels).

The loss pairs every student view with every teacher view, but never a view with itself (Figure 11.5 draws all of this on one picture). Small crops are cheap, and asking "does this small piece match the whole picture?" gives a useful local-to-global signal. The idea comes from SwAV (Caron et al., 2020); the slides do not credit it, but Chapter 10's overview of the three anti-collapse families (Section 10.2) introduces SwAV. The standard augmentations (colour jitter, Gaussian blur, solarisation) apply throughout (Lecture 07, slide 26).

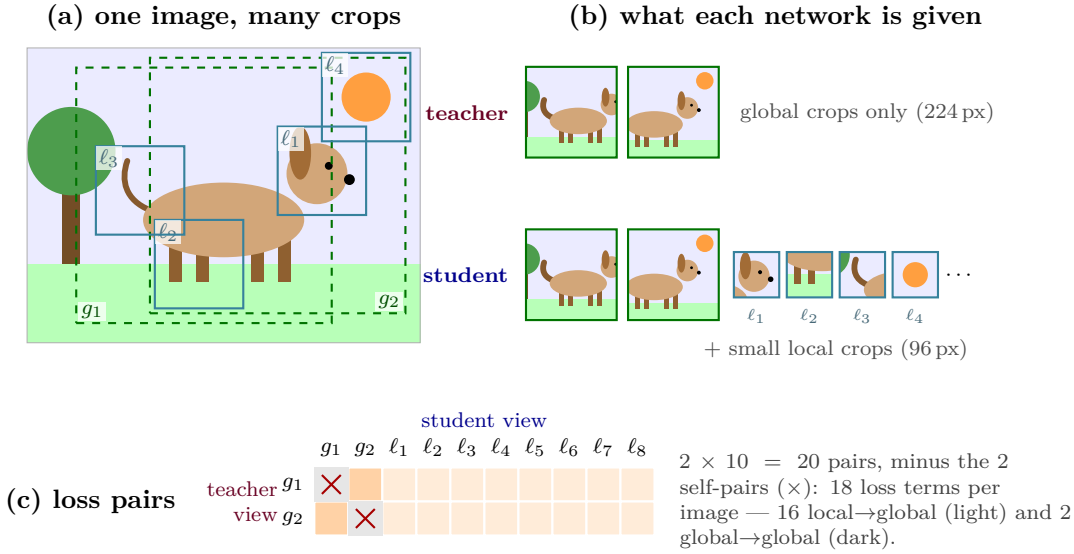


Figure 11.5. Multi-crop on one (drawn) image. (a) Two large *global* crops (green, dashed) each cover most of the scene; small *local* crops (cyan) see only a piece of it — the dog's head, its legs, its tail, a patch of sky. (b) The crops as the networks receive them: the teacher gets only the two global crops, the student gets those plus every local crop (DINO uses 8 to 10; four are drawn). (c) The loss compares each teacher output with each student output from a *different* crop; the grid has one cell per pair, darker cells are global–global.

How to read it: Look at the local crop ℓ_1: it shows only a head, yet the student must give the same distribution the teacher gives for the whole scene in g_1 or g_2. The crop ℓ_4 is only sky: DINO has no idea which crops contain the object, so some targets are hard or even unfair — the loss is an average over many such pairs, not a promise that every one is solvable.

11.3.2 Intuition: what DINO is really asking for

The equations above are short, but three questions usually remain. What is the student actually learning to do? What are those K output numbers? And how can a network learn anything from a copy of itself?

In plain words — guess the whole from a part

> Show the teacher the whole photo of a dog and let it say, in its own private vocabulary, what the photo is about. Then show the student only a small patch — an ear, a bit of

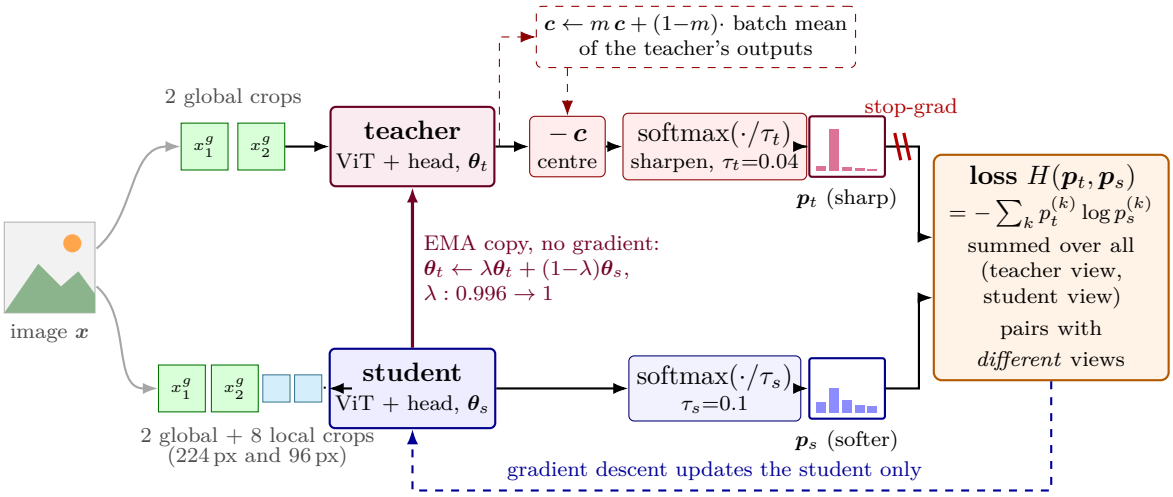


Figure 11.6. The DINO training step (redrawn from Lecture 07 slides 25–26; method of Caron et al., 2021). One image gives two large *global* crops (green) and several small *local* crops (cyan). The **teacher** (top) sees only the global crops; the **student** (bottom) sees all of them. The teacher's outputs are *centred* (the centre $\boldsymbol{c}$ is itself a running average of the teacher's outputs, dashed red) and *sharpened* by a low temperature; the student uses a higher one. The loss asks each student distribution to match each teacher distribution from a different view. Gradients reach the student only (blue); the teacher is never trained directly, it is a slow moving average of the student (purple). Temperatures are the paper's defaults; τ_t is warmed up from 0.04 to 0.07.

How to read it: Follow one crop down each row. A local crop only ever enters the bottom row, so the student must predict, from a small piece, the answer the teacher gives for the whole picture. Then find the two update arrows: blue (gradients) touches only the student, purple (EMA) is the only way the teacher ever changes.

> fur — and ask it to give the *same* answer. The only way to answer well is to know that this kind of ear usually belongs to this kind of animal. No label ever says "dog"; the demand "a part must say what the whole says" is enough to force the network to learn what objects are made of.

The K outputs are soft cluster assignments. The head ends in a softmax over K numbers (K = 65,536 in DINO). Nothing tells the network what the K entries mean; it invents them. The easiest way to read $\boldsymbol{p}_t$ is as "how strongly this image belongs to each of K self-made groups", like the pseudo-labels of DeepCluster (Chapter 3), except that the groups are learned on the fly instead of by running k-means. With that reading, the two anti-collapse operations have plain meanings:

- **Centering** keeps all the groups in use. A group that grabs every image would be a useless label, so its running average is subtracted until it stops winning by default.
- **Sharpening** makes each image commit to a few groups. A teacher that says "a bit of everything" gives the student nothing to learn, so its low temperature turns small preferences into clear ones.

Together they ask for what a good clustering needs: every image confidently in some group, and the groups shared out across the data.

Why a copy of yourself can teach you. The teacher is not wiser than the student in any deep sense; it is a moving average of the student's recent versions. That average turns out to be a slightly *better* network than the current student. Averaging many nearby weight vectors smooths out the noise of individual gradient steps, much like averaging several models. Caron et al. (2021) report exactly this: throughout training the teacher scores higher on k-NN evaluation than the student it is built from. So the student is always chasing a target a little ahead of it. As it improves, the average improves too, and the chase continues. Two details keep the chase honest. The teacher moves slowly (λ close to 1), so it cannot follow the student into a sudden collapse. And no gradient flows through the teacher, so the student cannot make the loss small by dragging the target towards itself.

Step by step — one DINO training step

1. Cut each image into 2 global crops (224 px) and, say, 8 local crops (96 px), each with its own colour jitter, blur and flips.
2. Teacher: run the 2 global crops through $\boldsymbol{\theta}_t$, subtract the centre $\boldsymbol{c}$, divide by τ_t, softmax. These $\boldsymbol{p}_t$ are targets: no gradient flows back through them.
3. Student: run all 10 crops through $\boldsymbol{\theta}_s$, divide by τ_s, softmax.
4. Loss: cross-entropy $H(\boldsymbol{p}_t, \boldsymbol{p}_s)$ for every pair of one teacher view and one *different* student view, averaged.
5. Update the student by gradient descent on that loss.
6. Update the teacher by EMA (Eq. (11.3)) and the centre by its own EMA (Eq. (11.4)).

Worked Example — counting the loss terms

With 2 global and 8 local crops, the teacher produces 2 distributions and the student 10. Pairing each teacher view with each student view gives $2 \times 10 = 20$ pairs, minus the 2 pairs where a global crop would be compared with itself: 18 cross-entropy terms per image. Of these, 16 are local-to-global (a small patch must match the whole) and 2 are global-to-global (one large view must match the other). The self-pairs are dropped because matching a crop to itself is trivially easy and teaches nothing. Figure 11.5(c) draws these pairs as a grid. Most of the training signal is the local-to-global kind, which is why multi-crop matters so much (Section 11.4).

Everyday analogy — an apprentice and a slowly updated notebook

An apprentice keeps a notebook of how they described each photo, and updates it only slowly — a little of today's opinion mixed into a lot of last month's. Each day they are shown fragments of photos and must describe each fragment the way the notebook describes the whole photo. The notebook is their own past thinking, averaged, so it is

steadier than today's opinion; matching it from fragments forces them to understand what the photos contain.

Where the analogy breaks. A real notebook could be copied word for word. DINO's two safeguards — centering and sharpening — are what stop the apprentice from writing the same description for every photo, which would also match the notebook perfectly.

11.3.3 Two collapse modes, two counter-measures

Why are centering *and* sharpening both needed? Because the distributional target introduces two degenerate optima, and each operation blocks exactly one.

- **Collapse by domination.** If one output dimension wins the softmax regardless of input, teacher and student agree trivially and the cross-entropy is tiny. Centering prevents this: subtracting the running mean $\boldsymbol{c}$ means that if dimension k is systematically large, $\boldsymbol{c}_k$ grows until the advantage is removed. No dimension can dominate *on average*.
- **Collapse by uniformity.** If the teacher emits a near-uniform distribution for every input, matching it is again trivial and uninformative. Sharpening prevents this: a teacher temperature $\tau_t < \tau_s$ forces the teacher toward peaked, low-entropy outputs, so an exactly uniform teacher cannot be a fixed point of the loop.

Used alone, each causes the other's failure: sharpening alone invites single-dimension domination; centering alone softens the teacher toward uniformity. Together they pin the teacher to a middle regime — peaked but input-dependent — which is the whole trick. (Lecture 07 slide 25 states the mechanics; the complementarity is the analysis the slides compress.)

Worked Example — Centering and sharpening on a 3-way softmax

Let the teacher's raw logits on a batch be $\boldsymbol{g} = (2.0,\ 1.0,\ 0.0)^\top$ for every image (the worst case: the output depends on nothing), and suppose the running center is $\boldsymbol{c} = (0.5,\ 0.2,\ 0.1)^\top$.

Sharpened without centering, $\tau_t = 0.1$. The scaled logits g_k/τ_t are $(20, 10, 0)$; exponentiating gives $e^{20} \approx 4.85 \times 10^8$, $e^{10} \approx 2.20 \times 10^4$, $e^0 = 1$, so

$$\boldsymbol{p}_t \approx (0.99995,\ 0.00005,\ 0.00000).$$

A one-hot answer, confident, identical for every image: domination collapse. Cross-entropy against any student placing mass on dimension 1 is near zero; nothing here depends on the input.

Centered, mild temperature $\tau_t = 1.0$. Centered logits: $\boldsymbol{g} - \boldsymbol{c} = (1.5,\ 0.8,\ -0.1)$. Exponentiate: $e^{1.5} \approx 4.48$, $e^{0.8} \approx 2.23$, $e^{-0.1} \approx 0.90$, sum ≈ 7.61, so

$$\boldsymbol{p}_t \approx (0.589,\ 0.293,\ 0.119).$$

Still dominated by dimension 1 (every image produced the same $\boldsymbol{g}$), but watch the *center update*: (11.4) with momentum $m = 0.9$ and batch-mean output equal to $\boldsymbol{g}$ gives

$$\boldsymbol{c} \leftarrow 0.9\,(0.5, 0.2, 0.1) + 0.1\,(2.0, 1.0, 0.0) = (0.65,\ 0.28,\ 0.09).$$

Each step pushes $\boldsymbol{c}$ toward the batch mean; after a few dozen steps $\boldsymbol{c} \approx \boldsymbol{g}$, the centered logits vanish, and the teacher goes uniform. That is exactly where sharpening matters: centering removes only the *input-independent* part of the logits, and dividing by the small τ_t magnifies whatever *input-dependent* differences remain, so the teacher can sit at the uniform answer only if its logits are genuinely identical for every image. The two operations corner the dynamics: centering erases any constant lead, sharpening amplifies real differences. What survives is a peaked, input-dependent teacher. Figure 11.7 shows where this ends up once $\boldsymbol{c}$ has settled, on a batch of three images: only the combination gives each image its own confident answer.

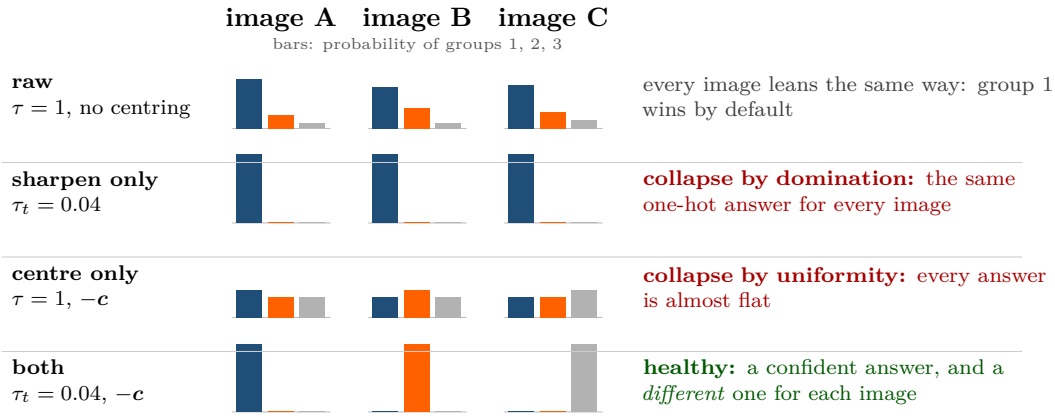


Figure 11.7. What centring and sharpening do to a batch of three images. The teacher's logits share a large common part, $(2, 1, 0)$ for every image, plus a small image-specific part: $+0.3$ on group 1 for image A, on group 2 for B, on group 3 for C. The centre $\boldsymbol{c} = (2.1, 1.1, 0.1)$ is the batch mean, where Eq. (11.4) settles. Every bar is an exact softmax of these numbers. *Sharpening only* blows up the common lead: all three images get the same one-hot answer. *Centring only* removes the common part, leaving differences of ± 0.3 that a temperature of 1 turns into nearly flat distributions. *Both*: centring removes what the images share, and the low temperature magnifies what is left — exactly the part that makes each image different.

How to read it: Read down one column, then across one row. Collapse is not a property of one distribution but of the whole batch*: in the two red rows the three columns look alike. Only the bottom row gives the student a target that depends on the image.*

Common Confusion

DINO loss vs. DINOv2 loss vs. iBOT loss. These names collide in the literature and on the slides (Lecture 07, slides 39–41). (i) *DINO loss* $\mathcal{L}_{\text{DINO}} = -\sum_k p_t^{(k)} \log p_s^{(k)}$ is computed on the **[CLS]** token's distribution over K prototypes — an image-level

objective; DINOv2 keeps it unchanged. (ii) *iBOT loss* $\mathcal{L}_{\text{iBOT}} = -\sum_i \sum_k p_{t,i}^{(k)} \log p_{s,i}^{(k)}$ is the *same form* computed per *masked patch token* i: masked image modelling with an online self-distilled tokenizer, not a new kind of cross-entropy (Zhou et al., 2022). (iii) "The DINOv2 loss" is not a single formula: it is the weighted sum $\mathcal{L}_{\text{DINO}} + \mathcal{L}_{\text{iBOT}} + \lambda \mathcal{L}_{\text{KoLeo}}$. Saying "DINOv2 trains with the DINO loss" undersells it; saying "DINOv2 replaced DINO's loss with iBOT" is wrong.

Failure Modes

DINO.

- **Center momentum mis-set:** in the center EMA (11.4), momentum m too small makes $\boldsymbol{c}$ track single batches — noisy centering destabilizes the teacher early; too large and domination persists for many steps before the center catches up. The interplay of center momentum with teacher temperature is the collapse dial of the method.
- **Teacher temperature too low:** the teacher goes near-one-hot early; the EMA loop self-confirms a constant winner. Classic symptom: the entropy of $\boldsymbol{p}_t$ collapses within the first epochs.
- **Teacher temperature too high:** the teacher stays nearly uniform; the student matches trivially; probe accuracy crawls — uniformity collapse without any warning from the training loss.
- **No warmup on the teacher temperature:** early training is the fragile phase; monitor the entropies of both distributions, not just the loss.

11.4 Evaluating DINO: k-NN and emergent attention

11.4.1 The result table: DINO prefers Transformers

DINO's results. The lecture shows one ImageNet table (Lecture 07, slide 33) with two columns — linear probe on frozen features, and k-nearest neighbours — and methods grouped by backbone. Table 11.1 reproduces a selection. Two rows carry the argument:

- **ResNet-50:** DINO gets 75.3 linear / 67.5 k-NN — a good method among good methods. It ties SwAV (Caron et al., 2020) on linear and beats it on k-NN, but, like BYOL and Barlow Twins, stays below supervised training.
- **ViT-S:** DINO jumps to 77.0 / 74.5, within about three points of supervised training on the same backbone. Meanwhile BYOL and SwAV (starred rows are re-implementations) do *worse* on ViT than on ResNet-50, and MoCo v2 (Chen et al., 2020c) improves by under two points.

In the lecture's words: "With similar size, DINO works much better on ViT than on ResNet." Self-distillation does not just *work* with Vision Transformers (Dosovitskiy et al., 2021) — it

Table 11.1. ImageNet-1k evaluation of frozen features (top-1 accuracy, %), after the table on Lecture 07 slide 33 (numbers from Caron et al., 2021; starred rows are ViT re-implementations of methods published with ResNet backbones). "Linear" is a linear probe on frozen features; "k-NN" is the weighted nearest-neighbour classifier of Section 11.4.

Backbone	Method	Linear	k-NN
ResNet-50	Supervised	79.3	79.3
ResNet-50	SimCLR (Chen et al., 2020a)	69.1	60.7
ResNet-50	MoCo v2 (Chen et al., 2020c)	71.1	61.9
ResNet-50	Barlow Twins (Zbontar et al., 2021)	73.2	66.0
ResNet-50	BYOL (Grill et al., 2020)	74.4	64.8
ResNet-50	SwAV (Caron et al., 2020)	75.3	65.7
ResNet-50	DINO (Caron et al., 2021)	75.3	67.5
ViT-S/16	Supervised	79.8	79.8
ViT-S/16	BYOL*	71.4	66.6
ViT-S/16	MoCo v2*	72.7	64.4
ViT-S/16	SwAV*	73.5	66.3
ViT-S/16	DINO	**77.0**	**74.5**

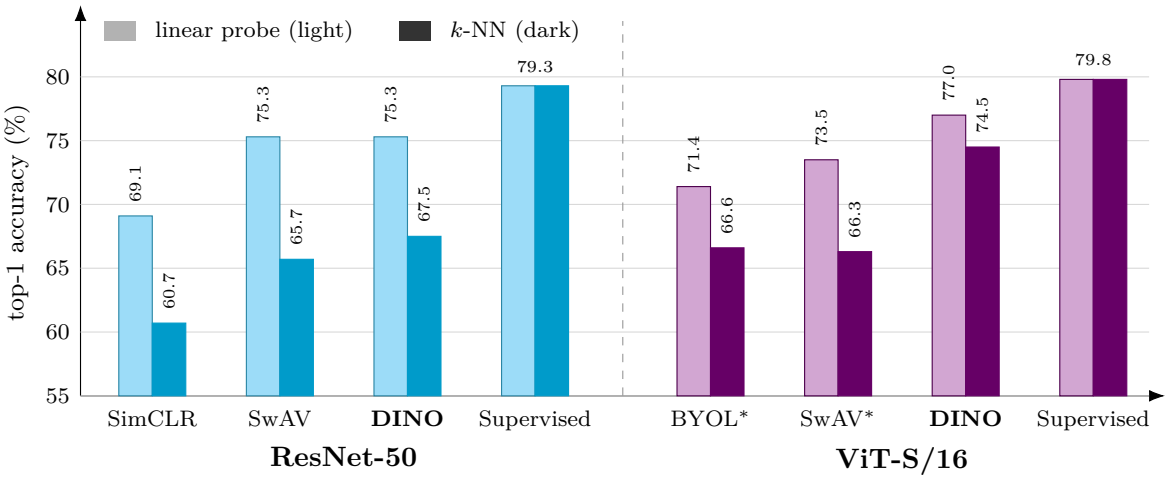


Figure 11.8. Frozen-feature evaluation on ImageNet-1k, redrawn from the table of Lecture 07 slide 33 (data: Caron et al., 2021). Light bars: linear probe; dark bars: k-NN. On ResNet-50, DINO ties but does not dominate the joint embedding field; on ViT-S/16 it overtakes every SSL competitor and its k-NN score nearly matches the linear probe's — evidence that the feature space itself is neighbourhood-coherent, not merely linearly separable. (Axial detailing: accuracies start at 55% to make the differences legible.)

benefits from them unusually. Chapter 5's masked autoencoders (He et al., 2022) show the same pattern from the reconstruction side, and together the two findings made the ViT the default SSL backbone.

Figure 11.8 draws the same numbers as bars, because the visual gap matters for what follows: on ViT-S, DINO's k-NN accuracy (74.5) is nearly as high as its linear-probe accuracy

(77.0) — a property none of the competitors shows. To appreciate why that observation changed evaluation practice, we need the novelty slide behind it.

11.4.2 Why k-NN was a novel evaluation

Why DINO reports k-NN, not just a linear probe. Slide 34 circles the k-NN column and quotes the paper (this chapter's epigraph): fine-tuning results "are sensitive to hyperparameters with large variance between runs". Compare the two ways of scoring frozen features.

Linear probe: train a linear classifier on top of the features. That training has its own learning rate, weight decay and so on, and changing those can move the score by several points. So you partly measure the probe, not only the features.

k-NN: no training at all. For each test image, find the k training images with the closest features and let them vote. It asks one simple question: *are images of the same class already sitting next to each other?*

DINO's 74.5% by k-NN means yes, to a remarkable degree. The training movie on slides 35–36 shows the same thing visually: a 2-D projection of the features, a grey blob at epoch 0, and separate coloured clusters — insects, birds, primates, dogs, vehicles — by epoch 300. That movie is the k-NN number, drawn.

Key Concept — Weighted k-NN evaluation on frozen features

Extract the frozen feature $\boldsymbol{z}$ of every training and test image; classify each test image by a temperature-weighted vote of its k nearest training features under cosine or Euclidean distance. The protocol has no trainable parameters, so it is reproducible per feature extractor, and it measures *neighbourhood structure* directly rather than linear separability after a fitted readout. Its weakness mirrors Chapter 2's: it is only as good as the distance metric, and it reports instance-level, not linear, separability.

Common Confusion

Linear probe vs. k-NN are not interchangeable. A high linear-probe score with a weak k-NN score means classes are linearly separable somewhere in the space but not locally clustered; a high k-NN score with a weak linear probe would signal class structure expressed nonlinearly (in practice, for DINO family models both are high, with k-NN close behind linear). Reporting only one hides which geometry your features actually have. The correct pairing also mirrors Chapter 2's tools: k-NN inherits the metric's curse of dimensionality, which is one reason careful papers always report a linear probe beside it.

11.4.3 Emergent patch-level structure from an image-level loss

The famous attention maps (slide 37, the paper's Figure 3). Take a ViT-S/8 trained with DINO and colour each last-layer attention head's map differently. Different heads light

up *different objects or object parts*: one the zebra against a black background, one the roofline of a house, one the carrots on a plate.

What makes this striking is how indirect it is. The DINO loss touches *only* the **[CLS]** token (Rep_{CLS}) — an image-level objective that never saw a box, a mask or a label. Yet the patch tokens organise themselves into something close to an unsupervised semantic segmentation. The teaser figure of slide 24 (bird, boat, giraffes, planes) shows the same thing in photographs.

Figure 11.9. The single most reproduced figure in modern self-supervised learning. Each pair is a photograph and, beside it, the self-attention of the `[CLS]` token in the last layer of a DINO-trained ViT. The object is picked out cleanly from the background — in effect an unsupervised segmentation. Nothing in DINO's training ever saw a mask, a box or a label, and the loss was applied *only* to the `[CLS]` token, never to the patch tokens that produce this map.

How to read it: Compare the boat and the giraffe. The attention follows the object outline, including thin structures like the mast and the legs — detail an image-level loss had no direct way to ask for. That is what "emergent" means here: nobody specified it, and it appeared anyway.

Source: Lecture 07, slide 24 (Caron et al., 2021).

Why should this happen? The intuition the analysis supports is architectural pressure: the only information the [CLS] token can exploit to match the teacher across crops is what attention aggregates from patch tokens, and crops of the same object are easier to align if attention isolates the object stably — fore/background separation is a low-variance solution to view-invariance. What must be said honestly is that this is a *hypothesis*, not a theorem (the honesty rule of Part III applies): nothing in the objective provably forces object-centric attention, and the effect is weak in ResNet backbones and depends on patch size and data composition. Treat it as empirically robust within the DINO family, and as open research beyond it. Chapter 15 returns to competing explanations.

11.4.4 Ablations that matter: multi-crop

Two ablations later chapters reuse.

1. **The EMA teacher beats every other target.** Using the current student itself as the teacher (no momentum) fails completely in DINO's ablation; a teacher frozen from the previous epoch trains, but clearly worse than the EMA — Chapter 10's BYOL finding (Grill et al. 2020) re-run inside DINO's loss.

2. **Multi-crop is the cheapest accuracy in the chapter.** Adding a few small local crops that only the student sees raises accuracy for little compute: a 96×96 crop costs a small fraction of a 224×224 one. The lecture's summary (slide 55) generalises it: "multi-crop emerges as useful general augmentation" — it is not tied to DINO.

Multi-crop also nudges *what* is learned. The student must guess the teacher's view of the whole image from a small piece, which pushes features towards recognising *objects* rather than matching *textures*.

11.5 DINOv2: better data, two more losses

DINOv2 (Oquab et al., 2024) is not a new paradigm; it is DINO engineered into a foundation model. The architecture, centering, sharpening, and EMA teacher survive untouched. The changes, per Lecture 07 slides 38–41, are (i) a much larger, deliberately *curated* training set (Section 11.6), and (ii) two additional losses stacked on the same [CLS]/patch outputs, shown accumulating across the three versions of the DINOv2 pipeline figure.

11.5.1 The iBOT loss: masked-token DINO

Chapter 5's masked autoencoders (He et al., 2022) reconstruct *pixels* of masked patches. iBOT (Zhou et al., 2022) asks a sharper question: why reconstruct pixels when DINO already maintains a self-distilled *distributional tokenizer*? Slide 40's answer is to copy the DINO mechanism one level down. Mask a fraction of the student's patch tokens; let the teacher (which sees the unmasked global crops) emit, per patch position i, a K-way distribution $p_{t,i}$ through the same centering and sharpening machinery; train the student's masked-token distributions $p_{s,i}$ by cross-entropy:

$$\mathcal{L}_{\text{iBOT}} \;=\; -\sum_{i \in \mathcal{M}} \sum_{k=1}^{K} p_{t,i}^{(k)} \log p_{s,i}^{(k)}, \tag{11.5}$$

where $\mathcal{M}$ is the set of masked patch positions and $p_{t,i}$, $p_{s,i}$ are distributions over the same K prototypes.

In plain words: this is masked image modelling where the "vocabulary" of targets is *learned on the fly* — the teacher's own prototypes — instead of a fixed set of visual tokens. It reuses the EMA teacher and stop-gradient unchanged.

The iBOT head sits beside the DINO head on the same backbone, and the losses simply add:

- $\mathcal{L}_{\text{DINO}}$ on [CLS] (slide 39) shapes *image-level* meaning;
- $\mathcal{L}_{\text{iBOT}}$ on masked patches (slide 40) shapes *per-patch* (dense) meaning.

Slide 43 exploits exactly this: frozen DINOv2 features plus a linear probe give sensible depth-like and segmentation-like maps even on unusual inputs such as watercolours and sketches.

11.5.2 The KoLeo regularizer: spread out the batch

Cross-entropy alignment, even with centering, has no term that cares about how the batch's features are arranged *relative to each other* — the failure will become concrete in Chapter 16 as "good alignment, poor uniformity". Slide 41 adds the Kozachenko–Leonenko (KoLeo) regularizer (Sablayrolles et al., 2019),

$$\mathcal{L}_{\text{KoLeo}} \;=\; -\frac{1}{n}\sum_{i=1}^{n}\log d_{n,i}, \qquad d_{n,i} = \min_{j\neq i}\|\boldsymbol{z}_i - \boldsymbol{z}_j\|_2, \tag{11.6}$$

where $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$ are the (length-normalised) features of one batch and $d_{n,i}$ is the distance from sample i to its *nearest other sample in the batch*.

In plain words — what KoLeo does

> For every image in the batch, find its closest neighbour, and reward the network for making that distance larger. Nobody is allowed to crowd anybody else. The result is features spread evenly over the space instead of bunched up — the same "uniformity" goal as Chapters 9–10, but achieved using only the current batch, with no queue and no negatives.

The odd name is a citation: Sablayrolles et al. (2019) derived it from the Kozachenko–Leonenko entropy estimator, so KoLeo is a cheap way of making the feature distribution high-entropy — spread out. It costs almost nothing: the nearest neighbours are found inside one batch, from a single pairwise-distance matrix.

Worked Example — KoLeo on a batch of four

Let four normalized features on the unit circle sit at angles 0°, 10°, 180°, 270° (a clumpy batch: two points are nearly duplicates). Chord distances: between the 0° and 10° points ≈ 0.174; between the 0° and 180° points 2.0. Now compute each nearest neighbour and the loss:

i	$d_{n,i}$	$-\log d_{n,i}$
0°	0.174	1.749
10°	0.174	1.749
180°	$\sqrt{2} \approx 1.414$	-0.347
270°	$\sqrt{2} \approx 1.414$	-0.347

$$\mathcal{L}_{\text{KoLeo}} = \tfrac{1}{4}(1.749 + 1.749 - 0.347 - 0.347) \approx 0.70.$$

Now move the 10° point to 90° so the four points spread evenly: every nearest-neighbour distance becomes $\sqrt{2}$, and $\mathcal{L}_{\text{KoLeo}} = -\log\sqrt{2} \approx -0.35$. Removing one duplicate dropped

the loss by more than a full nat, and the gradient concentrates on the offending pair: only *the closest pair in the batch* dominates each sample's term. KoLeo is a local, cheap crowding penalty — and a reminder why large, de-duplicated batches interact with it (exact duplicates have $d_{n,i} = 0$, and $\log 0 = -\infty$). Figure 11.10 draws both configurations and every nearest-neighbour segment.

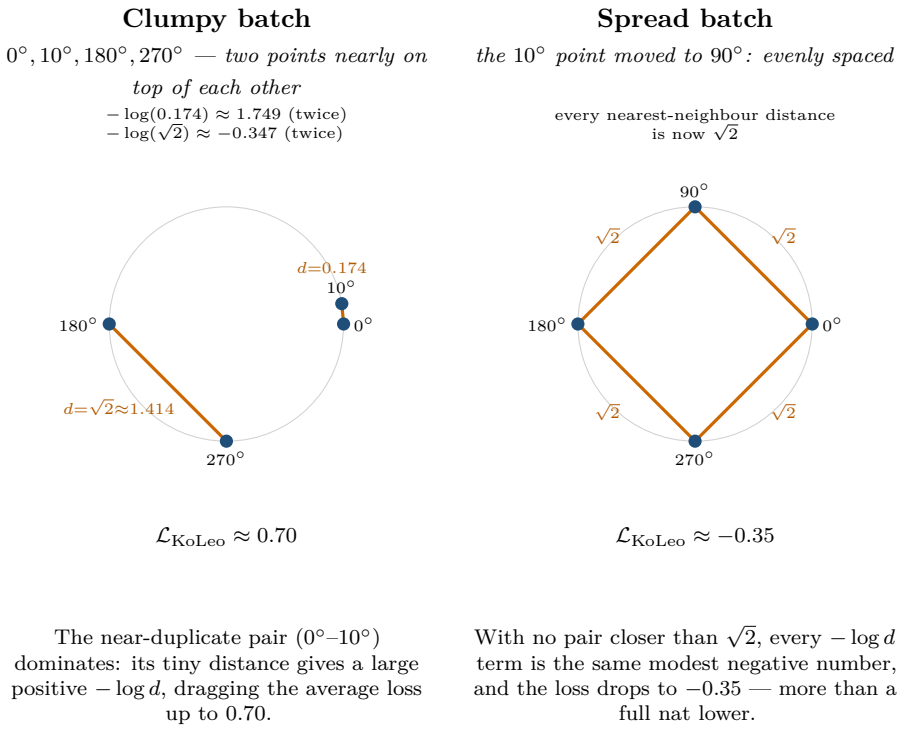
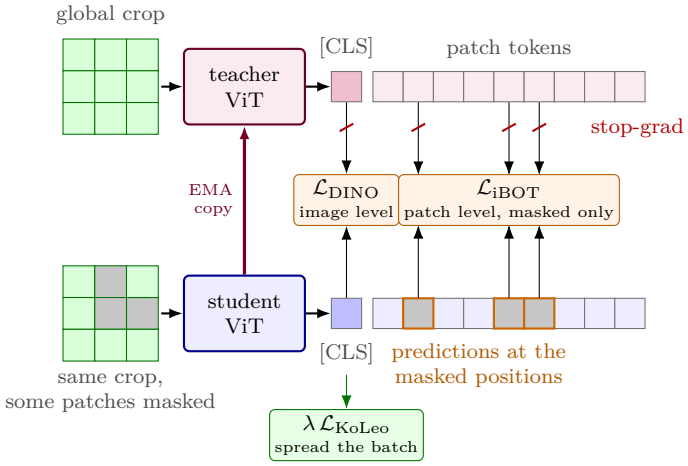


Figure 11.10. KoLeo on a batch of four unit-circle features, before and after breaking up the near-duplicate. Orange segments join each point to its nearest neighbour; that segment's length is exactly the $d_{n,i}$ each point's loss term penalizes. *Left:* the 10° point sits almost on top of 0°, so both contribute a large $-\log d \approx 1.749$; the two far-apart points $(180°, 270°)$ contribute a small negative number instead. *Right:* moving the duplicate to 90° makes every nearest-neighbour distance equal, and the loss falls by more than a full nat.

How to read it: the two orange segments in the left panel: one is barely visible (the near-duplicate pair), one spans most of the circle. KoLeo's entire gradient, for this batch, comes from those two segments — everything else about the four points' positions is invisible to this loss.

11.5.3 Results: matching weak supervision without text

The payoff (slide 42). ImageNet-1k linear-probe accuracy from the DINOv2 paper:



Each token has its own head. The [CLS] token goes through the DINO head, every patch token through a separate iBOT head; each head ends in a softmax over prototypes. On the teacher side the outputs are centred (by Sinkhorn–Knopp normalisation in DINOv2) and sharpened, and no gradient flows back.

Three losses, three jobs.
$\mathcal{L}_{\text{DINO}}$: the whole image means the same from every view.
$\mathcal{L}_{\text{iBOT}}$: a hidden patch can be predicted from its surroundings.
$\lambda\,\mathcal{L}_{\text{KoLeo}}$: different images keep their [CLS] features apart.

$$\text{total} = \mathcal{L}_{\text{DINO}} + \mathcal{L}_{\text{iBOT}} + \lambda\,\mathcal{L}_{\text{KoLeo}}$$

Figure 11.11. The DINOv2 training step (after the progressive figure of Lecture 07 slides 39–41; method of Oquab et al., 2024), drawn for one global crop cut into 3×3 patches. The teacher sees the crop whole; the student sees it with some patches masked (grey). $\mathcal{L}_{\text{DINO}}$ compares the two [CLS] outputs, exactly as in DINO. $\mathcal{L}_{\text{iBOT}}$ compares patch outputs *only at the masked positions*: the student must say, for a patch it cannot see, what the teacher says about that patch. $\lambda\,\mathcal{L}_{\text{KoLeo}}$ acts on the student's [CLS] features across the batch. Local crops (not drawn) still enter $\mathcal{L}_{\text{DINO}}$ as in Figure 11.5.

How to read it: Find the three grey patches, then follow them right: they are the only positions with iBOT arrows. The unmasked patch tokens are computed but not compared — predicting a patch you can see would be copying.

Supervision	Model	Linear
image–text (weak)	CLIP ViT-L/14 (Radford et al., 2021)	84.3
	OpenCLIP ViT-G/14	86.2
	EVA-CLIP ViT-g/14	86.4
none (DINOv2)	ViT-S/14	81.1
	ViT-B/14	84.5
	ViT-L/14	86.3
	ViT-g/14	86.5

The largest DINOv2 model matches the best image–text models *with no text and no labels at all*. For this book's argument, what matters is *how* the gap left by Chapter 9's contrastive methods was closed: not by a new kind of objective, but by DINO's own losses plus data (next section).

What the table does *not* show: it is ImageNet-style recognition only. Dense prediction is where DINOv2's patch losses matter most; slide 43's figure is the qualitative evidence — frozen ViT-g/14 features give sensible depth-like and segmentation-like linear-probe maps even on watercolours, sketches and old photographs.

Failure Modes

DINOv2-specific.

- **KoLeo on duplicated or near-duplicated batches:** $d_{n,i} \to 0$ sends $\log d_{n,i} \to -\infty$; the regularizer explodes on batches containing near-copies. This is one concrete reason the data pipeline (Section 11.6) deduplicates *before* training, and why KoLeo weighting λ must stay small.
- **iBOT masking ratio mis-tuned:** too high and the patch loss degenerates toward reconstruction-without-context (the Chapter 5 masking-ratio trade-off re-appears, now with a distributional target); too low and the patch level contributes nothing over $\mathcal{L}_{\text{DINO}}$ alone.
- **Head interference:** the DINO and iBOT heads share one backbone; without the recipes of the paper (separate learnable head parameters, patch vs. [CLS] feature routing), the two gradients can fight. Ablate one loss at a time — the failure is silent, and the proxy metric (k-NN) moves only a little while dense probing collapses.
- **Reading the headline too broadly:** "matches weakly-supervised" is a linear-probe statement on classified-image benchmarks, not a claim of parity on retrieval, counting, or dense tasks.

11.6 Scaling with data engines: DINOv3 and beyond

11.6.1 The size progression, and why it forces a new chapter of the story

The data grew, not the loss. Slide 45 puts the lecture's first fact in three lines:

DINO	ImageNet-1k, 1.3 M images
DINOv2 (Oquab et al., 2024)	142 M images
DINOv3 (Siméoni et al., 2025)	1689 M images

From DINO to DINOv3 the objective gained two extra losses, but the training *set* grew about a thousandfold. The lecture's summary bullet says it directly: *scaling requires data engines.* This section explains what that means in practice.

The payoff, from the same slide's dense-probing table (DINOv3 paper Table 3): with frozen backbones and linear heads, DINOv3 7B/16 reports 55.9 mIoU on ADE20k, 81.1 on Cityscapes, 86.6 on PASCAL VOC, with NYUv2/KITTI depth errors of 0.309/2.346 — the best overall row, ahead of agglomerative and weakly-supervised comparators of similar scale. Dense linear probing is exactly where Section 11.5's patch losses should pay; scale, here, is what makes them pay *at all.*

11.6.2 Curated data versus crawl data

In plain words — why a self-supervised model cares so much about its data

A supervised model is told what matters: the labels name the classes, and every class gets its own column in the loss. A self-supervised model is told nothing. It learns whatever makes its pretext task easier on the images it is shown, and it spends its capacity in proportion to how often things appear. Feed it the raw web, and it becomes an expert on what the web is full of — memes, product shots, the same viral photo in ten thousand copies — and learns little about the rare things you may later care about. With no labels to steer it, choosing the data *is* choosing what the model will know.

The two kinds of data (slide 47).

- **Curated** datasets (ImageNet, Google Landmarks, Mapillary) come with licence information, annotations and no duplicates — but they are small, and biased towards whatever someone once chose to photograph and label.
- **Crawled** data is practically unlimited, but "highly unbalanced", with "no content moderation" and "watermarks and other artifacts".

A *data engine* aims for the scale of the crawl with the good properties of curation, by doing the curation *automatically, in embedding space*: images are compared by the distance between their feature vectors, never by a human. Each weakness of raw crawl data has its own repair step, and the rest of this section walks through them in pipeline order.

Problem in the crawl	What it does to SSL	Repair step
Exact and near-duplicate copies	the model sees one image thousands of times: wasted compute, and the pretext task can be solved by memorising	hashing, then copy detection (stages 1–2)
Copies of benchmark test images	evaluation measures memory, not generalisation (test-set leakage)	benchmark deduplication (stage 2)
Unsafe content, identifiable faces	legal and ethical risk, and content nobody wants the model to model	NSFW filter, face blurring (stage 1)
Off-topic images (screenshots, text, clutter)	capacity spent on images unlike any downstream task	retrieval around curated seeds (stage 4)
Long tail: a few concepts flood the pool	features specialise in the frequent concepts; rare ones get little capacity	balancing with hierarchical k-means (Section 11.6.5)

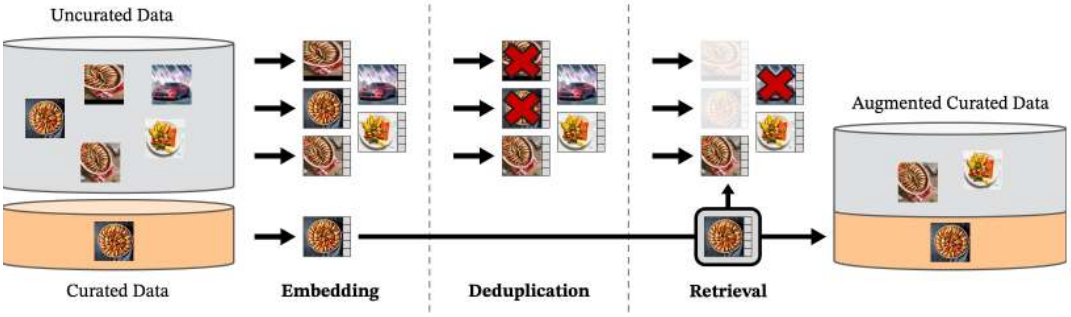


Figure 11.12. The DINOv2 data pipeline as drawn on Lecture 07, slide 49 (figure of Oquab et al., 2024). Web images (grey bucket) and curated images (orange bucket) are embedded; duplicates are crossed out; each curated image then *retrieves* similar web images, which join it in the "augmented curated" bucket.

How to read it: Follow the pizza. The curated pizza photo at the bottom travels right to the retrieval step, where it picks out the similar web pizzas above it; the red-crossed images are duplicates removed earlier, and the unrelated web images simply never get retrieved.

Source: Lecture 07, slide 49 (Oquab et al., 2024).

11.6.3 The DINOv2 pipeline, stage by stage

Slide 48 lists four stages and slide 49 draws them as buckets (Figure 11.12). The inputs are two pools. The *uncurated* pool is images linked from a public repository of crawled web pages. The *curated* pool is a list of trusted datasets — ImageNet-22k, ImageNet-1k, Google Landmarks, and a set of small fine-grained, segmentation, depth and retrieval datasets (Oquab et al., 2024). The curated images are the *seeds*: they define what "good" data looks like. Figure 11.13 follows the image counts through the pipeline.

Stage 1 — download and post-process. *What:* collect image links from the `<img>` tags of crawled web pages, drop unsafe or restricted domains, download. *How:* three cheap filters run on every image as it arrives: a compact *hash* computed from the image content (a PCA hash), so that identical images get identical codes and can be dropped by a table lookup; an NSFW classifier; and a face detector whose detections are blurred. *Why first:* these steps look at one image at a time, so they are cheap enough for billions of images, and everything later is cheaper on a smaller, safer pool. *Result:* about 1.2 billion unique images (the paper's main text; its appendix starts the next stage from 1.3 B).

Stage 2 — near-duplicate removal (copy detection). *What:* a hash only catches exact copies. A resized, re-compressed or lightly cropped copy has different pixels and a different hash, but it is still the same photo. *How:* embed every image with a network trained specifically for copy detection (SSCD, Pizzi et al., 2022), whose features are close for two versions of one photo and far for different photos. For each image, find its 64 nearest

neighbours; connect any pair with cosine similarity above 0.6; every connected group of images is one photo in many copies, and only one representative is kept. *Result:* 1.3 B → 1.1 B. Then the same procedure is run *against the evaluation benchmarks*, with a stricter threshold of 0.45: any web image that looks like a test (or train) image of a benchmark the paper reports on is discarded. *Result:* 1.1 B → 744 M. *Why:* without the second pass, a model could score well on a benchmark simply because it had seen near-copies of the test images during pretraining. This is the decontamination that keeps Chapter 1's evaluation honest at web scale.

Stage 3 — embed and cluster. *What:* give every remaining image a general-purpose feature vector and group the pool into visual concepts. *How:* a self-supervised ViT-H/16 pretrained on ImageNet-22k embeds each image; similarity is cosine similarity; then k-means splits the pool into 100,000 clusters. No image is removed here — this stage builds an *index* for the next one. *Why k-means? (the callout on slide 50).* Stage 4 needs, for some seed datasets, "more images of this kind of thing", and a cluster is exactly a ready-made list of images of one kind of thing. Figure 11.14 and the worked example below show why nearest neighbours alone are not enough.

Stage 4 — retrieval around the curated seeds. *What:* keep only web images that resemble some curated seed; everything else — the screenshots, the memes, the clutter — is never selected. *How:* two rules, chosen by the size of the seed dataset.

- **Large seed datasets (over 1 M images): nearest neighbours.** Each seed image retrieves its k nearest web images: $k = 4$ for ImageNet-22k and Google Landmarks, $k = 32$ for ImageNet-1k (to make it a core part of the data). The slide's "4 similar web-crawl images" is this rule. The paper tried larger k; the retrieved images still looked good, but more of them were retrieved by several seeds at once, so $k = 4$ was the better trade-off.
- **Small seed datasets: whole clusters.** Every k-means cluster that contains more than 3 of the seed images contributes 10,000 random images; each seed dataset is capped at 1 M retrieved images so that no single dataset dominates.

Some curated datasets (ImageNet-22k, Google Landmarks, Mapillary street sequences) are also included *as is*. *Result:* LVD-142M, 142,109,386 images. *Engineering:* all nearest-neighbour searches use the Faiss library (Johnson et al., 2021) with compressed GPU indices; the whole pipeline ran in under two days on 20 machines with 8 GPUs each.

Worked Example — where the 142 million images come from

Table 15 of Oquab et al. (2024) lists every source. Three rows do most of the work:

- ImageNet-22k: 14,197,086 images, $k = 4$ neighbours each: $4 \times 14{,}197{,}086 = 56{,}788{,}344$ retrieved.
- ImageNet-1k: 1,281,167 images, $k = 32$ each: $32 \times 1{,}281{,}167 = 40{,}997{,}344$ retrieved.

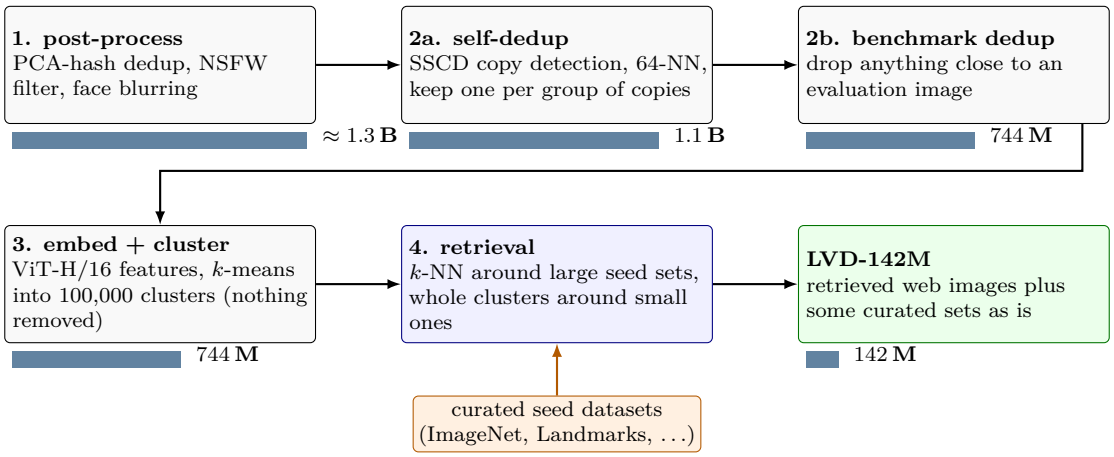


Figure 11.13. The DINOv2 curation funnel with the image counts reported by Oquab et al. (2024). Bar lengths are proportional to the number of images after each stage. Deduplication removes about 40% of the crawl; stage 3 removes nothing but builds the cluster index; retrieval keeps only the web images that resemble a curated seed.

How to read it: The big cut is the last one: from 744 *M candidates to* 142 *M. Everything before it removes* bad copies*; retrieval removes* irrelevant *images, which is the step that gives the dataset its character.*

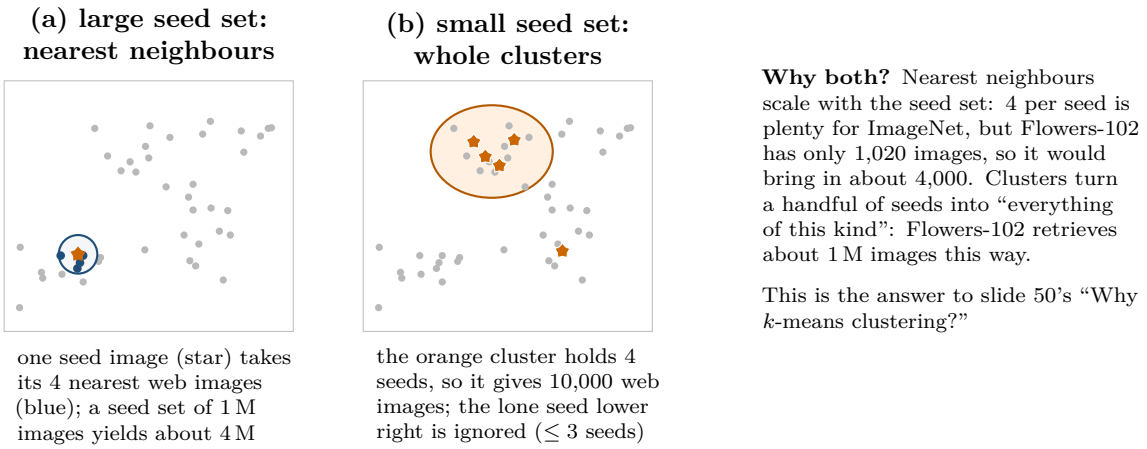


Figure 11.14. The two retrieval rules of DINOv2's stage 4, drawn on the same toy embedding space (grey dots are web images, k-means clusters are the four clumps). (a) Around a large seed dataset, each seed image takes its nearest neighbours. (b) Around a small one, the seeds only *point at* clusters, and the clusters supply the images; a cluster with too few seeds is skipped, so a single stray seed does not drag in an unrelated concept.

- Google Landmarks v2: 1,580,470 images, $k = 4$ each: 6,321,880 retrieved.

Together that is 104.1 M images from nearest-neighbour retrieval. The curated sets added as they are (ImageNet-22k, Google Landmarks, Mapillary) contribute 17.2 M. The remaining 20.8 M come from the cluster rule: 21 small-dataset retrievals, 19 of them

hitting the 1 M cap. Flowers-102 is the clearest case: 1,020 seed images, about 1.06 M retrieved through their clusters, capped at 1 M — roughly a thousand web images per seed, where nearest neighbours would have given four. Sum: $104.1 + 17.2 + 20.8 = 142.1$ M. ✓

The lesson in the numbers: about three quarters of LVD-142M is "the web's version of ImageNet and Landmarks". The small datasets cannot dominate because of the cap, but each one still gets a million images of its kind — far more than it could ever contribute on its own.

11.6.4 The DINOv3 pool: the same engine, a hundred times bigger

DINOv3 (Siméoni et al., 2025) keeps the engine and changes the source. Slide 46 makes the social point by circling the logo under the phone screenshots: the raw pool is about 17 billion public Instagram images, which only a platform owner can collect. These posts have already passed the platform's content moderation, which takes over stage 1's safety filtering. Slide 51 then splits the pool into three parts, each with a different job:

Part	How it is built	Its job
A	hierarchical k-means on DINOv2 embeddings, then balanced sampling (next subsection): 17 B → about 1.7 B	*diversity*: every visual concept represented, the long tail flattened
B	k-NN retrieval from curated seed datasets into the pool, as in DINOv2's stage 4	*relevance*: plenty of images like the ones downstream tasks use
C	public datasets as they are: ImageNet-1k, ImageNet-22k, Mapillary street sequences (about 16 M)	*quality*: known, clean data

At training time the parts are mixed: 10% of batches are *homogeneous* batches of ImageNet-1k only, and 90% are heterogeneous mixes of all parts.

Why three parts rather than one? The DINOv3 paper's data ablation (a shorter training run) answers it. Compared with training on the raw pool, the clustered data (Part A style) gains most on tasks that need breadth — fine-grained species recognition on iNaturalist, 70.1 → 81.3, and landmark retrieval on Paris, 63.3 → 85.2 — but is slightly worse on ImageNet k-NN (80.1 → 79.4). Retrieved data (Part B style) is best on ImageNet-like tasks (k-NN 84.0). The mixture beats both on every column (Siméoni et al., 2025). Balance buys coverage, retrieval buys relevance, and a data engine needs both.

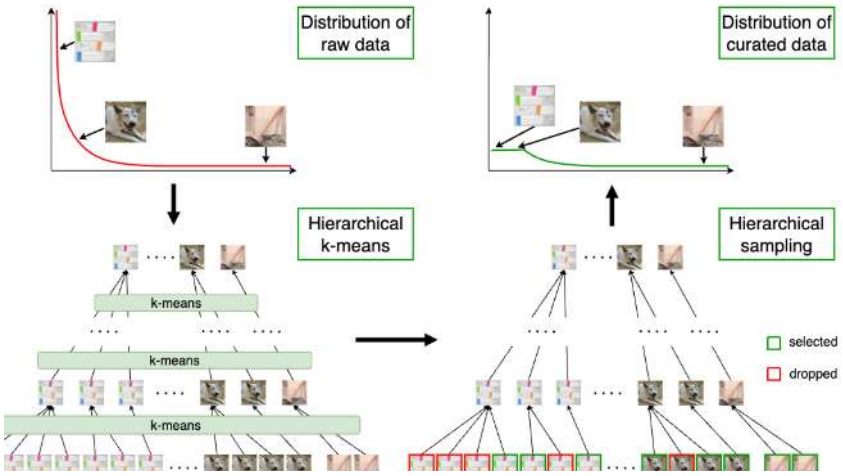


Figure 11.15. Hierarchical k-means for data balancing, Lecture 07, slide 52 (figure of Vo et al., 2024). Left: the long-tailed raw distribution is clustered by k-means, then the centroids are clustered again, level by level. Right: images are sampled down the tree so that every branch gets a fair share (green: selected, red: dropped), which flattens the distribution.

Source: Lecture 07, slide 52 (Vo et al., 2024).

11.6.5 Balancing the pool: hierarchical k-means (Vo et al.)

The problem. Retrieval and deduplication decide *what* is in the pool; they do not decide *how much* of each thing. Crawled data is long-tailed (slide 52, left curve): a few concepts (pets, food, faces) make up a large share of the pool while most concepts are rare. The goal of Vo et al. (2024) is to turn that curve into the flat one on the right of the slide, without any labels to say what the concepts are.

Why plain k-means is not enough. The natural idea is: run k-means, then take the same number of images from every cluster. It helps, but not enough, because k-means spends its centroids where the data is dense. Its objective is the total squared distance to the nearest centroid, and a concept with millions of images lowers that total most when it gets many centroids. So dogs end up split into many clusters (dogs on grass, dogs on sofas, puppies, ...) while a rare concept gets one. Sampling equally per cluster then still gives dogs a share proportional to their cluster count. Figure 11.16 puts numbers on this.

Fix 1: build a hierarchy. Cluster the images into many small clusters (level 1), then cluster the level-1 *centroids* into fewer, coarser groups (level 2), and so on up to a few very coarse groups at the top. The six dog clusters of Figure 11.16 are separate at level 1 but merge into one node at level 2, so a budget split evenly at the top treats "dog" as one concept among many. DINOv3 uses five levels, with 200 M, 8 M, 800 k, 100 k and 25 k clusters from the finest to the coarsest (Siméoni et al., 2025).

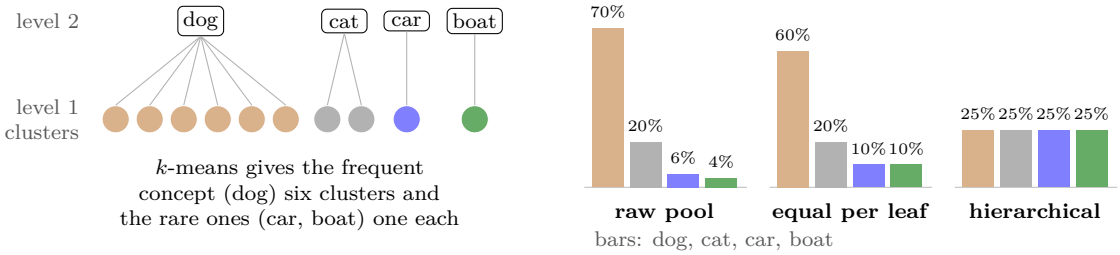


Figure 11.16. A toy pool whose images are 70% dogs, 20% cats, 6% cars and 4% boats. Level-1 k-means gives the concepts six, two, one and one clusters. Sampling 40 images *equally per leaf cluster* (4 each) still gives dogs 24 of the 40 (60%). Sampling *down the tree* — 10 per level-2 node, then split among its leaves — gives each concept 25%. (Numbers invented for illustration; in the real method the higher levels are also found by k-means, not named by hand.)

How to read it: Compare the middle and right bar groups. One level of clustering already helps (dogs fall from 70% to 60%), but only the hierarchy reaches balance, because only at level 2 are all six dog clusters seen as one concept.

Fix 2: resample before re-clustering. Even at one level, the centroids crowd into dense regions. Vo et al. therefore re-run each level's k-means on a *balanced summary* of the data: take the same small number r_t of points from every current cluster, run k-means on that resampled set, and reassign all points to the new centroids. Because big and small clusters now contribute equally many points, the new centroids spread out towards the sparse regions. Repeating this m times moves the centroids steadily towards an even coverage of the data. (This is the inner loop of the paper's Algorithm 1 on slide 53.)

Building the tree (Algorithm 1, slide 53). Start from features $X \in \mathbb{R}^{n \times d}$.

1. Level 1: run k-means with k_1 clusters on the data.
2. Resample r_1 points from each cluster, re-run k-means on those points, reassign all data; repeat m times.
3. Level 2: repeat steps 1–2 on the level-1 *centroids*, with $k_2 < k_1$ clusters.
4. Continue for T levels.

Sampling from the tree (slide 54) hands out the budget from the top down, as evenly as possible. Let s_j be the number of data points in subtree j (counted bottom-up) and N the number of samples to draw from a node. Choose the per-subtree quota n satisfying

$$N \;=\; \sum_{j} \min(n, s_j), \tag{11.7}$$

(the slide writes it as choosing n to minimise $\left|N - \sum_j \min(n, s_j)\right|$, since an exact solution may not exist in whole numbers), then recurse one level down with each subtree's allocation $\min(n, s_j)$ until the leaves, where the images inside a leaf cluster can be drawn at random ("r"), closest to the centroid ("c"), or furthest ("f"); the paper compares these options. The rule in one sentence: *give every subtree an equal share n, capped at what it can actually supply*

Table 11.2. Worked instance of the $\min(n, s_j)$ balancing rule (11.7) for one node with four subtrees (sizes s_j in the left column) and budget $N = 420$: too small an n leaves budget unspent; the solution $n^* = 120$ spends exactly N.

Subtree	s_j	$n = 80$	$n = 100$	alloc. at $n^* = 120$	share of images
A (pets)	500	80	100	120	24%
B (food)	300	80	100	120	40%
C (landscapes)	120	80	100	120	100%
D (tools)	60	60	60	60	100%
$\sum_j \min(n, s_j)$	—	**300**	**360**	**420** $= N$	—

s_j, and hand the unspent budget to the subtrees that still have data. Small concepts keep everything they have; large concepts split the remainder evenly.

Worked Example — Solving $\min(n, s_j)$ by hand

Use Table 11.2: four subtrees of sizes $(500, 300, 120, 60)$; budget $N = 420$. At $n = 80$ the allocation sums to $80 + 80 + 80 + 60 = 300 < N$ — too small, we are leaving budget on the table. At $n = 100$: $100 + 100 + 100 + 60 = 360$, still short. For n between 60 and 120, D is capped and A, B, C are not, so each unit of n adds *three* units of total: solve $3n + 60 = 420 \Rightarrow n = 120$. Check: $\min(120, 500) + \min(120, 300) + \min(120, 120) + \min(120, 60) = 120 + 120 + 120 + 60 = 420$. ✓ Allocation: A:120, B:120, C:120 (just exhausted), D:60 (exhausted). (Had the answer come out above 120, C would have become capped too and only A and B would keep growing — always re-check which subtrees are capped.) In a single pass the raw shares 51%/31%/12%/6% become 29%/29%/29%/14%: the head is compressed, the tail is protected, and no constraint is violated. The same computation repeats at every internal node on the way down the tree — which is why the algorithm needs the subtree sizes precomputed bottom-up.

This is also the conceptual hinge back to the course's first weeks: from the slide 3 exam question on DBSCAN to a production system whose first act is "run k-means at 17-billion scale," the classical toolbox of Chapters 2–3 never stopped being the machinery — learned embeddings only changed what the distances mean. Chapter 14's system-level case studies re-use the same pipeline pattern (embed, filter, retrieve) inside robots.

11.7 The BERT interlude: masked-token modelling for language

The lecture interrupts its vision sequence after SimSiam and before DINO (Lecture 07, slides 18–23) with BERT (Devlin et al., 2019). The placement is not filler: BERT is where *masked token modelling* — the objective that Chapter 5's MAE and this chapter's iBOT loss both port back into vision — had its defining success, and where the feature-based-versus-

finetuning question that organizes all of this chapter's evaluation sections was first argued at scale.

11.7.1 Setup and the two pre-training tasks

BERT is a bidirectional Transformer encoder (Vaswani et al., 2017) pre-trained on unlabelled text and then transferred two ways: as a frozen feature extractor (the ELMo style of Peters et al., 2018) or finetuned end-to-end with a small task head (the GPT style of Radford et al., 2018). Slide 19's spam-filter pipeline makes the finetuning mode concrete: text in, BERT block, a feed-forward classifier with softmax, an 85%/15% spam decision out.

Task 1: masked language modelling (MLM, slide 21). Hide 15% of the input tokens at random and predict each hidden token from the words around it. For "... stick to [MASK] in this skit" the model outputs a softmax over the whole vocabulary, with 10% on the right word and a long tail below.

The real hiding rule has three cases, which the slide's figure leaves out. Of the tokens chosen for prediction (15% of the sequence):

- 80% are replaced by the [MASK] symbol;
- 10% are replaced by a random token;
- 10% are left unchanged.

Why not always [MASK]? Because [MASK] never appears during fine-tuning. A model trained only on [MASK] would learn to trust that symbol as "the place to predict"; the 80/10/10 mix teaches it to check *every* token.

The second task: next-sentence prediction. BERT was also given pairs of sentences A and B and asked: *does B really follow A in the original text?* (Slide 22 pairs "the man [MASK] to the store" with "penguin [MASK] are flightless birds" — clearly not, and BERT says so with 99% confidence.) The idea was to teach relationships between sentences, useful for question answering.

It didn't survive. Later work found that dropping this task, and simply training longer on bigger batches, gave *better* results. A lesson that recurs through this book: an extra task that sounds useful is not necessarily useful. Modern masked language models use only the fill-in-the-blank task.

11.7.2 Bidirectional versus causal objectives

BERT vs GPT: which way attention looks. Slide 20 is the lecture's one figure on Transformer internals.

- **BERT**: every token attends to *all* positions, left and right. That is allowed, because the masked word is only filled in, never generated.

- **GPT** (Radford et al., 2018): position i attends only to positions $\leq i$. The job is to *generate* the next token, and peeking ahead would make that trivial.

Two consequences for this book:

1. BERT's MLM is a *denoising* objective in the Chapter 5/7 sense — rebuild the damaged token from context. MAE and iBOT inherit exactly this.
2. The two are not interchangeable. MLM sees both sides, so it cannot generate text on its own. Causal LM can generate, but each token's representation only sees half the context.

Slides 21 (masking) and 22 (NSP) together say that in 2018–2019, *understanding representations* and *generation* were separate research bets. Chapter 16 returns to how the field later merged them.

11.7.3 Results: frozen features versus finetuning

BERT's results (slide 23). *Table 1 — SWAG* (Zellers et al., 2018), choosing the most sensible of four continuations:

ESIM+GloVe	52.7 test
ESIM+ELMo	59.2
GPT, fine-tuned	78.0 test
$\text{BERT}_{\text{BASE}}$, fine-tuned	81.6 dev
human expert (slide)	85.0
$\text{BERT}_{\text{LARGE}}$, fine-tuned	**86.3** test

Table 2 — CoNLL-2003 NER (Tjong Kim Sang and De Meulder, 2003) (the paper's Table 7) is subtler. It asks: can you use BERT *frozen*, as a feature extractor, and if so, which layer?

$\text{BERT}_{\text{BASE}}$ fine-tuned	96.4 dev
frozen $\text{BERT}_{\text{BASE}}$: last four layers concatenated	96.1 dev
frozen $\text{BERT}_{\text{BASE}}$: all twelve layers, weighted sum	95.5 dev
ELMo features	95.7 dev
frozen: embeddings only	91.0 dev

The best frozen option is only about 0.3 points behind fine-tuning — much closer than the previous feature-based state of the art. Two lessons, read through Chapter 1's lens:

- *Which layer* you probe matters as much as whether you probe (embeddings alone: 91.0).
- "Frozen or fine-tuned?" has no universal answer. It depends on how much of the task is already linearly readable in the middle layers.

Section 11.4's linear-probe-versus-k-NN pairing asks the same question for vision — which is why slide 23 deliberately echoes slide 10's evaluation taxonomy.

Check Your Understanding

S1. (*Level 1*) DINO reports both linear-probe and k-NN accuracy on ImageNet. Which of the two requires fitting any parameters at evaluation time, and which question does each answer about the feature space?

S2. (*Level 1*) In Table 11.1, which backbone change produces the largest DINO improvement, and by how much on the k-NN column?

S3. (*Level 2*) Why does BERT corrupt only 80% of the selected tokens with [MASK], replacing 10% with random tokens and keeping 10% unchanged? What mismatch would always-masking create?

S4. (*Level 2*) The iBOT loss (11.5) is applied to masked patch tokens, with the teacher emitting targets from the *unmasked* view. Why is running it without the teacher (predicting tokens from a frozen target distribution) a worse idea than it sounds?

S5. (*Level 2*) Compute $\mathcal{L}_{\text{KoLeo}}$ for a batch of three features at angles $0°, 60°, 65°$ on the unit circle, and state qualitatively which pair the gradient acts on.

S6. (*Level 3*) DINO's k-NN score nearly matches its linear-probe score on ViT-S, while the competitors show a 5–8 point gap. Construct a toy 2-D class-conditional geometry (describe it) in which linear probing succeeds but k-NN fails, and one where the opposite holds.

S7. (*Level 3*) In the hierarchical sampler, give the formula governing per-subtree allocation and solve it for subtree sizes $(50, 30, 8, 2)$ with budget $N = 40$.

S8. (*Level 3*) DINOv3 mixes 10% ImageNet and 90% heterogeneous parts at training time. Give two distinct reasons why a 100% "final-pool" mixture might underperform this, connecting one to Chapter 12's semi-supervised discussion.

S9. (*Level 4*) The emergent attention maps appear although the DINO loss touches only the [CLS] token. Propose two competing mechanistic explanations and one experiment that would discriminate between them.

S10. (*Level 4*) Why does KoLeo (11.6) need batch-level de-duplication upstream, and what does its Kozachenko–Leonenko provenance imply about what it is (approximately) maximizing?

S11. (*Level 5*) The lecture claims "scaling requires data engines." Argue both sides: construct the strongest version of the claim from this chapter's evidence, then the strongest counterexample from Chapters 9–10, and say what observation would settle the disagreement.

S12. (*Level 5*) Frozen $\text{BERT}_{\text{BASE}}$ concatenating its last four layers to within 0.3 F1 of its own finetuned score on NER, while the SWAG gap is enormous. Propose a property of a downstream task that predicts which regime will hold, and connect it to this chapter's linear-probe-versus-k-NN discussion for DINO.

Solution sketches.

S1. Linear probe fits a classifier (parameters!); k-NN fits nothing. Linear asks "is a hyperplane enough?"; k-NN asks "do neighbourhoods already respect class identity?" in the raw metric.

S2. ResNet-50 $\to$ ViT-S lifts DINO k-NN from 67.5 to 74.5 (+7.0), while its linear probe rises 75.3 $\to$ 77.0; the same switch *lowers* BYOL and SwAV. Architecture alignment, not just objective quality, drives the gain.

S3. Always-masking teaches the network to rely on a symbol absent at finetuning time — a train/test input mismatch. The 10% random and 10% unchanged tokens force every representation to encode a denoised guess of the true token, not to detect [MASK].

S4. A fixed target cannot co-evolve with the student: it either freezes to pixels (recovering MAE-style reconstruction, which Chapter 5 shows is weaker for linear probing) or decouples from the feature geometry the alignment loss is shaping. The online teacher supplies a vocabulary that improves as the masker improves — the same self-distillation argument as (11.2), one level down.

S5. Chord distances: 0°–60° is 1, 60°–65° is $2\sin 2.5° \approx 0.087$, and 0°–65° is ≈ 1.075. Nearest neighbours: $0° \to 60°$ ($d = 1$), $60° \to 65°$ and $65° \to 60°$ (both 0.087). So $\mathcal{L} = -\frac{1}{3}(\log 1 + 2\log 0.087) \approx 1.63$. The gradient mostly pushes the 60°/65° pair apart (the derivative of $-\log d$ is $-1/d$, about eleven times larger there than for the 0° point).

S6. (a) Linear works, k-NN fails: two classes separated cleanly along one axis, plus a second, irrelevant axis with much larger spread — distances are dominated by the irrelevant axis, so nearest neighbours are often from the other class, yet a linear boundary on the first axis is perfect. (b) k-NN works, linear fails: two concentric circles (or an XOR pattern) — every point's neighbours share its class, but no straight line separates them. DINO's parity of the two metrics says its space resembles neither pathology.

S7. Rule: $N = \sum_j \min(n, s_j)$. Sizes $(50, 30, 8, 2)$, $N = 40$: $n = 10$ gives $10+10+8+2 = 30$; $n = 12$ gives $12 + 12 + 8 + 2 = 34$; uncapped are the two large subtrees, solve $2n + 8 + 2 = 40 \Rightarrow n = 15$; check $15 + 15 + 8 + 2 = 40$ ✓.

S8. (i) ImageNet remains the highest-quality-curated anchor: regularizing 10% of steps against known-clean distribution guards against crawl artifacts; (ii) evaluation-centric: downstream benchmarks are ImageNet-like, and a constant supervised-batch share is implicit distillation of that domain — cf. Chapter 12's argument that pseudo-label mixtures bias toward the labeler's distribution.

S9. Explanation A (architectural): attention is the only path from patches to [CLS]; view-invariance is minimized by segmenting the stable object. Explanation B (data/object-centric): natural crops at DINO's scales select for object-centric invariances in the loss landscape itself. Discriminator: train on textures/synthetic data with matched

statistics but no objects — A predicts segmentation-like maps still emerge; B predicts they vanish.

S10. Exact or near duplicates give $d_{n,i} \to 0$ and $\log d_{n,i} \to -\infty$: the loss diverges and its gradient concentrates on the copies, not on the geometry. KoLeo derives from the Kozachenko–Leonenko differential-entropy estimator, so minimizing it approximately maximizes the feature distribution's entropy — a uniformity surrogate (Chapters 9–10 uniformity view) computed from the batch alone.

S11. For the claim: the only visible difference between DINO (77.0) and DINOv3 (best dense-probing row) is data scale/curation at fixed loss family; the objective changed little. Against: SimSiam/BYOL/Barlow Twins made years of progress on the *same* ImageNet by changing objectives (Chapters 9–10) — data was constant. Deciding observation: DINO-family training re-run on ImageNet-1k with DINOv2/v3 losses — if it recovers most of the gap, engines lose; if not, objectives alone cap out below scale gains.

S12. Predictor: whether the task's decision structure is present (nearly) linearly in intermediate layers of the pretrained model — token-level sequence tagging is; multi-sentence commonsense inference is not. For DINO: linear probe and k-NN parity on ImageNet says image classification is essentially present in the frozen geometry, the image-domain analog of the NER result.

Concept Map — how this chapter connects

Builds on: Chapter 9 (contrastive losses as distribution matching; the uniformity–alignment split that KoLeo completes); Chapter 10 (collapse, stop-gradient, EMA teachers, predictor asymmetries — every one of DINO's stabilizers); Chapter 5 (masked autoencoders, the pixel-level ancestor of the iBOT loss); Chapters 2–3 (k-NN, k-means, deduplication — the tools the data engines industrialize).

This chapter: self-distillation as a distributional joint-embedding objective; centering + sharpening as collapse control; k-NN as the hyperparameter-free evaluation; iBOT and KoLeo as the two-level completion; data engines — curated vs. crawl, retrieve-and-balance — as the true unit of scaling; BERT as the masked-modelling original and the frozen-finetune archetype.

Leads to: Chapter 12 (distillation with labels, self-training and pseudo-labels — the supervised continuation of this chapter's teacher machinery, and the semi-supervised reading of slide 51's data mixtures); Chapter 14 (system-level SSL, where the embed–dedupe–retrieve pipeline reappears inside robots as map supervision and cross-modal teachers); the theory of Chapter 15 (why any of this learns); Chapter 16's method-family comparisons.

CHAPTER 12

Self-Training, Distillation, and Cycle Consistency

The student can outperform the teacher.

Lecture 08, recurring thesis

In this chapter

12.1 Self-Training: The Model Becomes Its Own Teacher ..... 552 • The student–teacher template • Pseudo-labels for semi-supervised learning • Worked example: self-training Fashion-MNIST under confidence filtering • Noisy Student: iteration makes the student surpass the teacher
12.2 Knowledge Distillation: Soft Targets 561 • Distillation as model compression: from DINOv2 to Gemini Flash • Many teachers, one student: DUNE
12.3 Cycle Consistency: Learning Without Pairs 571 • Paired versus unpaired data: a taxonomy
12.4 CycleGAN: Two Intertwined Adversarial Games 572 • The constraint that replaces data • A toy cycle, done by hand • What the cycle weight buys: a controlled collapse
12.5 Why Self-Training Actually Works 578
12.6 Chapter summary 578
12.7 Exercises 579
12.8 Concept Map 582

Chapter roadmap

Before you start. You need cross-entropy and softmax (Chapter 2: turning logits into a probability over classes), what an EMA (exponential moving average) teacher is and why stop-gradient prevents collapse (Chapter 10–11, DINO), and the GAN minimax game — a generator trying to fool a discriminator (Chapter 6). Nothing else is assumed.
The idea in one sentence. When you cannot get a human to label more data, you can often get *something else* to stand in for a label — a model's own confident guess, a model's full output distribution, or a structural constraint like "translate there and back and you must return home" — and each substitute buys supervision at a different price.
What we cover.

- **Self-training** (Section 12.1). The model grades its own unlabeled data and trains on the grade. Simple, and it works — but it can also talk itself into being confidently wrong, so half of this section is about the mitigations that keep that from happening.
- **Knowledge distillation** (Section 12.2). Instead of keeping only a teacher's best guess, keep its whole distribution over classes — the near-misses turn out to carry real information. This is how a huge trained model gets compressed into one you can actually serve, and it scales from one teacher (Hinton) to a committee of incompatible ones (DUNE).
- **Cycle consistency** (Sections 12.3–12.4). When there is no teacher and no paired data at all — just two unrelated piles of images — a round-trip constraint (translate there, translate back, land where you started) can substitute for supervision that was never collected.
- **Why it all works** (Section 12.5). One sentence that explains pseudo-labels, distillation and cycles as the same move at different scales, and previews world models (Chapter 13) as the same move turned toward *dynamics* rather than a single prediction.

Afterwards you will be able to explain why a self-training loop can get worse instead of better, and name the four guards against it; compute what a temperature does to a softmax distribution and say what "dark knowledge" means in one sentence; describe why a Noisy Student should be noisier *and* at least as large as its teacher, not just different; and explain what cycle consistency does and does not prove about a translation.

Every objective so far assumes the raw data can go straight into the loss. Multi-crop assumes each crop shows one main thing; contrastive learning assumes two augmented views of an image mean the same. Much of the world's data does not fit: images from the wrong domain, recordings in another modality, collections too mixed for one objective.

The lecture behind this chapter (Lecture 08, slide 3) asks directly: *how can we use data that does not fit our loss?* It names three routes — domain transfer, modality transfer, and scaling to more data — and this chapter covers the two mechanisms that deliver them:

1. **Self-training and distillation** (slides 4–22): an existing model makes up targets for a new one.
2. **Cycle-consistency** (slides 24–31): when two domains cannot be paired at all, learn two mappings that must undo each other.

The lecture's third part, world models, is in the next chapter.

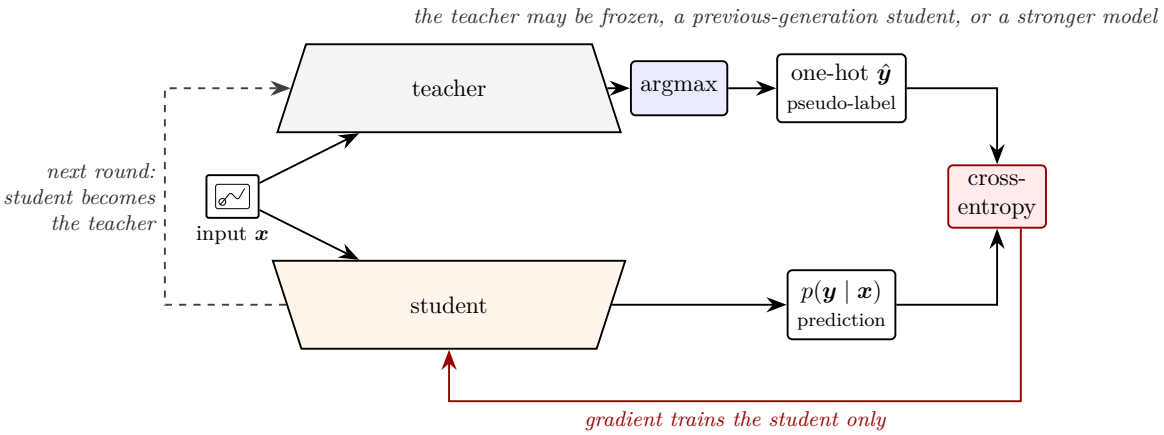


Figure 12.1. The self-training loop. A teacher turns an input $\boldsymbol{x}$ into a pseudo-label (here via arg max over its predicted distribution); the student is trained with ordinary cross-entropy against that machine-made target; and the trained or improved model can serve as the teacher of the next round (dashed retrain loop). Everything downstream of "teacher" needs no human labels, only unlabeled data. (Redrawn from Lecture 08, slides 4–5.)

12.1 Self-Training: The Model Becomes Its Own Teacher

12.1.1 The student–teacher template

Key Concept — Student–teacher training

In *student–teacher* training an already-trained (or frozen) *teacher* network produces targets, and a *student* network is trained to match them. The teacher defines what counts as a label; the student never sees ground truth from humans on the data the teacher labels.

The scheme is generic: both networks see the same input $\boldsymbol{x}$, the teacher emits a prediction, and the student is optimized so its own prediction matches the teacher's (Figure 12.1). What varies across methods is only *what* the teacher emits and *who* the teacher is. The simplest choice is an arg max: the teacher's most confident class becomes a one-hot target, just like a human label — except it was produced by a machine and may be wrong.

12.1.2 Pseudo-labels for semi-supervised learning

Key Concept — Pseudo-label

A *pseudo-label* (Lee, 2013) is a label generated by the model itself for an unlabeled sample, typically the arg max of its predicted class distribution, which is then used as if it were a human annotation in the training loss. Formally, for unlabeled input $\boldsymbol{x}$ with teacher distribution $\boldsymbol{q} = p_{\boldsymbol{\theta}_T}(\cdot \mid \boldsymbol{x})$, the pseudo-label is $\hat{\boldsymbol{y}} = \arg\max_k q_k$, and the student objective

is the cross-entropy $-\log p_k$ evaluated at $k = \hat{\boldsymbol{y}}$, where p_k is the student's predicted probability of class k.

Common Confusion

Semi-supervised vs. self-supervised. Pseudo-labeling in the style of Lee (2013) is *semi*-supervised learning: it starts from a seed of human labels (600 or 1000 MNIST digits on the lecture's slides) and expands them automatically. *Self*-supervised learning starts from *no* human labels at all — the supervisory signal is derived from the structure of the data (clustering, augmentations, cycles). The two are siblings, not synonyms: pseudo-labels are the tool that lets a partially labeled pipeline behave like a label-free one, and downstream of the teacher the two become indistinguishable.

Model at a glance: Pseudo-Label Lee, 2013

In one line	Take the model's own most-confident class guess on an unlabeled example, treat it exactly like a human label, and train on it with ordinary cross-entropy.
Problem it solves	Only a handful of labeled examples exist, but a much larger pool of unlabeled examples from the same distribution is sitting unused.
Key intuition	A model trained on even a small labeled seed is already *better than chance* on the unlabeled pool. Anything better than chance is a usable training target.
Inputs	A small labeled set and a large unlabeled set of the same kind of data (in Lee's case, both are MNIST digit images).
Architecture	No constraint — any classifier with a softmax output. The mechanism is entirely about *where the target comes from*, not the network shape.
Loss	The joint objective of Eq. (12.1): ordinary supervised cross-entropy on the labeled seed, plus cross-entropy against the model's own arg max on the unlabeled pool, the second term ramped in slowly.
Training	Warm up on labeled data only ($\alpha(t)$=0), then ramp $\alpha(t)$ up so the unlabeled loss switches on only once the decision boundary is no longer flailing.
What you keep	The whole trained network — there is no separate artifact to discard, unlike, say, DeepCluster's classification head (Chapter 3).
Strengths	Extremely simple to implement; needs no second network, no augmentation library, no extra hyperparameter beyond τ and the ramp schedule; +3–4 points of accuracy from otherwise-wasted data (below).
Main failure modes	*Confirmation bias*: a confident wrong guess becomes a target, and the next round is trained to be confidently wrong in the same place (Figure 12.3).
Leads to	Confidence thresholds and curriculum (this section); Noisy Student, which scales the same idea to ImageNet with noise and iteration; DeepCluster (Chapter 3), which is the same template with a clustering algorithm in place of the teacher.

The classical evidence that this works is small but striking (Figure 12.1 is exactly Lee's setting). With only 600 labeled MNIST digits and 59,400 unlabeled ones, plain supervised training reaches 91 % accuracy; adding pseudo-labels for the unlabeled pool raises it to 95 %. With 1000 labeled and 59,000 unlabeled digits the gain is 93 % → 96 % (Lee, 2013). Three to four points of accuracy from data that was previously useless. The mechanism is *entropy minimization*: a model that has seen enough labeled examples to be roughly right will be pushed toward confident, low-entropy predictions on the unlabeled ones, which sharpens the decision boundary between classes.

Formally, Lee's setting makes the network its own teacher (Lee, 2013). Write $\mathcal{L}_L$ for the ordinary cross-entropy on the labeled seed set and $\mathcal{L}_U$ for the cross-entropy of the network's output on unlabeled inputs against the network's own arg max pseudo-labels; training then minimizes the joint objective

$$\mathcal{L}(\boldsymbol{\theta}) \;=\; \mathcal{L}_L(\boldsymbol{\theta}) \;+\; \alpha(t)\,\mathcal{L}_U(\boldsymbol{\theta}), \tag{12.1}$$

where t indexes training steps and the coefficient $\alpha(t)$ ramps from 0 to its final value: $\alpha(t) = 0$ during an initial warm-up in which only the labeled loss is optimized, then grows (linearly, or by a step schedule) until the pseudo-labeled term is fully active. The ramp is a necessity, not a nicety. Pseudo-labels produced in the first iterations come from an almost untrained network, and accepting them immediately injects label noise at precisely the moment the decision boundary is least stable; the warm-up lets the labeled data place the boundary before the network starts grading its own homework.

Self-training is entropy minimisation in disguise. Training the prediction $\boldsymbol{q}$ towards the model's own one-hot guess pushes the prediction to be *more confident*. That is (a discrete version of) minimising its Shannon entropy, $\mathcal{H}(\boldsymbol{q}) = -\sum_k q_k \log q_k$, over the unlabelled pool.

When is that a good idea? Under the *low-density assumption* — the classes form well-separated clusters. Then "be confident everywhere" pushes the decision boundary into the empty gaps between clusters, not through them.

But the same push also *freezes* whatever boundary exists when the ramp finishes:

- if the warm-up placed it well, self-training sharpens it;
- if the few labels misled it, self-training locks in the mistake.

These two outcomes of Eq. (12.1) are exactly what the next subsection calls *confirmation bias*.

Left unchecked, however, the model is also pushed to be confident about its *mistakes*. Modern self-training therefore gates pseudo-labels by confidence: only predictions whose maximum softmax probability $\max_k q_k$ exceeds a threshold τ (e.g. 0.95) contribute to the loss. Figure 12.2 shows the idea schematically — the density of teacher confidences typically splits into a high-confidence lobe that is mostly correct and a long low-confidence tail that is mostly wrong, and τ operates as a precision/recall knob on which predictions graduate to labels.

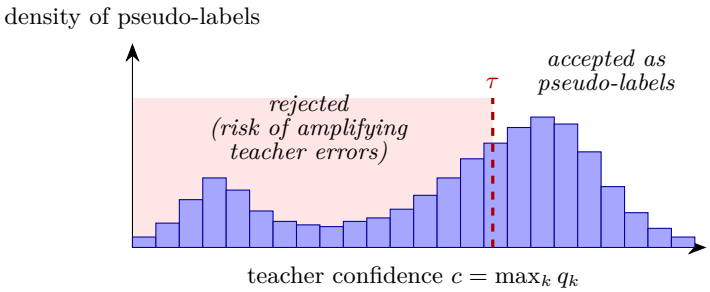


Figure 12.2. Confidence gating for pseudo-labels (schematic). The histogram of the teacher's maximum softmax probability c across unlabeled samples typically shows a high-confidence lobe of mostly-correct predictions and a long low-confidence tail. A threshold τ admits only the right lobe into the loss, trading coverage against the risk of *confirmation bias*: training on wrong labels makes the model more confidently wrong next round.

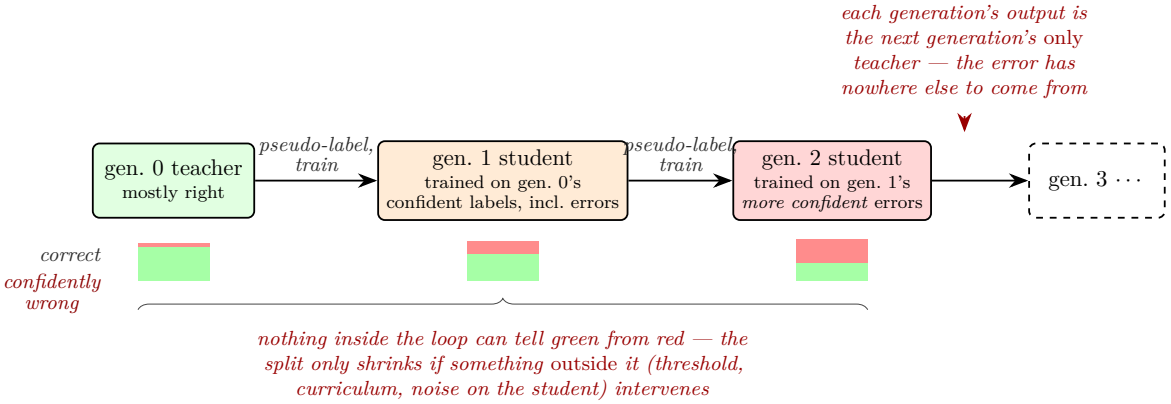


Figure 12.3. Confirmation bias as a loop, not just a statement (companion to Figure 12.1, which draws the same machinery running correctly). Each generation trains on the previous generation's *confident* pseudo-labels; the bar beneath each box is the same confidence budget split into labels that happen to be correct (green) and labels that are wrong but confident enough to pass the filter (red). Nothing in the loop can tell the two apart, so the red slice does not shrink on its own — it only shrinks if something *outside* the loop (a confidence threshold, a curriculum, noise on the student) makes it harder for wrong answers to look confident.

How to read it: compare the green/red split under each box left to right — that shrinking green slice is *confirmation bias, drawn as arithmetic instead of asserted in prose.*

Figure 12.1 draws the mechanism that makes self-training work; Figure 12.3 draws the mechanism that makes it fail. They are the same loop, run with the guard rails off.

Failure Modes

Confirmation bias and early-error lock-in. Self-training's central fragility is that the teacher grades its own homework. A wrong but confident prediction becomes a training target; the student internalizes it; the next teacher generation is now confidently wrong and produces more of the same target. Errors made *early*, when the seed model

is weakest, are the most dangerous — they get locked in and amplified over iterations. Mitigations: (i) *confidence thresholds* as in Fig. 12.2; (ii) *curriculum*: start from the highest-confidence examples and lower τ as the student improves; (iii) *class balancing*: pseudo-label distributions collapse toward frequent classes unless re-weighted; (iv) *noise on the student* (next subsection): a student forced to be consistent through augmentations and dropout cannot overfit individual wrong labels as easily. Semi-supervised systems such as FixMatch combine all four.

12.1.3 Worked example: self-training Fashion-MNIST under confidence filtering

The mitigations above are abstract until they are given knobs and numbers. Here is the pipeline, at a scale you could run on a laptop in an evening (Lecture 08's companion case-study code instantiates exactly this recipe). The setting: 1,000 labeled Fashion-MNIST images and a pool of 3,000 unlabeled ones — the same garment classes as Lee's MNIST digits, but with the confusions (shirt/coat, sneaker/ankle-boot) that make pseudo-labeling genuinely risky.

1. **Teacher from the seed.** Train a small CNN (width 16 channels) on the 1,000 labeled samples under *weak* augmentation only. The teacher should be precise on what it knows; noise belongs to the student.
2. **Inference on the pool.** Run the teacher over all 3,000 unlabeled images. Softmax, $\arg\max$, and record two things per sample: the pseudo-label and its confidence $c = \max_k q_k$.
3. **Relative confidence filter.** Rather than fixing an absolute τ, take the 0.6 *quantile* of the confidence scores and keep the top 40 % — about 1,200 pseudo-labeled samples. A quantile filter adapts to how calibrated the teacher happens to be on the data at hand; a hard-coded 0.95 admits nothing at all from an uncertain teacher and lets a confidently wrong teacher admit its own errors. Figure 12.2, read with this recipe in mind, is the trade the quantile is making.
4. **Joint training with asymmetric weights.** Train a *wider* student (width 20) on the concatenation of the labeled seed and the accepted pseudo-labeled samples, with per-sample loss weights: 1.0 for true labels, 0.5 for pseudo-labels, so that the few pseudo-labels that are wrong (they exist — even the top quantile contains confident mistakes) exert half the pull of the labeled anchor. The student sees *strong* augmentation (random crop, horizontal flip, random erasing): this is the small-scale analogue of Noisy Student's bargain — the teacher labels clean images with precision, the student must reproduce those labels under perturbations, and the regularization pressure is what lets the student generalize past its teacher.
5. **Monitor balance, not just accuracy.** Track the class histogram of the *accepted* pseudo-labels. If the accepted set collapses toward the majority classes — the early-

warning symptom of confirmation bias — loosen or tighten the quantile per class rather than globally.

Each numbered step answers one of the chapter's fragilities: the quantile filter gates early-error lock-in, the 0.5/1.0 weighting cheapens teacher errors, the wider noised student converts pseudo-labels into smoothed knowledge rather than memorized ones, and the histogram watch guards against class collapse. On an exam (and, more importantly, in the lab), asking "where in this pipeline is confirmation bias weakest?" is a diagnostic the recipe now answers step by step.

Learning note from study sessions

In a study session built from this exact case study, the stress-test question was deliberately inverted: *you have* 80 *labeled factory-defect images and* 20,000 *unlabeled ones, and the baseline classifier is confidently wrong on rare defects — what do you do?* The model's own recipe was the four-step pipeline above, wearing a harder hat: confidence-filter *per class* so rare defects are not filtered out by the majority class's calibration, weight pseudo-labels below true labels, perturb the student, and never remove the true-label anchor — removing the anchor and trusting all pseudo-labels equally is the single mistake every counter-model in the session tripped over.

What changed — From plain supervised training to self-training

What stayed. The network, the loss function (cross-entropy), and the optimizer. A pseudo-labeled batch and a human-labeled batch are numerically identical once they reach the loss — Eq. (12.1) adds a second cross-entropy term, it does not replace the first or invent a new kind of arithmetic. **What changed.** *Where the target on the unlabeled majority of the data comes from.* Plain supervised training has a zero gradient on every unlabeled example — there is nothing for the loss to compare a prediction against. Self-training manufactures that missing comparison out of the model's own confidence. **What it cost.** A new failure mode that plain supervised training structurally cannot have: confirmation bias (Figure 12.3). Plain supervised training can be wrong, but it cannot be wrong *because it believed itself* — self-training can, and everything from the confidence threshold τ to the ramp schedule $\alpha(t)$ exists only to manage that new risk.

Key takeaway — Self-training in one line

A model that is better than chance on unlabeled data can bootstrap itself — but only if something outside the loop (a threshold, a curriculum, a true-label anchor) keeps it from also bootstrapping its own mistakes. Everything in this subsection is either the bootstrap or a guard against that one risk.

12.1.4 Noisy Student: iteration makes the student surpass the teacher

Noisy Student: self-training at full scale. Pseudo-labelling starts from very few labels, so its first guesses are weak. *Noisy Student* (Xie et al., 2020) ran the self-training loop of Figure 12.1 at industrial scale, with three changes:

1. **The student is at least as large as the teacher** — not smaller, as in ordinary distillation.
2. **The student learns under heavy noise** (strong augmentation, dropout, stochastic depth), while the teacher labels *clean* images. The student must be right about harder inputs than the teacher ever saw.
3. **The student becomes the next teacher**, and the loop repeats (Figure 12.4).

With no extra human labels at all, this set a new ImageNet record at the time: 88.4% top-1, from an EfficientNet-L2 student trained using 300 million unlabelled JFT images.

Model at a glance: Noisy Student — Xie, Luong, Hovy & Le, 2020

In one line	Run self-training at industrial scale, but make every student *equal-or-larger* than its teacher and train it under heavy noise on the teacher's clean-image pseudo-labels, then repeat with the student as the new teacher.
Problem it solves	Plain self-training (Lee, 2013) plateaus: a same-size, noise-free student mostly just reproduces its teacher, errors included, with little room to improve generation over generation.
Key intuition	Matching a clean teacher's label *under noise* is a harder problem than matching it on clean inputs — solving the harder problem produces a genuinely better function, not a copy.
Inputs	A labeled seed set (ImageNet) and a much larger unlabeled pool (JFT, 300M images), filtered and balanced per class.
Architecture	EfficientNet family; the student network is deliberately sized *equal to or larger than* the teacher at every generation.
Loss	Ordinary cross-entropy against pseudo-labels (hard, in the original recipe) — the novelty is entirely in training conditions, not the loss formula.
Training	Three kinds of noise on the student only: RandAugment (input), dropout (activations), stochastic depth (layers are randomly skipped). The teacher sees clean images at inference time. After convergence, the student is frozen as the next teacher and the whole loop repeats (Figure 12.4).
What you keep	The final generation's student network — earlier generations exist only to produce that generation's pseudo-labels.
Strengths	88.4 % ImageNet top-1 at publication, beating contemporary supervised state of the art, using zero extra human labels; also measurably more robust to corrupted/adversarial inputs, a side effect of training under noise.
Main failure modes	Needs a reasonably good generation-0 teacher to start from; each iteration is expensive (a full training run); confirmation bias is still possible if noise and class balancing are skipped.

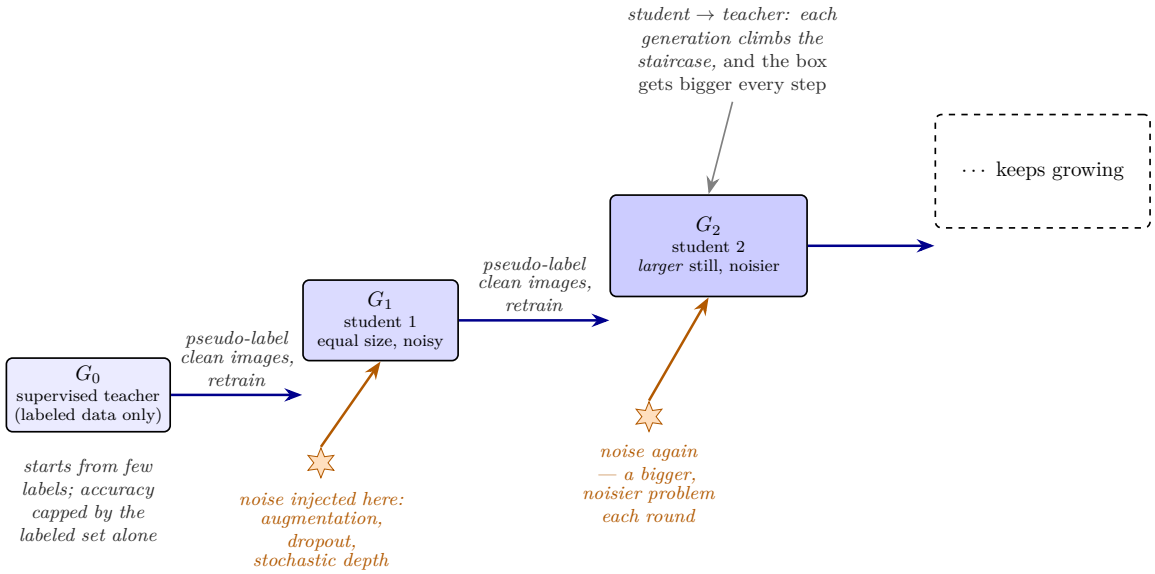


Figure 12.4. Noisy Student iterative self-training (Xie et al., 2020), with both upgrades made visible. Generation G_0 is a supervised teacher trained on labeled data only; each step distills it into an *equal-or-larger* student (box size grows left to right) trained under explicit noise injection (orange bursts: RandAugment, dropout, stochastic depth) on pseudo-labels the teacher produced from *clean* images; the student then becomes the next, bigger teacher. Iteration, not any single generation, is what lets the student surpass the teacher (Lecture 08, slides 9–10).

How to read it: follow one box left to right: it gets both physically bigger and gains an orange noise burst underneath it — "noisier and larger" is two separate upgrades happening at the same step, not one.

Leads to	DINO's EMA teacher (Chapter 11), which removes the discrete generations entirely and makes the teacher a smoothed *running average* of the student, updated every step instead of every training run.

Why the noise matters. A student trained on clean images would end up copying the teacher — mistakes included. Noise changes the deal:

- the **teacher** labels *clean* images, so its targets are as accurate as it can make them;
- the **student** must reproduce those labels on *noised* images (augmentations, dropout, stochastic depth), so it has to learn decision boundaries that hold up under those changes.

That asymmetry is where each generation's improvement comes from: the student learns a smoother, more robust version of the teacher.

Size matters too. A student the same size as the teacher still improves in the paper's ablations, but a larger student improves more: extra capacity lets it treat the noise as

regularisation rather than damage. So "noisier" and "at least as large" go together; they are not two independent knobs.

What changed — From Pseudo-Label to Noisy Student

What stayed. The core template: a teacher produces targets, a student trains on them, cross-entropy throughout. **What changed.** Three deliberate amplifications — scale (ImageNet plus 300M unlabeled images, not 60k), noise (three kinds, applied only to the student), and iteration (the student becomes the next teacher, repeatedly, rather than a single teacher→student handoff). **What it cost.** Compute: each generation is a full training run, and the recipe needs several generations to pay off. It also inherits self-training's confirmation-bias risk at a scale where a systematic error is now baked into a model other people will deploy — the class-balancing and noise machinery are not optional extras at this scale, they are load-bearing.

Key takeaway — Noisy Student in one line

Bigger and noisier is not a decoration on self-training — it is the fix for the thing that makes naive self-training stall: a same-size, noise-free student has nothing to do but memorize its teacher, errors included; a larger student solving a harder (noised) version of the same problem has room to actually improve on it.

12.2 Knowledge Distillation: Soft Targets

Pseudo-labels throw away everything except the winning class. Hinton et al. (2015) made famous the observation that this discards the most interesting signal. A teacher that says "this cat image is 85 % cat, 14 % tiger, 0.5 % dog" tells the student not just *what* the answer is but *what the answer almost was* — the teacher's confusions encode which classes are visually similar. Hinton called this information *dark knowledge*: invisible at arg max, dominant in the loss.

Model at a glance: Knowledge Distillation — Hinton, Vinyals & Dean, 2015

In one line	Train a (usually smaller) student to match a teacher's whole softened output distribution, not just its top class.
Problem it solves	Hard pseudo-labels throw away the teacher's confusions — exactly the information that says which wrong answers were *nearly* right, and hence which classes the model finds visually or semantically similar.
Key intuition	A softmax temperature T (Eq. (12.2)) is a dial: turn it up and the teacher's relative confidence in the *wrong* classes — "dark knowledge" — becomes numerically visible instead of rounding to zero.
Inputs	Any input the teacher can score; no additional labels required beyond what trained the teacher.
Architecture	No constraint on student or teacher architecture — they need not even be the same family, only comparable output spaces.
Loss	$T^2 D_{\mathrm{KL}}(q_T^{(T)} \| q_S^{(T)})$, optionally averaged with ordinary hard-label cross-entropy if true labels are also available.
Training	Compute the teacher's softened distribution once (teacher frozen); train the student to match it, with the same temperature applied to both sides.
What you keep	The student only — the teacher (or ensemble of teachers) is discarded after training, its knowledge now folded into one smaller network.
Strengths	Transfers far more signal per example than a hard label, so it needs less data to match the teacher's function; the standard mechanism behind model compression (below) and label-free variants like DINO (Chapter 11).
Main failure modes	The student inherits the teacher's systematic errors — KL rewards *faithfully* reproducing the teacher's distribution, wrong parts included; forgetting the T^2 correction silently shrinks the gradient at high T (worked example below).
Leads to	Multi-teacher distillation (DUNE, this section) and EMA self-distillation (DINO, Chapter 11), where the "teacher" is not a separately trained model at all but a smoothed copy of the student.

Worked Example — What arg max throws away: hard targets vs. soft targets

Let a teacher's distribution over three classes for one ambiguous sample be $\boldsymbol{q}_T = (0.45, 0.48, 0.07)$, and let the student's current prediction be $\boldsymbol{q}_S = (0.20, 0.70, 0.10)$.

Both fields of view are close to correct — index 1 is the modal class — but the two supervision choices differ sharply. **Hard pseudo-label.** The teacher's arg max is index 1; the cross-entropy with a hard target has the classic softmax gradient $\partial\mathcal{L}/\partial z_i = q_{S,i} - y_i$, which evaluates to

$$(0.20,\ 0.70,\ 0.10) - (0, 1, 0) = (0.20,\ -0.30,\ 0.10).$$

Class-1 probability is pulled up, class 0 and class 2 pushed down, and the student's gradient is silent on the fact that class 0 was *nearly as plausible as class* 1. **Soft target.** Matching $\boldsymbol{q}_S$ against the full teacher distribution gives the cross-entropy gradient

$$(0.20,\ 0.70,\ 0.10) - (0.45,\ 0.48,\ 0.07) = (-0.25,\ 0.22,\ 0.03).$$

Read it: every sign flipped relative to the hard case wherever the mass allocation was informative. The student is told to *lower* its class-1 score (it overshot the teacher's 0.48), *raise* its class-0 score (the teacher saw real evidence for it), and hold class 2 nearly fixed. The teacher's near-tie becomes a literal numerical instruction; at arg max it is invisible. This single comparison — same teacher, same student, different target granularity — is the entire argument for soft targets in one line of arithmetic. Figure 12.5 draws exactly what the arg max step deletes.

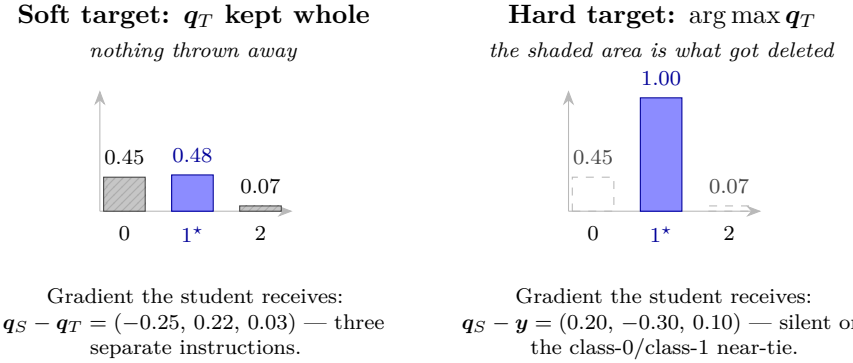


Figure 12.5. What arg max throws away, for the teacher distribution $\boldsymbol{q}_T = (0.45, 0.48, 0.07)$ of the worked example. *Left:* the full soft target — every class keeps its mass, including the near-tied class 0. *Right:* the hard pseudo-label the student would train on instead. The dashed outlines mark where classes 0 and 2 used to be; their combined mass, $0.45 + 0.07 = 0.52$, has been reassigned to class 1, which is inflated from 0.48 to a false certainty of 1.00. That reassigned 0.52 is the "dark knowledge" the soft target preserves and the hard target destroys.

How to read it: Compare the two class-0 bars: on the left it stands at 0.45, nearly tied with the winning class; on the right it has vanished to nothing, and its dashed outline is the only trace that it was ever almost as plausible as the class the model actually predicted. That gap between solid bar and dashed ghost, summed over every non-argmax class, is literally what a hard pseudo-label deletes.

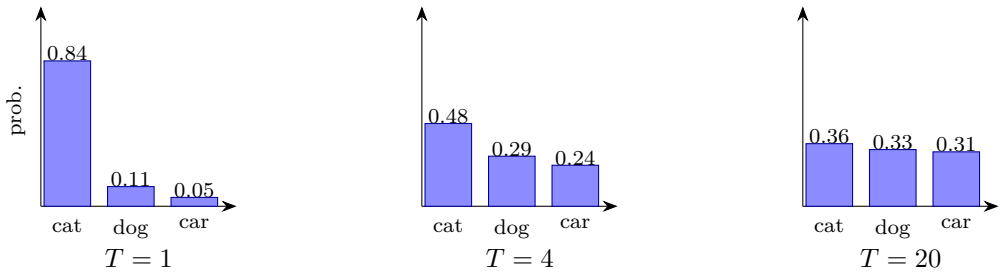


Figure 12.6. Temperature softening of the distillation target (Eq. (12.2)) for the same teacher logits $\boldsymbol{z} = (z_{\text{cat}}, z_{\text{dog}}, z_{\text{car}}) = (3, 1, 0.2)$. At T=1 softmax is decisive (0.84 on cat) but not one-hot, and the "dark knowledge" about which non-cat class is most plausible is down at the few-percent level; at T=4 the cat/dog/car ambiguity becomes explicit; at T=20 the distribution is nearly uniform and carries almost only similarity structure, no decision.

Historical Note

Lineage of distillation. The modern formulation is due to Hinton et al. (2015) (Google, 2015), but the idea that a small model can be trained to imitate a large one predates deep learning's current wave: model compression via an ensemble's outputs was studied by Buciluǎ, Caruana and colleagues in 2006, and "soft targets" were part of the early neural-net folklore. Hinton, Vinyals and Dean added the temperature trick and the name that stuck — *knowledge distillation* — and demonstrated compression of speech and MNIST models. A decade later the same mathematics re-appears, label free, as DINO's "distillation with no labels" (Caron et al., 2021).

Formally, let z_i be the teacher's logit (pre-softmax score) for class i. Plain softmax produces $q_i \propto e^{z_i}$, which for a confident teacher is nearly one-hot. *Temperature scaling* softens it:

$$q_i^{(T)} = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}, \tag{12.2}$$

where $T \geq 1$ is the *temperature*:

- T=1 is the ordinary softmax;
- larger T flattens the distribution;
- $T \to \infty$ makes it uniform.

The student is trained to match $q^{(T)}$, using cross-entropy between the two distributions or, equivalently, the KL divergence $D_{\text{KL}}(q^{(T)} \,\|\, p^{(T)})$, where $p^{(T)}$ is the student's own output at the same temperature.

Why bother? A full distribution says much more than its top answer. "70% cat, 25% dog, 5% car" also tells the student that cats look like dogs and not like cars. Because every example carries that extra information, far less data (labelled or not) is needed to pass the teacher's knowledge to the student. Figure 12.6 shows what T does to a concrete set of logits.

Worked Example — Distilling Fashion-MNIST at $T = 2$, and why the loss is multiplied by T^2

The companion case study distills the width-16 Fashion-MNIST teacher of the self-training example into a student half its size (width 8), using softened targets at temperature $T = 2$ plus ordinary cross-entropy on the 1,000 labeled samples:

$$\mathcal{L} \;=\; T^2\, D_{\mathrm{KL}}(\boldsymbol{q}_T^{(T)} \,\|\, \boldsymbol{q}_S^{(T)}) \;+\; \alpha\, \mathcal{L}_{\mathrm{CE}},$$

where $\boldsymbol{q}_T^{(T)} = \mathrm{softmax}(\boldsymbol{z}_T/T)$ and $\boldsymbol{q}_S^{(T)} = \mathrm{softmax}(\boldsymbol{z}_S/T)$ are the two temperature-scaled distributions. Consider the teacher's judgment on an ambiguous garment image: 51 % shirt, 49 % coat. An arg max target converts this to 100 % shirt; the teacher's actual statement — "this is a shirt, but only barely" — and in particular the information that shirt and coat are mutually confusable, is destroyed. The soft target preserves it, and the student inherits the similarity geometry directly.

The T^2 prefactor looks cosmetic and is not; it keeps the gradient scale comparable across choices of T. With $\sigma_i(\cdot)$ the ith softmax output, differentiating cross-entropy through temperature-scaled logits gives

$$\frac{\partial}{\partial \boldsymbol{z}_S}\, D_{\mathrm{KL}}(\boldsymbol{q}_T^{(T)} \,\|\, \boldsymbol{q}_S^{(T)}) \;=\; \frac{1}{T}\,(\boldsymbol{q}_S^{(T)} - \boldsymbol{q}_T^{(T)}).$$

As T grows, each softened probability approaches uniformity: the first-order expansion of Eq. (12.2) is $q_i^{(T)} \approx \frac{1}{K}(1 + (z_i - \bar{\boldsymbol{z}})/T)$ with $\bar{\boldsymbol{z}}$ the logit mean, so the *differences* between probabilities shrink like $1/T$. The gradient therefore carries an overall $1/T$ from the chain rule times a further $1/T$ in the differences it transmits — $\mathcal{O}(1/T^2)$ in total. Multiplying the objective by T^2 cancels this, so that changing the temperature reshapes *what* the student learns instead of merely shrinking the step size on it. Leave the prefactor out and high-T distillation silently degenerates into tiny gradients that fit only the soft target's coarsest structure. Figure 12.7 plots the shirt/coat pair itself at $T = 1$ and $T = 2$.

Simplified intuition vs. more precise view

Simplified: Distillation is "copy the teacher's answer including the confidence." **More precise:** it is a way of transferring the *geometry* of the learned function. The teacher's soft outputs partition input space into regions of smooth class similarity; the student, typically smaller, fits this similarity structure rather than the harder original decision boundary. The T^2 gradient scaling in Hinton's formulation and the usual weighting of soft-target loss against any available hard labels ($\mathcal{L} = \alpha\, \mathcal{L}_{\mathrm{soft}} + (1 - \alpha)\, \mathcal{L}_{\mathrm{hard}}$) are practical necessities, not theory. Three kinds of teacher are common: a *single strong model* (the compression setting), an *ensemble* (averaged as a better teacher), or — as in DINO — an *EMA teacher* that is a running average of the student itself, in which case no labels exist

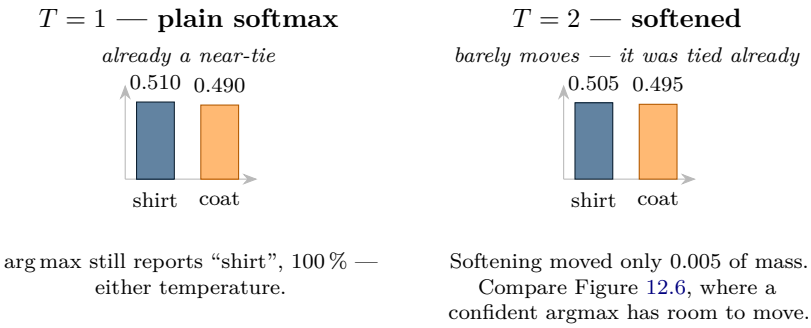


Figure 12.7. The Fashion-MNIST shirt/coat example of the worked example, at the two temperatures the loss actually uses. The text gives only these two class probabilities (the other eight Fashion-MNIST classes are not specified, so they are omitted here rather than invented); raising T from 1 to 2 moves barely 0.5 percentage points, because this teacher output is already nearly tied at $T = 1$ — unlike the confident, 84 %-on-one-class illustration of Figure 12.6, where a much larger reserve of "wrongness" is waiting to be revealed as T grows.

How to read it: Both bars barely move between the two panels — and that is the point of this particular *example. The teacher was already saying "shirt, but only barely" at $T = 1$; temperature reveals dark knowledge only where the raw softmax was hiding it behind a confident* arg max. *Here the confusion was never hidden, so there is little left for T to uncover.*

anywhere and distillation *is* the learning objective; see Figure 12.10 for *multiple* teachers of different kinds.

What changed — From Noisy Student to Knowledge Distillation

What stayed. A teacher, a student, and a loss that pulls the student toward the teacher's output — both are instances of the student–teacher template opening this chapter. **What changed.** The *granularity* of the target. Noisy Student (and pseudo-labeling before it) reduces the teacher to a single class index; distillation keeps the full distribution, softened by temperature. This is a change to *how much of the teacher's answer survives*, not to the teacher/student machinery itself — you could run Noisy Student's exact training loop and simply swap its hard cross-entropy for Eq. (12.2)'s soft one. **What it cost.** A new hyperparameter (T) with a subtle failure mode (the T^2 correction) invisible unless you derive it, plus a new inherited risk: the student now matches the teacher's *whole* distribution, so it faithfully reproduces systematic wrongness too, not just argmax errors.

Key takeaway — Distillation in one line

An argmax label answers one question ("what is it?"); a temperature-softened distribution answers a second one for free ("what else did the teacher almost think?"), and that second answer is worth more supervision, per example, than the first.

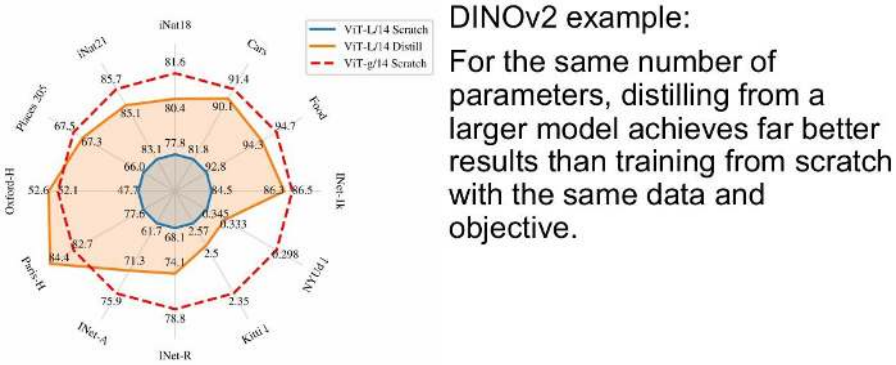


Figure 12.8. DINOv2's distillation result, reproduced from the paper (Oquab et al., 2024) (Lecture 08, slide 15). Each spoke is a downstream benchmark (classification, retrieval, depth); the orange region is a ViT-L/14 *distilled* from the larger ViT-g/14 teacher (red dashed), the small blue region is the *same* ViT-L/14 architecture trained from scratch on the same data and objective. Same parameter count, same data, same loss — the only difference is where the training target came from.

Source: Lecture 08, slide 15; figure from Oquab et al. (2024).

How to read it: compare the orange (distilled) and blue (scratch) regions at any one spoke — they are the same *architecture, so the gap between them* is *the value of distillation, isolated from every other variable.*

12.2.1 Distillation as model compression: from DINOv2 to Gemini Flash

The industrial point of soft targets is compression. Figure 12.8 reproduces a radar chart from the DINOv2 paper (Oquab et al., 2024): a ViT-L/14 *distilled* from a larger teacher consistently beats the same ViT-L/14 trained from scratch on the same data and objective — on downstream classification, retrieval, and depth estimation — nearly matching the much larger ViT-g/14 it was distilled from. Distillation buys capability per parameter.

The same trade drives today's language models. The Gemini 1.5 technical report (Gemini Team, Google, 2024) states that Gemini 1.5 Flash — the small, fast serving model — was "trained by 1.5 Pro through a process called 'distillation,' where the most essential knowledge and skills from a larger model are transferred to a smaller, more efficient model." The payoff is latency: the slide quotes the report's (Gemini Team, Google, 2024) time per output character of 1.5 ms (Flash) vs. 4.3 ms (Pro) for English, 4.3 vs. 10.9 ms for Japanese, 5.2 vs. 14.1 ms for Chinese and 1.9 vs. 4.7 ms for French (Figure 12.9).

Research Frontier

Distillation as the standard model-compression step. What began as a way to shrink an ensemble into one network is becoming a default stage of foundation-model engineering: train the largest model you can, then distill it into the sizes you can actually serve — DINOv2's distilled ViT-S/B/L variants, Gemini Flash, and DUNE (below) all follow this template. Open problems: (i) distillation in modalities other than classification

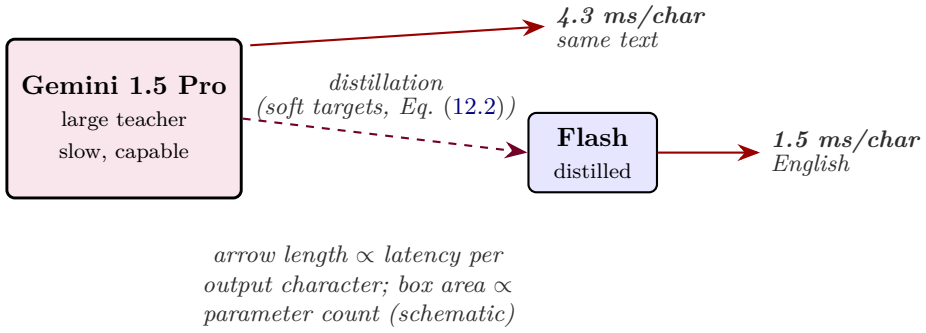


Figure 12.9. Distillation as size/latency compression (schematic, using the numbers from the Gemini 1.5 technical report (Gemini Team, Google, 2024) as quoted in Lecture 08). The large Pro model distills into the small Flash student; the serving cost of the student is roughly a third of the teacher's while retaining most capability across languages.

logits (feature maps, attention, reward signals); (ii) *self*-distillation loops in LLMs where no larger teacher exists; (iii) a principled account of when students exceed teachers (data scale and teacher correlation, as DUNE argues, are currently empirical answers, not theorems).

12.2.2 Many teachers, one student: DUNE

If one teacher transfers one skill, can several teachers of *different kinds* transfer all of theirs into a single encoder? DUNE (Sariyildiz et al., 2025) answers yes by construction. Its teachers are three frozen foundation models with incompatible objectives: DINOv2 (self-distilled 2D image features), MASt3R (3D-consistent matching and geometry), and Multi-HMR (human pose and shape estimation). Its student is one ViT encoder whose feature stream is routed through *three teacher-specific projection heads* ("TP"), each trained with its own distillation loss against its own teacher (Figure 12.10). The losses can coexist because only the projection heads are teacher-specific; the shared trunk gets gradient from all three and must *interleave* the three feature geometries in one space.

Model at a glance: DUNE Sarıyıldız, Weinzaepfel, Lucas, de Jorge, Larlus & Kalantidis, 2025

In one line	Distill three frozen foundation models with *incompatible* objectives — 2D semantics, 3D geometry, human pose — into a single ViT encoder, using one projection head per teacher.
Problem it solves	Specialist foundation models are individually excellent but cannot be combined: their training objectives and data are incompatible, so you either pick one specialist per deployment or run three separate networks.
Key intuition	Distillation only needs a teacher's *outputs*, not its objective — so teachers that could never be trained jointly can still be matched jointly, each through its own head.

Inputs	A heterogeneous 20.7M-image corpus, with *every* teacher's loss computed on *every* image (no per-teacher filtering).
Architecture	One shared ViT-Base encoder feeding three small teacher-specific projection heads ("TP"), each mapping the shared features into one teacher's output space (Figure 12.10).
Loss	Sum of three independent feature-matching losses, one per teacher, each computed only through that teacher's own head.
Training	Forward each image through the shared encoder and all three heads; backpropagate all three losses into the shared trunk, which must satisfy all three simultaneously.
What you keep	The shared encoder only — the three heads and all three teachers are discarded after training.
Strengths	One ViT-Base matches or beats *each* ViT-Large teacher on that teacher's own benchmark family; per-teacher data filtering turns out to be unnecessary — train every loss on everything.
Main failure modes	The student's ceiling is the *union* of its teachers — a capability none of the three teachers has cannot appear; it also inherits each teacher's blind spots on that teacher's weak data.
Leads to	The chapter's closing thesis (Section 12.5): distillation as a general-purpose compiler from heterogeneous frozen models into one deployable encoder.

Why does the student need separate heads? The case study uses two very different teachers:

- a Fashion-MNIST classifier, whose output is a list of 10 class probabilities;
- a fixed *Sobel edge filter*, whose output is a 7×7 map of edges.

These two outputs are different kinds of object. You cannot average a list of probabilities with an edge map, and there is no single loss that compares the student to both. So the student gets **one shared encoder and two heads**:

- a **semantic head** (one linear layer) trained with KL divergence to match the classifier;
- an **edge head** (a small convolution) trained with L_1 to match the edge map.

Each head translates the shared features into its own teacher's language. Both losses train the *same* encoder, which therefore has to learn features from which both "what garment is this?" and "where are its edges?" can be read. After training the heads are thrown away; the encoder is the product.

Swap the edge filter for MASt3R, the linear head for DINOv2's projector, and the small encoder for a ViT, and this toy is exactly DUNE (Figure 12.10).

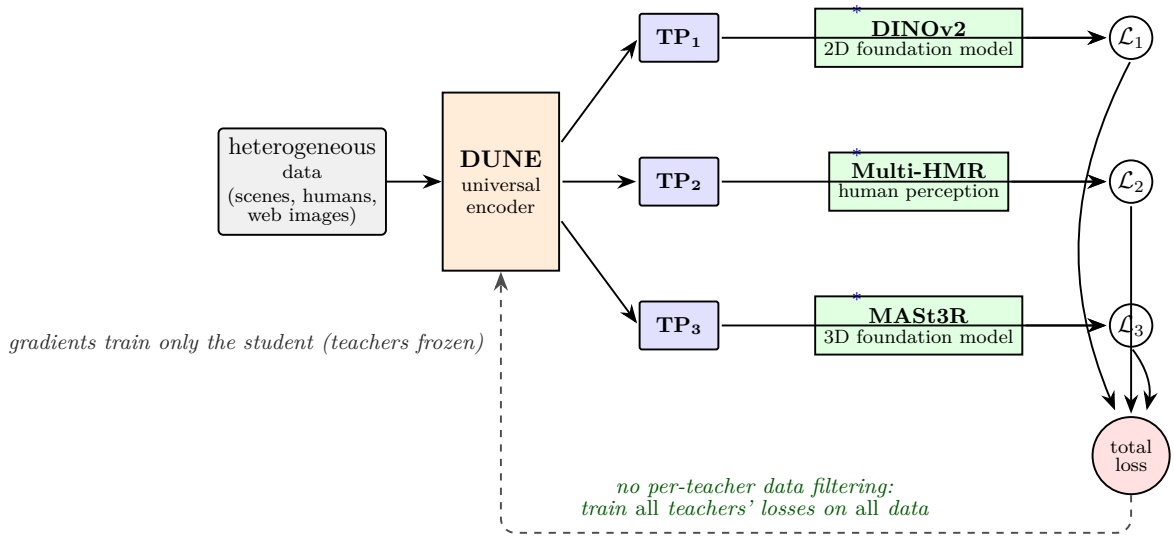


Figure 12.10. DUNE multi-teacher distillation (Sariyildiz et al., 2025) (Lecture 08, slides 17–22). One student encoder sees heterogeneous data and is pulled toward three frozen, mutually incompatible teachers; each branch $\mathrm{TP}_i \to \mathcal{L}_i$ aligns the student's projected features with teacher i's feature space. Central empirical finding: per-teacher data filtering is *unnecessary* — every teacher's loss can be computed on every sample. (Frozen teachers indicated by snowflake-style markers in the original figures.)

Paper Analysis: DUNE (Sarıyıldız et al., 2025)

Paper. *DUNE: Distilling a Universal Encoder from Heterogeneous 2D and 3D Teachers* (Sarıyıldız et al., NAVER LABS Europe, 2025). **Problem.** Foundation models specialized per task (2D semantics, 3D geometry, human pose) are individually excellent but cannot be composed: their training objectives and data are incompatible. **Prior work.** Per-task pretraining (DINOv2, MASt3R, Multi-HMR); multi-teacher orchestration such as AM-RADIO (Ranzinger et al., 2024). **Insight.** Distillation decouples *what* is learned from *how* it was learned: the student never runs the teachers' objectives, it only matches their outputs. **Method.** Single ViT-Base encoder, three teacher-specific projection heads, per-teacher feature-matching losses applied to *all* samples of a heterogeneous 20.7M-image corpus. **Experiments/Results.** The student matches or exceeds *each* teacher on its own benchmark family — 2D (ADE20k mIoU, NYUd depth RMSE), human pose (BEDLAM), and 3D camera-relative pose (MapFree AUC) — while being a ViT-Base against ViT-Large teachers; through data scale and *teacher correlation* the student can outperform all three teachers on several axes. **Limitations.** The student's ceiling is set by the union of the teachers; a capability absent from all of them cannot appear. It also inherits each teacher's biases on that teacher's weak data. **What to remember.** Distillation is a *compiler* from many heterogeneous frozen models into one deployable encoder; filtering which data goes to which teacher was hypothesized to matter and empirically does not — train each loss on everything.

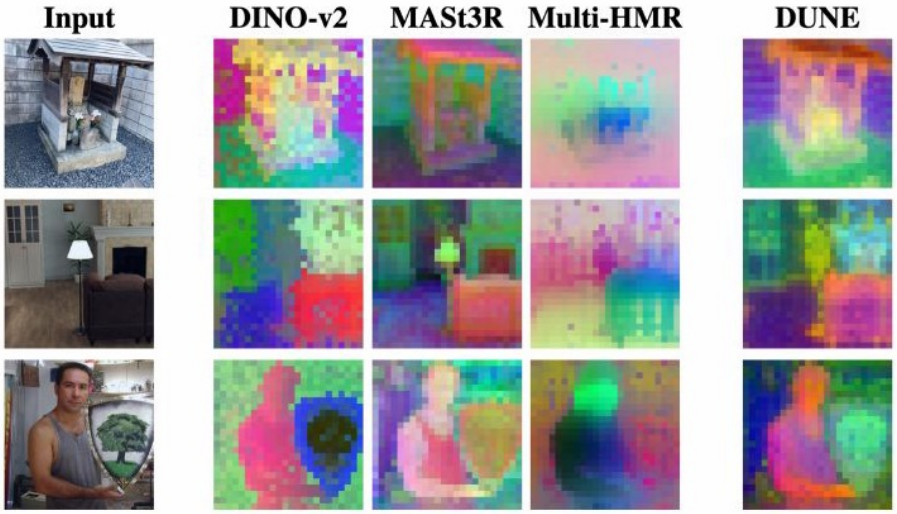


Figure 2. **PCA visualization of encoder outputs.** Given an image, we extract patch embeddings from the encoders of the teacher models and our student, and reduce their dimension to 3 via PCA.

Figure 12.11. What the shared encoder actually learns, reproduced from the DUNE paper (Sariyildiz et al., 2025) (Lecture 08, slide 19). Each row is one image; each column is a false-colour PCA projection of that model's patch features down to 3 dimensions (so similar colours mean similar features). DINOv2's columns pick out 2D semantic parts, MASt3R's pick out 3D-consistent structure, Multi-HMR's pick out the person; DUNE's column — from *one* encoder trained against all three — visibly carries traces of all three kinds of structure at once.

Source: Lecture 08, slide 19; figure from Sariyildiz et al. (2025).

How to read it: scan across one row: the four teacher/student columns are different views *of the same image, and DUNE's column is the one trained to be simultaneously compatible with the other three.*

What changed — From single-teacher distillation to DUNE

What stayed. The core move — a frozen teacher's output becomes a training target for a student, via a feature-matching loss. Each of DUNE's three branches, taken alone, is ordinary distillation. **What changed.** *Plurality.* One teacher becomes three, and the three disagree by construction (they were trained for different tasks on different data). The fix is architectural, not mathematical: a teacher-specific head per teacher, so the shared trunk never has to resolve a KL divergence against an edge map or an L_1 loss against a class posterior directly. **What it cost.** A new empirical question with no a priori answer — does every teacher's loss need to be restricted to the data that teacher understands, or can all three run on all the data? DUNE's result (filtering is unnecessary) is a finding, not a guarantee; the generic risk is that a poorly matched teacher injects noise into its branch on data it was never good at.

Key takeaway — DUNE in one line

> Distillation does not require one teacher, or compatible teachers — only one small head per teacher standing between a shared trunk and each teacher's private notion of what the output should look like.

This is the modern restatement of the chapter's opening question. The data that "does not fit our loss objectives" — posed humans, 3D pairs, generic web images — is simply fed through the frozen teachers, and each teacher converts whatever it understands of that data into a target the shared student can consume. Inheritance of skills replaces compatibility of objectives.

12.3 Cycle Consistency: Learning Without Pairs

12.3.1 Paired versus unpaired data: a taxonomy

Both self-training and distillation only move knowledge *within* one data distribution (or several teachers covering it). A different limitation arises when we have two *domains*, say Monet paintings and landscape photographs, and want to translate between them. The taxonomy in Figure 12.12 sorts this out along two axes: do *paired correspondences* exist, and does learning them need *human labels*?

Common Confusion

Paired data is not self-supervised — strictly. A dataset of aligned sketch/photo or day/night pairs used in conditional generation is supervision with a fancy target space: a human (or a sensor arrangement) had to create the pairing. The lecture makes the boundary precise: paired data counts as SSL only when pairs can be *mined without manual effort* — super-resolution (downgrade a photo to its low-resolution input), augmentations (two views of one image). Paired-by-construction data such as Pix2Pix's datasets is strictly *semi-supervised* in this taxonomy. Truly *unpaired* data — two sets with no known matches — is where CycleGAN operates (Figure 12.12).

Key takeaway — Paired vs. unpaired, in one line

> "Self-supervised" is not a property of whether pairs exist — it is a property of *who made them*. A program mining pairs from one image (augmentations) or from one measurement (super-resolution) costs nothing; a human aligning a sketch to a photo costs an annotator. CycleGAN's whole contribution is what to do when neither option is available: no pairs, and none mineable, at all.

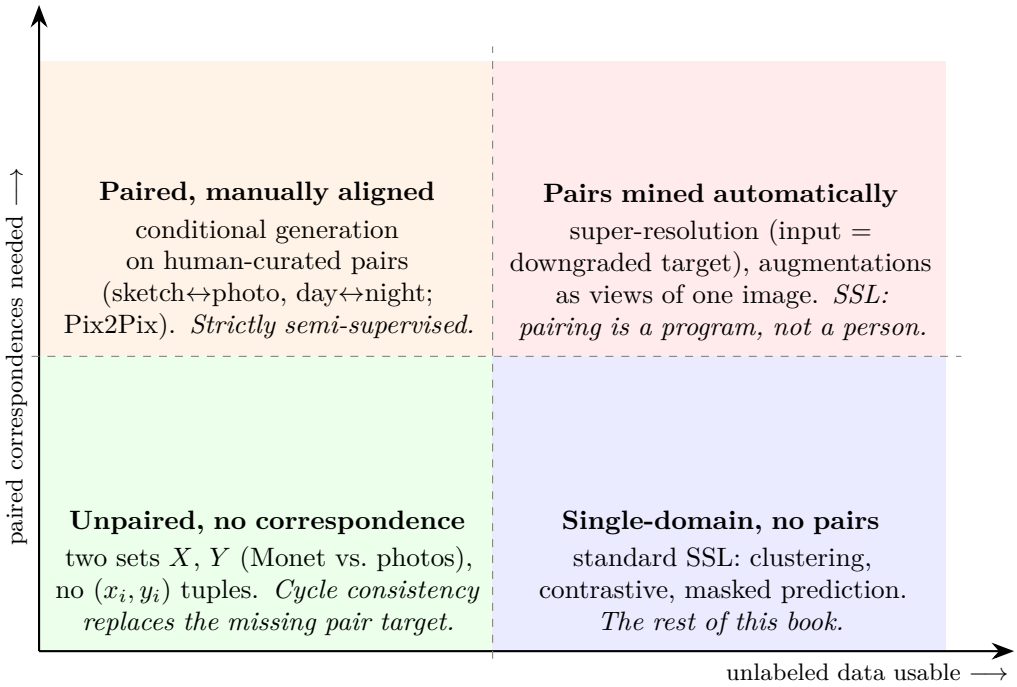


Figure 12.12. Paired versus unpaired data (Lecture 08, slide 24). The horizontal axis asks whether the training signal can be built without human labels; the vertical axis asks whether aligned pairs exist. CycleGAN lives in the lower-left quadrant; super-resolution and augmentation-based methods are "paired" versions that *do* count as self-supervised because the pairing is automatic.

12.4 CycleGAN: Two Intertwined Adversarial Games

12.4.1 The constraint that replaces data

With no pairs, the supervised target for a translation $x \in X \to Y$ simply does not exist. Zhu et al. (2017) replace it with a *structural* constraint rich enough to pin down the mapping: whatever $G : X \to Y$ produces must (i) look like a real sample of Y to a discriminator D_Y, and (ii) survive a round trip: mapping it *back* with a second generator $F : Y \to X$ must return the original, $F(G(x)) \approx x$. Symmetrically for $y \in Y$: $G(F(y)) \approx y$. This is *cycle consistency* (Figures 12.13 and 12.14).

Model at a glance: CycleGAN	Zhu, Park, Isola & Efros, 2017
In one line	Learn two generators $G : X \to Y$ and $F : Y \to X$ and two discriminators, and require translating there and back to return the original image.
Problem it solves	Image-to-image translation between two domains with no paired examples at all — the bottom-left, "unpaired" quadrant of Figure 12.12, where no supervised target exists for any single image.

Key intuition	A correct translation is invertible: if G throws away information a real translation would keep, no F can reconstruct the original from what G left behind.
Inputs	Two *unpaired* image sets X and Y — no correspondence between any x_i and any y_j is known or needed.
Architecture	Two generator networks (G, F), two discriminator networks (D_X, D_Y), one per domain (Figure 12.13).
Loss	Two adversarial (GAN) losses, one per direction, plus a weighted L_1 cycle-consistency term (Eq. (12.3)).
Training	Alternate discriminator and generator updates as in a standard GAN, with every generator update also carrying gradient from the round-trip reconstruction error in both directions.
What you keep	Typically one direction of translation (G or F) for deployment; the discriminators and the unused generator are training scaffolding only.
Strengths	Needs no paired data anywhere; produces convincing translations (Monet↔photo, zebra↔horse) purely from two unrelated image collections.
Main failure modes	No geometric invariance (fails when domains differ in shape, not just texture); steganographic hiding (see the failure-modes box below); mode collapse; poor behaviour on content rare in the target domain.
Leads to	CUT and contrastive unpaired-translation variants, which replace the second generator with a patch-level feature-matching loss; more broadly, this chapter's closing thesis that a structural constraint can stand in for a missing label.

In plain words — Why a round trip can replace a paired example

You cannot ask a network "does this Monet-style output match the photo it came from?" — there is no ground-truth Monet of that exact scene to check against. But you *can* ask a cheaper, related question: "if you translate this photo to a painting and then translate that painting back, do you get the same photo you started with?" Getting back what you started with does not require ever having seen the "correct" painting — it only requires that the two translations undo each other. That is a constraint you can check on *any* photo, with no painting ever labeled as its match, which is exactly why it substitutes for a paired dataset that does not exist. It is not a perfect substitute (see the failure-modes box below) — but it is a supervision signal built entirely out of arithmetic on the model's own outputs, with zero human alignment required.

The full objective is the sum of the two GAN losses plus a weighted cycle term:

$$\mathcal{L}(G,F,D_X,D_Y) = \underbrace{\mathcal{L}_{\text{GAN}}(G,D_Y,X,Y)}_{\text{translate } X\to Y \text{ convincingly}} + \underbrace{\mathcal{L}_{\text{GAN}}(F,D_X,Y,X)}_{\text{translate } Y\to X \text{ convincingly}} + \underbrace{\lambda\,\mathcal{L}_{\text{cyc}}(G,F)}_{\text{round-trip is the identity}}, \tag{12.3}$$

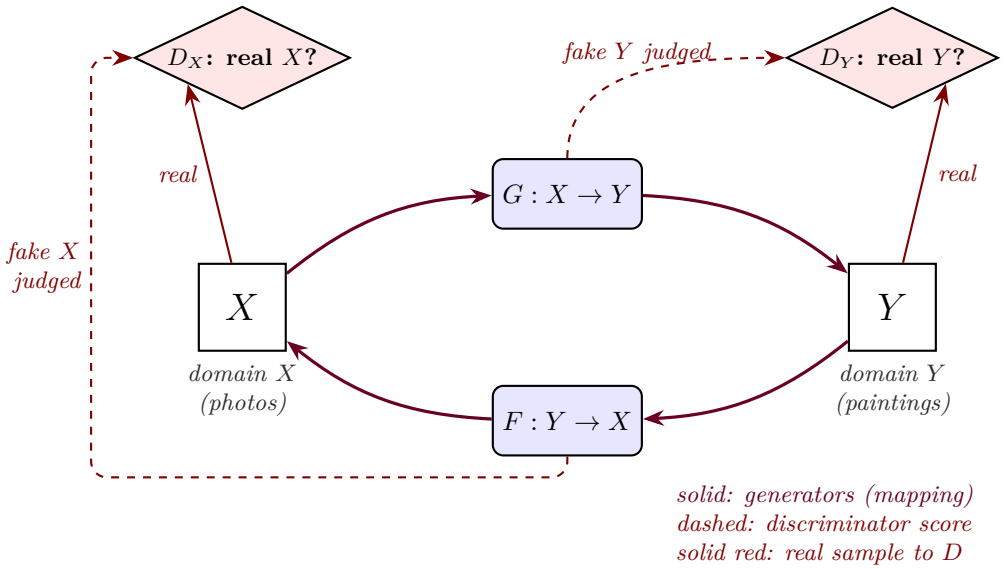


Figure 12.13. CycleGAN couples two adversarial games (Zhu et al., 2017). Generators $G : X \to Y$ and $F : Y \to X$ translate in opposite directions; domain-specific discriminators D_X, D_Y police whether translated samples are indistinguishable from real ones. The generators each try to fool one discriminator; both are additionally bound by cycle consistency, which the figure cannot show alone — see Figure 12.14.

The two ingredients of CycleGAN's loss. **1. The cycle term** — "go there and back, and you should end up where you started":

$$\mathcal{L}_{\text{cyc}}(G, F) \;=\; \mathbb{E}_{x\sim p_X}\big\|F(G(x)) - x\big\|_1 \;+\; \mathbb{E}_{y\sim p_Y}\big\|G(F(y)) - y\big\|_1.$$

Translate a horse photo x into a zebra with G, back into a horse with F, and measure the pixel-by-pixel difference from the original x. Do the same starting from a zebra y. $\|\cdot\|_1$ just adds up the absolute pixel differences.

2. The GAN term — "the translation must look real":

$$\min_G \max_D \; \mathbb{E}_{y\sim p_Y}[\log D(y)] \;+\; \mathbb{E}_{x\sim p_X}\big[\log(1 - D(G(x)))\big],$$

where the discriminator D tries to tell real zebras from translated ones, and G tries to fool it. There is one such term for each direction.

The weight λ (10 in the original paper) sets the balance: how much "must return to the original" matters compared with "must look real". Neither alone is enough — realism alone could produce any convincing zebra, unrelated to the input horse; the cycle term alone could do nothing at all and still return perfectly.

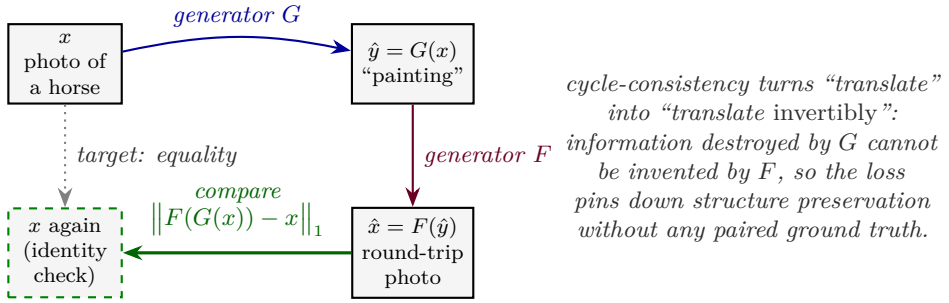


Figure 12.14. A four-state cycle for one image (worked example in text). The green comparison edge is the forward half of $\mathcal{L}_{\text{cyc}}$; the mirrored half runs $y \to F(y) \to G(F(y)) \to y$. Both must hold for the full objective (12.3).

12.4.2 A toy cycle, done by hand

Worked Example — Cycle consistency as an identity check

Take the smallest possible round trip with four states and ask what the loss enforces (Figure 12.14). Generator G maps a photo x of a horse to a painting-like image $\hat{y} = G(x)$; F maps it back, $\hat{x} = F(G(x))$. The identity term demands $\hat{x} = x$. Now imagine a "cheating" G that produces a convincing Monet *regardless of input* (mode collapse): $\hat{y}$ fools D_Y, but no F can reconstruct the *particular* horse x from an output that contains no trace of it — the cycle loss explodes. Conversely, a G that encodes all of x (say, by hiding it as invisible high-frequency texture in $\hat{y}$) lets F reconstruct x perfectly while also fooling D_Y: cycle-consistent *and* adversarially valid. The constraint eliminates content-destruction as a strategy; it cannot eliminate information-hiding.

Paper Analysis: CycleGAN (Zhu, Park, Isola, Efros, 2017)

Paper. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* (Zhu, Park, Isola, Efros, 2017). **Problem.** Learn image translation (Monet↔photos, zebras↔horses, summer↔winter) with *two sets and no pairs* — the strictly unpaired quadrant of Figure 12.12. **Prior work.** Paired conditional GANs (Pix2Pix); single-domain GANs. **Insight.** The missing pair supervision can be approximated by a *closure* property: whatever a translation does, its inverse must undo. **Method.** Two generators G, F; two domain discriminators D_X, D_Y; sum of two GAN losses plus a weighted L_1 cycle-consistency loss (12.3). **Experiments/Results.** Successful translation across diverse artistic and style transfer domains; iconic Monet↔photo panels. **Limitations.** See the failure-modes box: no geometric invariance, steganographic hiding, failure on rare content. **Later work.** CUT and contrastive variants replace the dual mapping with patch-level feature matching; DUNE (above) demonstrates how far the *distillation* alternative has come. **What to remember.** Where pairs don't exist, symmetry constraints can behave like supervision: learn a map *and* its inverse and require them to compose to the identity.

Figure 12.15. CycleGAN translations, reproduced from the paper (Zhu et al., 2017) (Lecture 08, slide 31). Left: Monet ↔ photograph, both directions. Right: zebra ↔ horse, both directions. No image on either side of any arrow was ever paired with its counterpart during training — the two generators and the cycle-consistency loss are the entire supervision.

Source: Lecture 08, slide 31; figure from Zhu et al. (2017).

How to read it: notice what survives translation in every panel — pose, layout, and count of the subject — and what changes: only texture and colour statistics. That split is exactly the failure mode below, seen from its successful side.

Failure Modes

Two ways CycleGAN can satisfy its loss without doing what you wanted. (i) *Geometric non-invariance.* The cycle loss measures pixel-level (L1) identity; it rewards copying pose and layout, not understanding them. Applying it to domain pairs with genuine shape differences fails reliably — zebras ↔ horses works because pose transfer is nearly pose-copy; dogs ↔ cats (different silhouettes) produces texture swapped onto the original shape. (ii) *Texture hiding / steganography.* Because $F \circ G$ must equal the identity, G is incentivized to smuggle a complete description of x into $\hat{y}$, and networks learn to hide it in imperceptible high-frequency texture: D_Y sees a convincing Monet, the cycle loss sees a perfect invertible code, and the semantic gap between the two is exactly where the model cheats. Additional, non-CycleGAN-specific hazards: mode collapse (every x maps to few outputs fits the GAN loss while the cycle term only catches *content-dependent* collapse) and poor behavior on content rare in the target domain.

12.4.3 What the cycle weight buys: a controlled collapse

The failure modes above remain plausible-but-abstract until λ is turned down to zero and watched. The case study does exactly this, on Fashion-MNIST: domain X is the dataset

as-is (bright garments on a dark background), domain Y is its photographic negative (dark garments on a bright background), and the two generators, two discriminators of Figure 12.13 are trained twice — once with $\lambda = 0$, once with $\lambda = 3$ — on otherwise identical code. The difference between the two runs is the purest demonstration of what the cycle term is for.

With $\lambda = 0$ (adversarial loss only). The outputs look *stylish*: each one copies domain Y's dark-on-bright look well enough to fool D_Y. But the garment drifts — a boot comes back as a t-shirt.

This is not a bug. The loss is doing exactly what it says. The adversarial term asks $\hat{y} = G(x)$ only one question: "would D_Y mistake you for a sample of Y?" Answering "yes" needs *nothing* from x. On Fashion-MNIST a t-shirt passes as easily as a boot. Since no term ever compares the output with the input, turning a boot into a t-shirt costs nothing.

The round trip then fails badly: $F(G(x))$ gets back the *style* of x but none of its *shape*, because the shape could only reach F through $\hat{y}$ — which has lost it.

With $\lambda = 3$, the style transfer survives but the garment identity does too. The $\|F(G(x)) - x\|_1$ term converts each translated pixel grid into a message F is required to be able to read; erasing the boot shape becomes expensive in exactly the unit — reconstruction error — that pains the optimizer. The two runs bracket the design space beautifully: the adversarial term pins the output to the *style manifold* of Y, the cycle term pins it to the *content fiber* of x, and neither constraint has a word to say about the other's guarantee.

The diagnostic the demo implants is the one the exam loves: *realistic outputs, failed round trips* is the signature of an under-weighted or missing $\lambda\mathcal{L}_{\text{cyc}}$. Raising the cycle weight, checking that the reconstruction is $F(G(x))$ against x (not against $G(x)$ by accident), and only then blaming the discriminator, is the debugging order the $\lambda = 0$ run teaches by negative example.

What changed — From distillation/DUNE to CycleGAN

What stayed. The chapter's underlying move: replace a missing label with a constraint the model must satisfy. **What changed.** *Everything about where the constraint comes from.* Distillation and DUNE still need a teacher — a model already trained on something. CycleGAN needs no teacher and no pretrained model of any kind: the constraint (round-trip identity) is purely structural, derived from the shape of the problem (two domains, a mapping and its claimed inverse) rather than from any model's prior knowledge. This is the widest jump in the chapter's lineage — from "borrow a signal from a model that already knows something" to "manufacture a signal from geometry alone." **What it cost.** A weaker guarantee. A distillation target is *some* model's real belief about the data; a cycle-consistency target is satisfiable by cheating (Figure 12.14's steganography argument) in a way pure distillation is not, because nothing in the loss ever compares the output to an independent notion of correctness.

Key takeaway — CycleGAN in one line

When there is no teacher and no paired data, ask the model to be its own check: require a round trip through both directions to return home, and optimize until it does — knowing that "returns home" is necessary for a faithful translation but is not sufficient, since a translation that merely hides the input also returns home.

12.5 Why Self-Training Actually Works

Self-training, distillation and cycle consistency share one mechanism at different granularities: *a constraint on the learned function stands in for missing labels.* Self-training constrains the student's decision boundary to match the teacher's; distillation constrains it in function space (the soft similarity between outputs); cycle consistency constrains it through invertibility. Each is a *compression* of information already in the world — in a pretrained model, an unlabeled corpus, or the symmetry of two domains — into a gradient signal. The failure modes are therefore all the same failure: the constraint is satisfied by something other than the intended structure — confidence in wrong answers, hidden texture, a student limited by its teachers' union.

Looking ahead: world models. The lecture's third thread goes one step further. Suppose an encoder (say DINOv2) maps observations o_t into a space where *distance means something.* Then we can train a model to predict the *next encoding*:

$$\mathcal{L}_{\text{pred}} = \|p_\theta(\text{enc}(o_{t-H:t}), a_{t-H:t}) - \text{enc}(o_{t+1})\|^2.$$

In words: given the recent encoded observations and actions, guess the encoding of what comes next. An agent can then plan at test time by searching for the actions whose predicted outcome lands closest to the encoding of a goal. That is the next chapter (DINO-WM, zero-shot planning). It is distillation turned inward: instead of one model teaching another, a frozen representation teaches a model of how the world changes.

Key takeaway — The chapter in one line

Every method here answers the same question — "what stands in for a label that does not exist?" — with a different substitute: the model's own confident guess, the model's own full distribution, or a structural symmetry the model is forced to respect. Every substitute is also a new way to be fooled, and knowing the substitute tells you exactly where to look for the failure.

12.6 Chapter summary

This chapter answered one question three times, each time further from a human label.

Self-training (Section 12.1) keeps the hole in the loss where a label would go, and fills it with the model's own best guess. It is cheap, and it worked in Lee's small MNIST experiment. But it is fragile the way anything that grades its own homework is fragile: *confirmation bias*, drawn in Figure 12.3 as a loop with no outside check.

Noisy Student showed that the fix is not caution but *difficulty*. Make the student bigger and its task harder (noise), and the loop that could have locked in errors instead corrects them, generation after generation.

Distillation (Section 12.2) keeps more of the teacher's answer instead of discarding it at arg max. A temperature T is the dial that makes a teacher's *near-misses* — the "dark knowledge" of which wrong classes were almost right — numerically visible to a loss function. This is the mechanism behind real compression pipelines (DINOv2's distilled encoders, Gemini Flash) and it scales past a single teacher: DUNE distils three foundation models with incompatible objectives into one encoder by giving each teacher its own head, and finds that the student can end up better than any one of them.

Cycle consistency (Sections 12.3–12.4) answers the question when there is no teacher at all — only two unrelated piles of images. CycleGAN's two coupled GANs and round-trip loss show that a purely structural requirement (translate there, translate back, arrive home) can substitute for a paired dataset that was never collected, at the cost of a guarantee: "arrives home" does not rule out a generator that hides the input rather than understanding it.

Table 12.1 collects the chapter's vocabulary in the order it appeared.

Key takeaway — Before moving on

> Keep one question in your pocket for every method in the rest of this book: *if there is no human label here, what exactly is standing in for one — and what would have to go wrong for that stand-in to be satisfied without the thing you actually wanted?*

12.7 Exercises

Check Your Understanding

1. **(Level 1)** What distinguishes a pseudo-label from a ground-truth label in the training loss?
2. **(Level 1)** In one sentence each: why does temperature $T>1$ soften softmax, and what information becomes visible?
3. **(Level 1)** Name the three kinds of noise Noisy Student applies *to the student*.
4. **(Level 2)** Lee (2013) used 600 or 1000 labeled MNIST digits plus tens of thousands of unlabeled ones. Restate the measured gain, and say in one line why adding unlabeled data cannot help a *purely* supervised loss.
5. **(Level 2)** Sketch the two-step cycle in self-training and label where human labels are needed and where they are not.

Table 12.1. The chapter's working vocabulary, in the order the ideas appeared.

Term	What it means, in plain words
Pseudo-label	A label the model made for itself, usually the arg max of its own prediction, used exactly as if a human had written it down.
Confirmation bias	Training on your own confident mistakes makes you more confident, and more wrong, about them next round — with nothing outside the loop to catch it.
Confidence threshold τ	Only pseudo-labels whose top probability exceeds τ are trusted; trades coverage of the unlabeled pool against the risk of admitting errors.
Noise injection	Perturbing the *student* only (augmentation, dropout, stochastic depth) so it cannot simply memorize the teacher, errors included.
Dark knowledge	The information in a teacher's non-winning class probabilities — which wrong answers were nearly right — invisible at arg max, visible after temperature softening.
Temperature T	A softmax dial: T=1 is ordinary softmax, larger T flattens the distribution and exposes dark knowledge, $T \to \infty$ approaches uniform.
Teacher-specific projection head	A small network that translates a shared encoder's features into one particular teacher's output space, so incompatible teachers can supervise the same trunk without their losses colliding.
Paired vs. unpaired data	Paired: aligned examples exist (mined for free, or collected by a human). Unpaired: two sets with no known correspondence — CycleGAN's setting.
Cycle consistency	The requirement that translating to the other domain and back returns (approximately) the original input — a structural stand-in for a paired target that does not exist.
Steganographic hiding	A generator satisfying cycle consistency by smuggling a full description of the input into imperceptible high-frequency texture, rather than by translating it honestly.

6. **(Level 2)** Compute $q^{(T)}$ at T=1 and T=4 for logits $\boldsymbol{z} = (3, 1, 0.2)$ and compare with Fig. 12.6.
7. **(Level 3)** Why does Noisy Student require the student to be *equal or larger* than the teacher and noised, rather than smaller and clean as in classical distillation?
8. **(Level 3)** In Eq. (12.3), suppose $\lambda \to \infty$. What does the model learn, and why might that still not be the identity map?
9. **(Level 3)** DUNE trains all three teacher losses on all data. Give one reason this could beat per-teacher data filtering, and one hypothetical setting where filtering should win.
10. **(Level 4)** Explain why cycle consistency *rules out* content destruction but *does not rule out* texture hiding. Construct a "cheating" G explicitly for 1-D $x \in [0, 1]$.
11. **(Level 4)** Argue, from Eq. (12.2), why cross-entropy against soft targets and the KL divergence to the soft targets give the same student gradient up to a constant.

12. **(Level 5)** Critique the claim "distillation transfers all the dark knowledge." What incentive does $D_{\mathrm{KL}}(q^{(T)} \| p^{(T)})$ give the student about classes where the *teacher* is systematically wrong?
13. **(Level 5)** Design a two-line test that would *detect* steganographic hiding by a CycleGAN model on a pair of domains where geometry must change.

Solution sketches

- **A1.** None: the loss consumes $\hat{\boldsymbol{y}}$ exactly like a human label; the difference is provenance (model output vs. human annotation) and hence error statistics.
- **A2.** Dividing logits by T compresses their range by $1/T$, pulling $q^{(T)}$ toward uniform; the relative magnitudes of *incorrect* classes (the dark knowledge) become numerically significant.
- **A3.** RandAugment-style input augmentation, dropout, and stochastic depth — all applied to the student only, on clean teacher pseudo-labels.
- **A4.** 91 %→95 % (600 labels) and 93 %→96 % (1000 labels). A labeled-data loss has zero gradient on inputs with no target.
- **A5.** (teacher frozen → pseudo-labels → train student). Human labels seed the first teacher only; everything downstream is label-free.
- **A6.** T=1: $\approx (0.84, 0.11, 0.05)$; T=4: $\approx (0.48, 0.29, 0.24)$; the bars in Fig. 12.6 visualize exactly these.
- **A7.** The goal is *improvement*, not size reduction: extra capacity lets the student absorb the noised, smoothed target better than the teacher realized it; noise is a consistency regularizer that prevents copying individual label errors.
- **A8.** The adversarial terms become negligible, so the only requirement left is $F \circ G \approx \mathrm{id}$ and $G \circ F \approx \mathrm{id}$: *any* invertible G with $F = G^{-1}$ is optimal. $G = \mathrm{id}$ (no translation at all) is one such solution, but so is any other bijection — the cycle term says nothing about *which* invertible map, and it is enforced only at training points.
- **A9.** For: correlation across teachers turns disagreements into denoising — the student averages out per-teacher per-domain idiosyncrasies. Against (*hypothetical*): data adversarial to a particular teacher's known bias (e.g. cartoon inputs for a human-pose teacher) inject pure noise into that branch; filtering should win.
- **A10.** Content destruction makes $F(G(x)) = x$ information-theoretically impossible across many x; hiding stores x's description in bits of $\hat{y}$ the discriminator is insensitive to. In 1-D: encode x in the low bits of $\hat{y} = s + \epsilon\,\mathrm{code}(x)$, where s is domain-plausible and ϵ is sub-perceptual.
- **A11.** $D_{\mathrm{KL}}(q\|p) = H(q,p) - H(q)$; $H(q)$ does not depend on the student, so $\nabla_{\text{student}} D_{\mathrm{KL}} = \nabla_{\text{student}} H(q,p)$ — identical gradient, different objective value.
- **A12.** The student imitates the teacher *including its errors*; KL rewards reproducing the wrong distribution faithfully. Self-distillation and data scale help only when the

teacher's errors are uncorrelated with what the next generation can learn — correlated error is inherited, not averaged out.

- **A13.** E.g.: (i) perturb $\hat{y}$ in ways invisible to D_Y but targeted at high-frequency bands and check whether $F(\hat{y})$ changes systematically; (ii) geometry-shift x (rotate the object) and test whether $G(x)$'s *geometry* moves correspondingly — a texture encoder passes the cycle loss either way, a true translator does not.

12.8 Concept Map

Concept Map — how this chapter connects

Backward links. This chapter generalizes threads started earlier. The arg max-and-cross-entropy pseudo-label (Ch. 3's DeepCluster treated **cluster assignments** as pseudo-labels; here a trained classifier plays that role, and DeepCluster is exactly its "clustering-as-teacher" special case (Caron et al., 2018)). The EMA teacher, centering and stop-gradient machinery of Ch. 11's DINO is distillation whose targets are the model's own past; Hinton's temperature (Eq. (12.2)) is the label-supervised ancestor of DINO's sharpening. The GAN minimax objective recalled here is the one built up in Ch. 6, and Pix2Pix's paired setting (Ch. 6) is the contrast case for this chapter's unpaired quadrant.

Forward links. DUNE's universal encoder previews Ch. 13's close cousin: world models *inside* frozen self-supervised feature spaces (DINO-WM predicts next-frame *features*, not pixels, and plans by matching the feature of a goal). The "does this count as self-supervised?" debate, opened here for paired data, returns with action-conditioned prediction in Ch. 13 and with full system-level pseudo-labeling pipelines (robots labeling their own experience) in Ch. 14. The compression thread — distillation as the bridge from largest-trainable to largest-deployable — reshapes what the scaling recipes of Ch. 11 ultimately *ship*.

The unifying frame. Across pseudo-labels, distillation and cycles the same contract holds: *use a structure constraint as a teacher when labels are absent*; the chapters around this one are all instances of choosing which structure.

CHAPTER 13

World Models and Planning in Representation Space

Make everything as simple as possible, but not simpler.
Attribution disputed; the sentiment belongs to physics, and world models are learned physics.

In this chapter

13.1 Abstraction: the goal of all feature extraction 584 • Evidence that pretrained spaces already carry semantics: the DIFT callback
13.2 World models: are they self-supervised? 587 • Case 1: the autoregressive world model — yes • Case 2: the action-conditioned model — yes, *if* actions are recoverable • Why the label matters
13.3 DINO-WM: a world model in a frozen representation space .. 592 • Architecture: observe, encode, predict • Training versus planning: two different jobs • Why plan inside frozen features? • A minimal world model to hold in your hands: CartPole with residual dynamics
13.4 Planning as test-time optimization 604 • How the best action is chosen, worked out by hand
13.5 Failure modes 613
13.6 The systems connection: who records the experience? 615
13.7 Chapter summary 615
13.8 Check Your Understanding 616
13.9 Concept Map 619

Chapter roadmap

Before you start. You need the idea of an embedding space with a meaningful distance (Chapter 2), a frozen pretrained encoder such as DINOv2 (Chapter 11), and the EMA/collapse-avoidance vocabulary of Chapter 10 — you will see it again as the reason a frozen target cannot collapse. No reinforcement-learning background is assumed; this chapter builds planning from scratch.

The idea in one sentence. If a frozen encoder already places similar scenes near each other, you can learn to predict *the next embedding* instead of the next image, and then

plan by searching for the sequence of actions whose imagined embedding lands closest to a goal embedding — no reward function, no policy network, no pixels anywhere in the loop.

What we cover.

- **Is a world model self-supervised?** (Section 13.2). The honest answer is "it depends on where the actions came from" — a three-way split that decides whether a whole method counts.
- **DINO-WM** (Section 13.3). One concrete system that commits to a frozen representation space instead of pixels, built up piece by piece: encoder, predictor, loss.
- **Planning as search** (Section 13.4). With a trained dynamics model, acting becomes "imagine many futures, keep the best one" — no policy network required.
- **What goes wrong** (Section 13.5). Predicted states drift from real ones the longer you imagine without checking — and why that one fact explains most of this chapter's design choices.

Afterwards you will be able to say precisely when an action-conditioned world model does and does not count as self-supervised; explain why predicting the next *latent* is not the same problem as predicting the next *image*, and why that difference is the whole argument for DINO-WM; walk through one round of test-time planning by hand; and name the failure mode (compounding rollout error) that most of this chapter's engineering exists to contain.

Every method in this book so far has used self-supervision to build a static artifact: an embedding space, a density, a segmentation, a similarity structure. This final chapter of the course material asks what that artifact is *for*. The lecture's answer is the most ambitious one we will see: a good enough representation space can itself become a *simulator*. Embed the current scene, embed the goal, predict how the embedding moves under your actions, then take the actions that move it toward the goal. That is a world model, and — as Lecture 08 argues in its closing nine slides — when the embedding space is a pretrained self-supervised one such as DINOv2, the whole pipeline trains without a single task-specific label and plans zero-shot.

13.1 Abstraction: the goal of all feature extraction

Recall Chapter 1's central idea: structure in the inputs can be used as a learning signal. Lecture 08 (slides 33–34) gives the other side of that coin. What we get from learning on that signal is a map

$$\Phi : \mathcal{X} \longrightarrow \mathcal{Z},$$

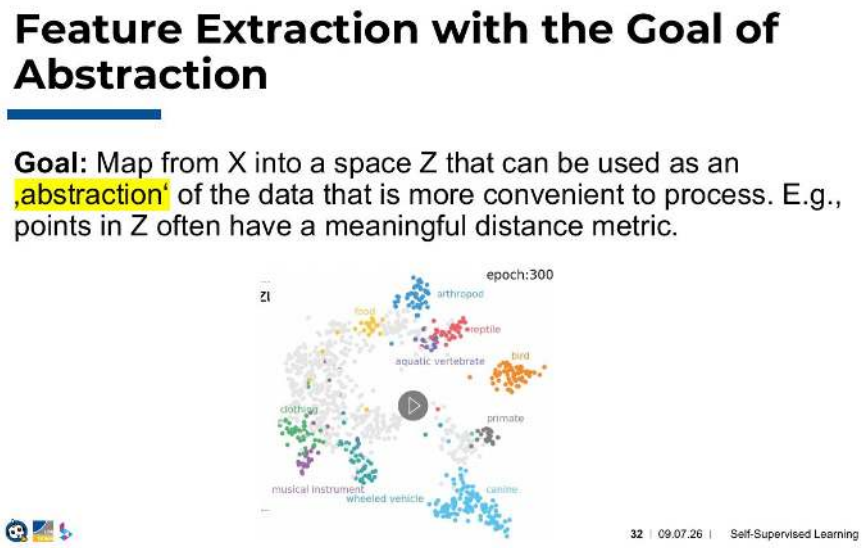


Figure 13.1. Reminder of what a learned abstraction looks like (Lecture 08, slide 33): a t-SNE view of a deep embedding space in which contiguous regions correspond to WordNet-level categories — arthropod, food, reptile, bird, aquatic vertebrate, primate, canine, wheeled vehicle, musical instrument, clothing. Neighborhoods in $\mathcal{Z}$ track semantic neighborhoods even though no label was used to shape the space.

from raw inputs $\mathcal{X}$ (images, point clouds, state trajectories) to a *representation space* $\mathcal{Z} \subseteq \mathbb{R}^d$, where d is the embedding size. The point is that $\mathcal{Z}$ is easier to work with, in two specific ways the rest of the chapter relies on:

1. points cluster by *meaning*;
2. distance is *meaningful*: the Euclidean distance $\|\boldsymbol{z}_1 - \boldsymbol{z}_2\|$ between two embeddings says something true about the inputs they came from.

Figure 13.1 shows the lecture's picture of such a space: a t-SNE map in which related words and images gather into labelled "continents". In Chapter 2 t-SNE was a diagnostic tool; here it is evidence. It shows that structure found *without supervision* is strong enough to put *canine* next to *primate* and far from *wheeled vehicle*. That idea has driven Parts I and II of this book.

What is new in this chapter: we treat positions and paths *inside* such a space like physics. $\mathcal{Z}$ becomes a state space, and we learn how actions move points around in it.

13.1.1 Evidence that pretrained spaces already carry semantics: the DIFT callback

Before we *predict* inside an embedding space, we should check that the space knows more than textures — that it knows, say, a beak and a wing play different roles. The lecture (slide 34)

Figure 13.2. Emergent semantic correspondence as evidence that self-supervised features carry semantics (Lecture 08, slide 34; method of Tang et al., 2023). A marked source point on the duck is matched purely by feature-space nearest neighbor to the anatomically corresponding point across instances (other birds), categories (mammals), and domains (drawings, statues). No correspondence labels were ever given.

points to evidence from Chapter 7: the emergent correspondences of *diffusion features*, DIFT (Tang et al., 2023). The recipe:

1. take a pretrained diffusion model (no text needed);
2. feed it two images and pull out features from the denoising U-Net at matching timesteps and stages;
3. for a point in the first image, find the nearest feature in the second image's feature grid.

Figure 13.2 shows the kind of result the lecture shows. One point marked on a duck is found — with no correspondence training at all — on other birds (cross-*instance*), on bears and cows (cross-*category*), and in drawings and statues (cross-*domain*).

Two lessons carry forward. First, these correspondences were never trained; they *emerged* from denoising pretraining, so a generic SSL objective left footprints of object-level structure in the features. Second, and this is why the lecture sets DIFT right before world models: when such features are in place, the space $\mathcal{Z}$ is no longer a black bottle of similarity. Its points have part-level meaning, and its local geometry — distance, motion, neighborhood — means something about the physical scene. World models put that local geometry to work.

Key takeaway — Abstraction, before we use it

Everything in this chapter leans on one borrowed fact: a good self-supervised encoder does not just classify, it places semantically similar things *near each other*. Once distance in $\mathcal{Z}$ means something, you can treat $\mathcal{Z}$ as a space to do physics in — which is exactly what the rest of the chapter does.

13.2 World models: are they self-supervised?

Key Concept — World model

A *world model* is a learned model of *environment dynamics*: a function that, given a description of the current state of the world (and usually a planned action), predicts the future state of the world. In the representation-space versions that concern us, both the observed and the predicted states live in an embedding space $\mathcal{Z}$, and the dynamics are learned from recorded experience rather than programmed.

Model at a glance: Latent World Model — general recipe, not one paper

In one line	Encode observations into a frozen (or separately trained) representation space, learn a small dynamics model that predicts the *next embedding* given the current one and an action, and plan by searching over actions in that space instead of pixel space. DINO-WM (Zhou et al., 2024) (below) is one instance; the wider family is sometimes called "JEPA-style" prediction.
Problem it solves	Predicting raw future observations (pixels, point clouds) wastes model capacity on scene detail no action can influence, and makes planning — which needs many imagined rollouts — too expensive to run at test time.
Key intuition	If distance in the representation space already tracks scene structure (Section 13.1.1), then a regression in that space learns *task-relevant* dynamics instead of photometric detail.
Inputs	A logged trajectory of observations and actions — no reward, no goal, no task label at training time.
The layer	An encoder $\text{enc} : \mathcal{X} \to \mathcal{Z}$ (frozen or learned) and a small predictor $p_{\boldsymbol{\theta}} : \mathcal{Z} \times \mathcal{A} \to \mathcal{Z}$.
Loss	One-step regression in $\mathcal{Z}$: $\|p_{\boldsymbol{\theta}}(\boldsymbol{z}_t, a_t) - \text{enc}(o_{t+1})\|^2$, exactly Eq. (13.4) below with the encoder made explicit.
Inference	No policy network: at test time, search over candidate action sequences, roll each through $p_{\boldsymbol{\theta}}$, and score the imagined endpoint against a goal embedding (Section 13.4).
What you keep	The predictor $p_{\boldsymbol{\theta}}$ (and the encoder, if it was trained rather than frozen) — there is no separate actor network to keep or discard.
Strengths	Task-agnostic: the same trained dynamics model can be aimed at any goal you can photograph or encode, with no retraining.

Main failure modes	Compounding rollout error under *imagined* multi-step prediction (Section 13.5); if the encoder is trained jointly with the predictor rather than frozen, the pair can collapse onto a trivial, easy-to-predict representation.
Leads to	DINO-WM, which commits to a *frozen*, already-trained self-supervised encoder — trading away joint optimization for an immunity to collapse.

The term is old — control theory has always had "models" — but in its modern neural form it is associated with learning a *compact simulation* of an environment inside which an agent can imagine, plan, or simply predict. The lecture calls world models "the current frontier for learning abstractions" (slide 35, illustrated with the Push-T manipulation arena of Figure 13.4) and then immediately asks the exam question this book has trained you to expect: is a world model self-supervised? The answer splits cleanly into two cases.

13.2.1 Case 1: the autoregressive world model — yes

The simplest dynamics model ignores actions entirely. Given a history of observations, predict the next one:

$$\hat{\boldsymbol{x}}_{t+1} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_t), \tag{13.1}$$

where $\boldsymbol{x}_t$ is the observation at time t (a video frame, a sensor reading), $f_{\boldsymbol{\theta}}$ is a trainable predictor with parameters $\boldsymbol{\theta}$, and $\hat{\boldsymbol{x}}_{t+1}$ is its guess for the next observation.

"What happens next?" is a question the data answers for free. Any recording over time (video, robot logs, audio) contains both the input and the target. Training compares the guess $\hat{\boldsymbol{x}}_{t+1}$ with the recorded $\boldsymbol{x}_{t+1}$ — a reconstruction or likelihood loss exactly like those of Chapters 4–7. Slide 36 says so plainly: an autoregressive model *can be a simple world model*, and "we can learn this in a self-supervised way from sequential data, such as videos." In Figure 13.3, this branch is clearly SSL.

13.2.2 Case 2: the action-conditioned model — yes, *if* actions are recoverable

Prediction alone does not let you *act*. For control we want a conditional dynamics model that also sees what the agent is doing:

$$\hat{\boldsymbol{x}}_{t+1} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_t; \boldsymbol{a}_0, \ldots, \boldsymbol{a}_t), \tag{13.2}$$

where now $\boldsymbol{a}_t \in \mathbb{R}^m$ is the action executed at step t (a gripper velocity, a steering command), with m the action dimension. Equation (13.2) answers the question that matters for planning: "what will happen next *if I do this*?" Slide 37 of the lecture draws this extension as a second input sequence feeding the same black wedge — and then draws the SSL question mark over that second sequence. Actions are not passively observable the way frames are. Where do they come from?

Three possibilities. (i) *Human-recorded actions*: teleoperation or hand-labeled interventions. This is supervision — the labels happen to be torques instead of class IDs — so the method is not self-supervised. (ii) *Proprioception*: a robot logs its own motor commands alongside its cameras. No external human effort is involved, and this is precisely the system-level SSL setting of Chapter 14, where the machine's own data streams provide the alignment for free. (iii) *Extracting actions from raw data*: if no action channel was recorded, one can sometimes *reconstruct pseudo-actions* from observation pairs — for example by training a small inverse-dynamics model $I_\phi(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}) \mapsto \hat{\boldsymbol{a}}_t$ that infers which action could plausibly transform one frame into its successor. The inferred action then supervises the forward model of Eq. (13.2). Whether you count case (iii) as SSL is a matter of taxonomy, and the lecture's split verdict is worth memorizing: in case (ii) and (iii) "we may call this SSL"; in case (i) we may not.

Common Confusion

World model vs. model-free RL vs. imitation. These three families share vocabulary (states, actions, goals) but differ in *what is learned*. *Model-free reinforcement learning* learns a policy $\pi(\boldsymbol{a} \mid s)$ or a value function $Q(s, \boldsymbol{a})$ directly from reward, with no explicit model of $p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{a}_t)$; it never predicts a single future frame. A *world model only* learns the dynamics $p(\boldsymbol{x}_{t+1} \mid \dots)$ or its representation-space counterpart — no reward, no policy; planning or a downstream task is bolted on later. *Imitation / behavior cloning* learns $\pi(\boldsymbol{a} \mid \boldsymbol{x})$ from demonstrations by supervised regression to recorded actions — Diffusion Policy is the canonical example of the "supervised imitation" caveat from Chapter 7 (discussed there). The boundary cases: model-free RL from proprioceptive rewards is *system-level* self-supervision (Chapter 14); an action-conditioned world model is *not* a policy — it cannot act by itself, it can only *imagine*.

13.2.3 Why the label matters

The classification sounds pedantic until you notice what it buys you. If a world model is self-supervised, then *any logged stream* — billions of hours of video, proprioceptive robot dumps, car trajectories — is training material, with no annotation bottleneck. That is exactly the pitch of the field: the world model is the learned analogue of the physics engine in a simulator, except instead of being written by a physicist it is extracted from data by the same self-supervision principles the rest of this book used for embeddings.

Slide 37 puts the whole decision in one sentence worth memorising: "either we need to record paired data (not SSL) or we can extract the action from the raw data (we may call this SSL)." Two refinements make it sharper.

(a) The half-supervised middle ground. Be clear about what is and is not in doubt.

- The **objective** — predict the next observation — is self-supervised in every version of Eq. (13.2), because the future frame is its own label.

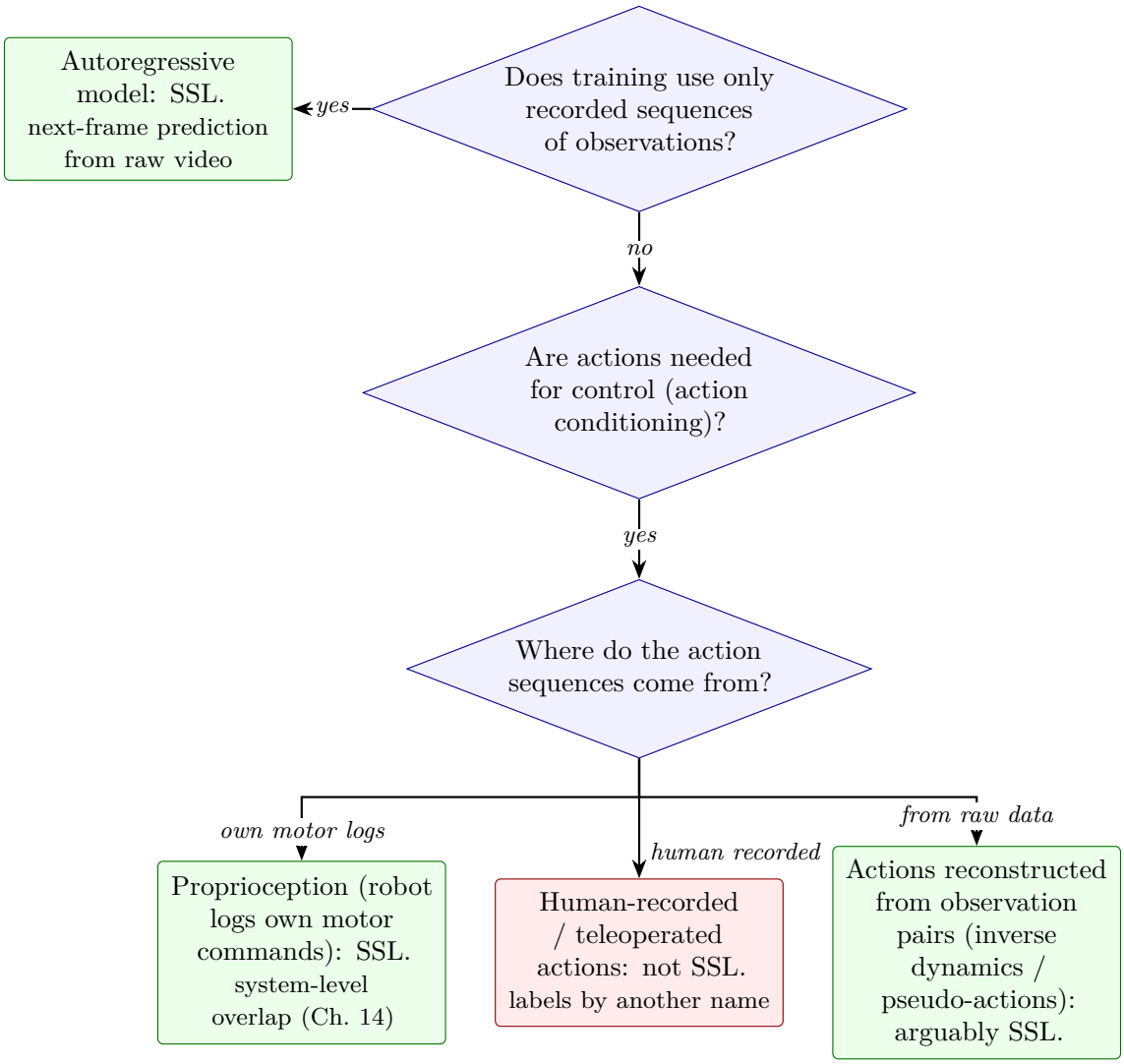


Figure 13.3. The "is it SSL?" decision tree for world models (decoding Lecture 08, slides 36–37). Plain autoregressive prediction is self-supervised because sequences of observations supervise themselves. The action-conditioned branch supervises itself only when the action channel is free: either logged by the system itself (proprioception) or recoverable from observation pairs. Human-recorded actions are labels, and the method is imitation/behavior cloning instead.

- The only possible outside supervision is the **action stream** $a_{0:t}$. If extra hardware logs the actions, the dataset is paired, so on the slide's reading it is semi-supervised — even though the objective is still SSL.
- *Proprioception* (Chapter 14) is the special case where the "sensor" is the robot itself logging its own motor commands, at no human cost. That puts the whole pipeline back on the SSL side.

The exam version (Q12 of the study questions) wants exactly this distinction: future observations can always be free targets; action–state pairs supplied from outside add supervision.

(b) Getting actions from raw pixels. The "arguably SSL" leaf of Figure 13.3 has a concrete recipe:

1. Train a small *inverse-dynamics* model $I_\phi(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}) \mapsto \hat{\boldsymbol{a}}_t$ — "what action turned this frame into that one?" — on whatever action labels exist. (A fully label-free version uses motion read straight off the video: optical flow between frames, or tracked keypoints of a hand or gripper.)
2. Run I_ϕ over any unlabelled video to tag it, after the fact, with *pseudo-actions*.
3. Train the forward model of Eq. (13.2) on the result.

Now time supplies the next frame for free and vision supplies the action for free, so by the slide's taxonomy the world model can be trained on raw video alone.

The last piece: who says what the goal is? Neither the observations nor the recovered actions define a task — someone still has to say what "done" looks like. There are three ways, and they are not equally self-supervised:

Goal given as	Example	Status
(i) coordinates	operator types a pose or joint vector	a label
(ii) a scoring rule	CartPole's weighted distance (Section 13.3.4): the upright, centred pose is not exactly reachable, so someone chose how angle trades against position	a label, once removed
(iii) a goal image	the robot's own camera photographs the desired scene; the frozen encoder turns it into $\boldsymbol{z}_g$	SSL

DINO-WM's contribution here is to replace (i) and (ii) with (iii). The task is then the same kind of object as the training data, so "set a new task" and "collect data" become the same operation.

Use this split when judging any claimed "self-supervised planner": the claim is exact for the *dynamics*, debatable for the *actions*, and depends on the *goal* channel.

Key takeaway — "Is it self-supervised?" is three separate questions

> Do not answer for the whole system at once. Ask separately about the *dynamics* (almost always yes — the next frame labels itself), the *actions* (yes if logged by the system or recovered from raw data, no if a human teleoperated the robot), and the *goal* (yes only if it is itself something the system could produce, like a photograph, rather than something a person had to write down).

13.3 DINO-WM: a world model in a frozen representation space

We now instantiate the abstract recipe with the lecture's capstone system, DINO-WM (Zhou et al., 2024) (slides 38–41). Two decisions define it. First, *where* the dynamics will be learned: not in pixel space, where every sunbeam and shadow has to be predicted, but in the *patch-feature space of a pretrained, frozen DINOv2 encoder* (Oquab et al., 2024). The Chapter 11 encoder, trained once on 142 million images, is used as-is and never updated. Second, *how* the model is used: at test time there is no policy network at all. Planning is optimization over candidate action sequences inside the learned dynamics.

Model at a glance: DINO-WM — Zhou, Pan, LeCun & Pinto, 2025

In one line	Freeze a pretrained DINOv2 encoder, train a small action-conditioned predictor on top of its patch features, and plan by searching over action sequences that steer the predicted features toward a goal image's features.
Problem it solves	World models should train on offline pre-collected trajectories, support test-time behaviour optimization, and generalize to new goals without per-task retraining — pixel-space models are expensive to roll out and waste capacity on task-irrelevant detail.
Key intuition	A representation space trained to put semantically similar scenes near each other (Section 13.1.1's DIFT evidence) is already most of the way to a good state space for dynamics — so do not learn one from scratch, borrow it.
Inputs	Offline trajectories of (o_t, a_t) pairs (no reward, no goal at training time); at test time, one goal image o_g.
Architecture	Frozen DINOv2 ViT encoder (patch tokens); a small action encoder $\boldsymbol{\phi}$; an action-conditioned predictor $p_{\boldsymbol{\theta}}$ (Figure 13.7).
Loss	$\mathcal{L}_{\text{pred}}$, Eq. (13.4): mean squared error between the predicted next embedding and the frozen encoding of the actually-observed next frame.
Inference	Model-predictive control: sample candidate action sequences, roll each through $p_{\boldsymbol{\theta}}$, score by distance to $\boldsymbol{z}_g = \text{enc}(o_g)$, execute the first action, replan (Section 13.4).
What you keep	The predictor $p_{\boldsymbol{\theta}}$ and action encoder $\boldsymbol{\phi}$; the encoder was never trained, so there is nothing to discard or keep from it beyond the frozen weights themselves.
Strengths	Zero-shot planning: retargeting to a new goal costs one forward pass of the frozen encoder, not a new training run; the encoder's size and cost are amortized across every downstream task.
Main failure modes	Compounding rollout error over long horizons; goal-embedding ambiguity (many scenes can share a similar $\boldsymbol{z}_g$); degrades outside DINOv2's pretraining distribution (Section 13.5).
Leads to	The systems-level question of Section 13.6: every ingredient of this pipeline — the trajectory log, the actions, the goal photograph — is something the system produced for itself.

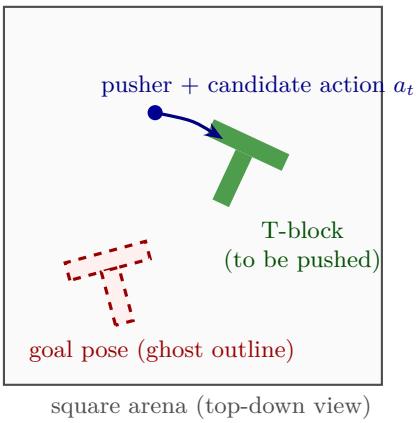


Figure 13.4. The Push-T environment (Lecture 08, slides 35–37 and 41), redrawn schematically. A top-down square arena contains a green T-shaped block and a circular pusher; the task is to move the pusher (blue action arrows) so that the block slides to the dashed goal pose. Because contact forces are local and unintuitive — where you push the T determines how it rotates — Push-T is a classic stress test that a learned model is preferable to a hand-engineered controller.

The running environment, Figure 13.4, is Push-T: a T-shaped block on a 2D tabletop must be pushed by a circular pusher to a goal pose — the standard manipulation benchmark used throughout the imitation-learning literature (Chi et al., 2023). Observations o_t are rendered top-down RGB images; actions a_t are 2D motions of the pusher applied per control step.

Before wiring the blocks together, two pictures make the design feel obvious. The first explains *why the state must be an embedding.*

When you drive home, your eyes take in every leaf of every tree on the way, but your planning ignores all of it. You track the road edge, the gap to the car ahead, the colour of the traffic light. Predicting the drive pixel by pixel would not just be expensive — it would bury the few signals that matter under thousands that do not.

That is the case for modelling dynamics in $\mathcal{Z}$ rather than in $\mathcal{X}$. It also answers the examiner's favourite question about this block: pixel-level world models fail for a *capacity* reason, not an aesthetic one. They spend their parameters on shadows, textures and lighting that no action can change. (When a lab light flickers, the shadow sliding across the Push-T table changes a large fraction of the RGB values and none of the physics.) The frozen DINOv2 embedding was trained to ignore exactly this kind of lighting change (Chapter 11), so in its space the flicker has almost vanished.

The CartPole case study does the same thing more crudely: its standardisation removes the units and offsets of the simulator's coordinates — the only "nuisance" that tiny state space has.

Figure 13.5 puts the two choices side by side so the trade is visible at a glance, not just argued in prose.

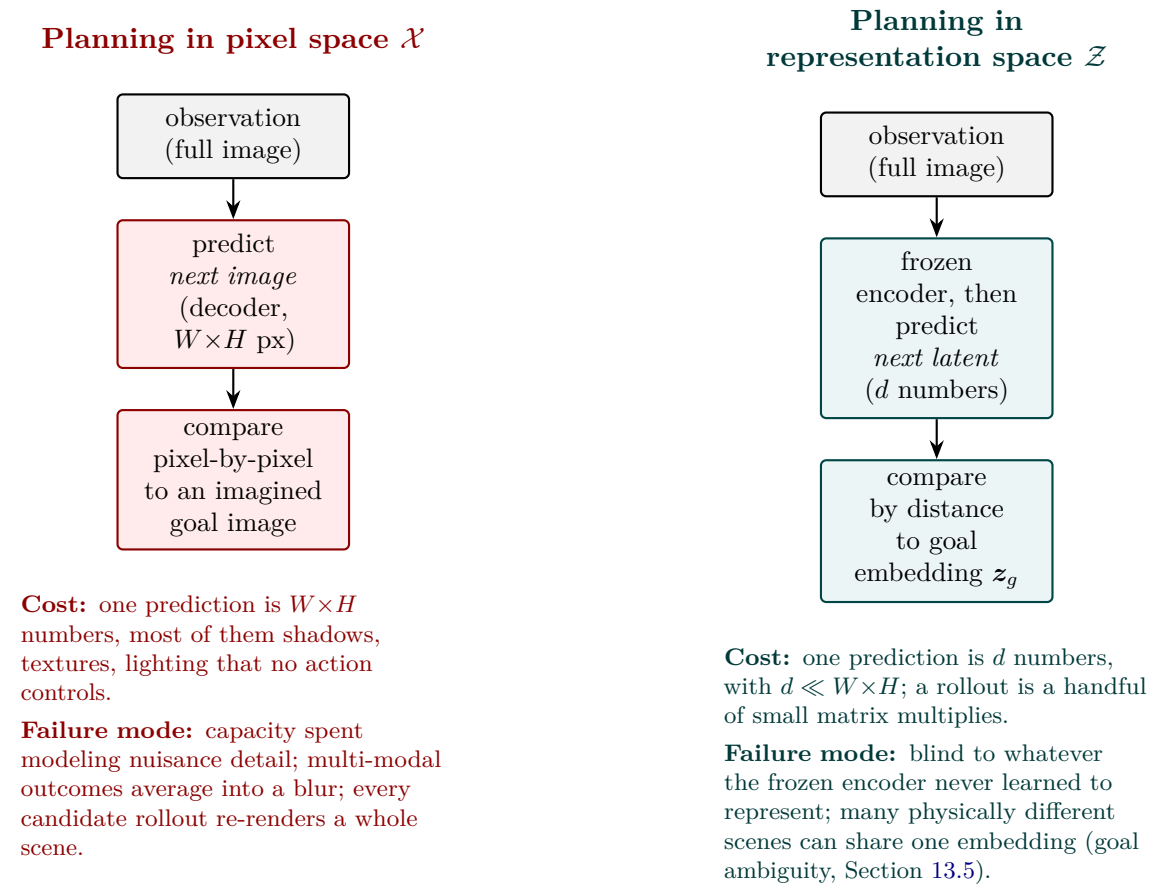


Figure 13.5. The same planning loop, run in two different state spaces. Left: predicting and comparing whole images means every rollout pays for pixels no action can change. Right: predicting and comparing embeddings pays only for the handful of numbers a frozen, semantically-organized encoder decided were worth keeping. DINO-WM commits entirely to the right column.

How to read it: compare the boxed step in the middle of each column — "predict the next image" versus "predict the next latent" — that one substitution is the entire argument of this section, and the cost/failure notes underneath are its consequences.

Why plan by searching, instead of learning a policy? Think of a tilting ball-in-a-maze puzzle. Before you move it, you run it in your head: *if I tilt left, the ball rolls left, hits the wall, and stops near the entrance.* You imagine a few different tilts, pick the one whose imagined outcome looks best, and actually do only its *first* move — then look again and repeat.

That is exactly the planning method below:

Maze puzzle	World-model planning
your mental simulation	the learned world model
the tilts you consider	candidate action sequences
the exit you picture	the goal embedding
do only the first move, then look again	receding-horizon control

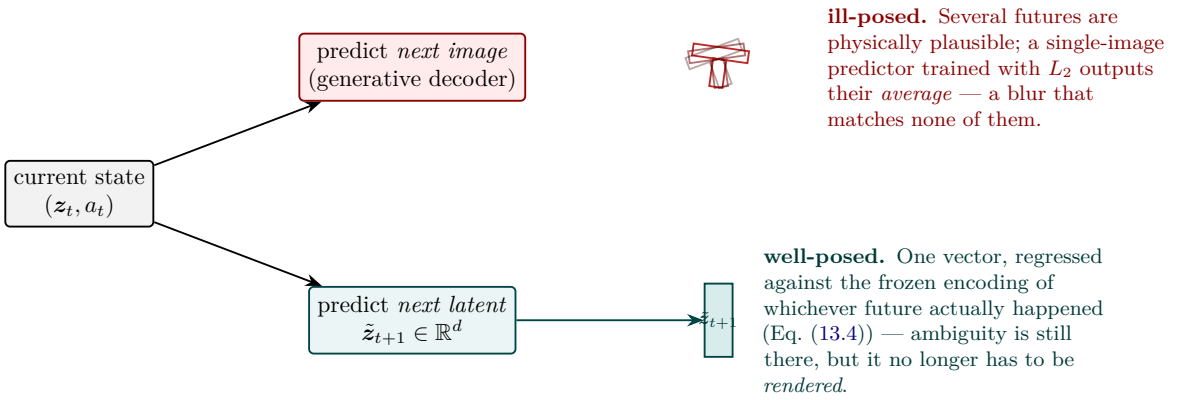


Figure 13.6. What a learned dynamics model actually predicts, and why it matters. Top: predicting the next raw image asks for a full $W \times H$ render of one specific future among several plausible ones; averaged under a squared-error loss, the plausible futures blur together (sketched here as three faint, offset silhouettes of the Push-T block). Bottom: predicting the next *latent* asks for one d-dimensional vector, regressed against a single, real target the frozen encoder already computed — the same underlying ambiguity about the future does not vanish, but it is no longer entangled with the separate, harder problem of rendering pixels.

How to read it: the red silhouettes are the same shape blurred by disagreement between plausible futures — that blur is what an image-space predictor is forced to output; the teal bar is a single number-vector with nothing to render, so it has no equivalent blur to produce.

The alternative, a *policy* network, would memorise "tilt left at this corner" for every corner. Search asks the model afresh each time instead. It costs more computation when acting, but you can change the goal — a different exit — without retraining anything.

13.3.1 Architecture: observe, encode, predict

At every control step the robot observes o_t, a $W \times H$ RGB image. The frozen DINOv2 encoder maps it to a grid of patch embeddings

$$\boldsymbol{z}_t = \mathrm{enc}(o_t) \in \mathbb{R}^{N \times d},$$

where $N = (W/14) \times (H/14)$ is the number of 14×14 patches (DINO-WM resizes images to 196×196, giving $14 \times 14 = 196$ patches) and d is the feature dimension of each patch token (384 in DINO-WM) (Zhou et al., 2024). No decoder, no pixels: $\boldsymbol{z}_t$ *is* the state.

Figure 13.6 makes the consequence of that one sentence concrete: a dynamics model that predicts the next *image* and a dynamics model that predicts the next *latent* are solving different-shaped problems, not the same problem in two encodings.

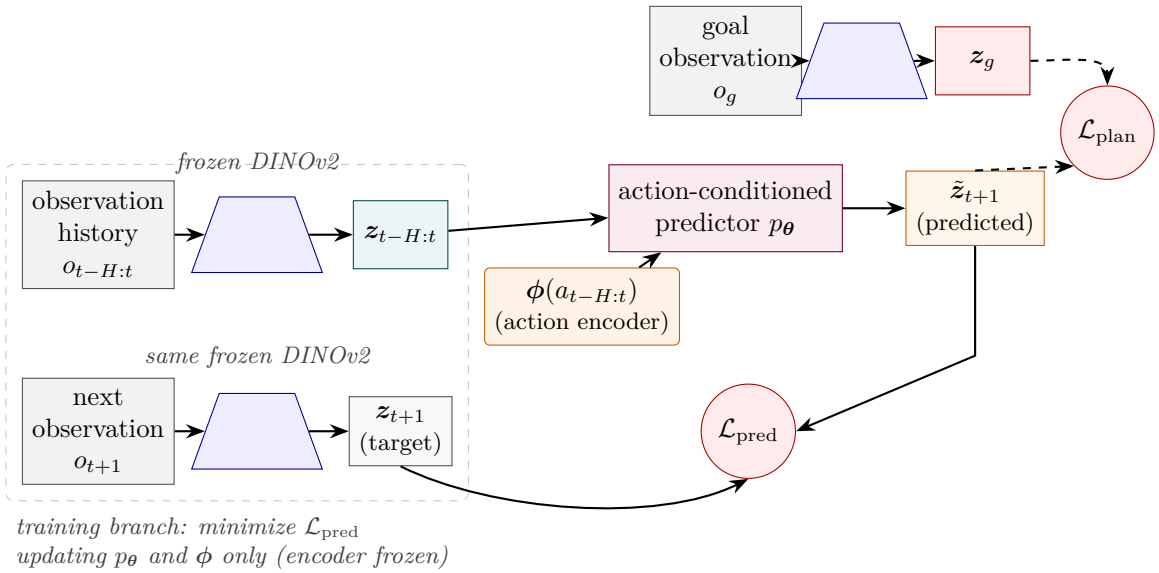


Figure 13.7. The DINO-WM block diagram (Lecture 08, slides 39–40). Left (dashed box): *training* — an observation history $o_{t-H:t}$ passes through the frozen DINOv2 encoder to the state history $\boldsymbol{z}_{t-H:t}$; the action-conditioned predictor $p_{\boldsymbol{\theta}}$, fed also with the embedded action history $\boldsymbol{\phi}(a_{t-H:t})$, predicts $\tilde{\boldsymbol{z}}_{t+1}$; the prediction loss $\mathcal{L}_{\text{pred}}$ of Eq. (13.4) pulls $\tilde{\boldsymbol{z}}_{t+1}$ toward the frozen encoding of the actually observed next frame o_{t+1}. Right: *planning* — a goal observation o_g is encoded to a fixed goal embedding $\boldsymbol{z}_g$; candidate action sequences roll the predictor forward and the planning loss $\mathcal{L}_{\text{plan}}$ of Eq. (13.6) measures how close the rollout ends to the goal.

The dynamics live on top of that state. A predictor network $p_{\boldsymbol{\theta}}$ receives a *history* of the last $H{+}1$ embeddings stacked with the corresponding embedded actions, and predicts the embedding one step ahead:

$$\tilde{\boldsymbol{z}}_{t+1} = p_{\boldsymbol{\theta}}(\boldsymbol{z}_{t-H:t},\, \boldsymbol{\phi}(a_{t-H:t})), \tag{13.3}$$

where H is the history horizon used for context (so $t-H:t$ spans $H{+}1$ time steps), $a_{t-H:t}$ is the action history, $\boldsymbol{\phi}$ is a small learned action encoder that lifts each low dimensional $a_t \in \mathbb{R}^2$ into the space the predictor operates in, and the tilde in $\tilde{\boldsymbol{z}}_{t+1}$ reminds us this is a *predicted* (rolled-out) embedding, not one computed from a real image. Figure 13.7 draws the full block diagram, including the goal embedding $\boldsymbol{z}_g$ that matters at planning time.

Training is a single regression loss in feature space:

$$\mathcal{L}_{\text{pred}} \;=\; \big\| \, p_{\boldsymbol{\theta}}(\text{enc}_{\boldsymbol{\theta}_{\text{frozen}}}(o_{t-H:t}),\, \boldsymbol{\phi}(a_{t-H:t})) \;-\; \text{enc}_{\boldsymbol{\theta}_{\text{frozen}}}(o_{t+1}) \big\|^2, \tag{13.4}$$

where:

- $\text{enc}_{\boldsymbol{\theta}_{\text{frozen}}}$ is the image encoder, *frozen* — in DINO-WM it is a pretrained DINOv2 that is never updated;
- $o_{t-H:t}$ are the last $H{+}1$ camera frames;

- the target $\text{enc}_{\boldsymbol{\theta}_{\text{frozen}}}(o_{t+1})$ is the encoding of the *next* real frame, treated as a fixed answer.

Only the predictor $\boldsymbol{\theta}$ and the action encoder $\boldsymbol{\phi}$ are trained. And every piece of supervision — the input frames and the target frame — comes from the same recorded video. Nothing was labelled; the future is the label.

Simplified intuition vs. more precise view

Simplified: "Train a network to predict the next DINOv2 feature given the current features and actions." **More precise:** the loss is a *deterministic* mean-squared error on patch tokens. Neither Eq. (13.4) nor the planning loss models uncertainty over outcomes, so the network learns a posterior *mean*: where dynamics are genuinely multi-modal (pushing the T along its exact symmetry axis could tip it left or right), the prediction averages over the modes and lands in $\mathcal{Z}$ where no real scene lands. Training one-step minimizes one-step error; all accumulation into multi-step error is the test-time issue of Section 13.5, not something Eq. (13.4) sees during training.

13.3.2 Training versus planning: two different jobs

DINO-WM has *two* losses, $\mathcal{L}_{\text{pred}}$ and $\mathcal{L}_{\text{plan}}$, and they are easy to mix up. They belong to two completely different phases, which happen at different times, change different things, and compare against different targets.

In plain words — learning physics versus using it

Think of learning to play billiards. First you spend hours just *watching what happens*: "if I hit the ball here, this hard, it rolls there". Nobody tells you which shot to play; you are only learning how the table behaves. That is **training**. Later, in a real game, you want to sink a particular ball. You do not learn anything new; you *use* what you know: you picture a few shots in your head, pick the one whose imagined result is best, and play it. That is **planning** (inference). DINO-WM does exactly these two things, in this order.

Phase 1 — training: learn "what happens next". The data are recordings of the robot moving around *with no task in mind*: videos of the pusher pushing the T-block in all sorts of ways, together with the action taken at each step. For Push-T the paper uses 18,500 such trajectories (Zhou et al., 2024). There is no goal and no reward in this data. From each recording we cut out small examples of the form

$$\text{(frames up to now, the action taken now)} \longrightarrow \text{the frame that really came next.}$$

The predictor sees the first part and must guess the features of the next frame. Its guess is compared with the real next frame's features:

$$\mathcal{L}_{\text{pred}} = \| \underbrace{p_{\boldsymbol{\theta}}(\boldsymbol{z}_{t-H:t},\, \boldsymbol{\phi}(a_{t-H:t}))}_{\text{predicted next features}} - \underbrace{\text{enc}(o_{t+1})}_{\text{real next features (from the video)}} \|^2.$$

This is Eq. (13.4) again. Gradient descent then changes the predictor's weights $\boldsymbol{\theta}$ (and the action encoder $\boldsymbol{\phi}$) so that the guess gets closer next time. The DINOv2 encoder is *not* changed. The inputs are always *real* recorded frames (this is called *teacher forcing*), and the target is always the *real* next frame: the video itself supplies the label, which is why this is self-supervised.

Worked Example — One training step, in the pretend 2-D feature space

Pretend each scene is described by just two feature numbers (the same toy is used for planning in Section 13.4.1). From a recorded video we take one example: the scene's features now are $\boldsymbol{z}_t = (0, 0)$, the recorded action was $a_t = (1, 1)$, and the features of the frame that *really* came next are $\boldsymbol{z}_{t+1} = (0.8, 1.1)$ (the block did not move quite as the push suggested).

- The predictor, early in training, guesses $\tilde{\boldsymbol{z}}_{t+1} = (1.0, 0.6)$.
- Loss: $\|(1.0, 0.6) - (0.8, 1.1)\|^2 = 0.2^2 + 0.5^2 = 0.04 + 0.25 = 0.29$.
- Gradient descent nudges $\boldsymbol{\theta}$ so that, for this situation and this action, the guess moves towards $(0.8, 1.1)$.

Repeat for millions of such examples. Afterwards, the predictor knows how the scene tends to change for any action — but it has *no idea what anyone wants*. No goal appeared anywhere.

Phase 2 — planning (inference): use the model to choose actions. Now the robot is given a task: a goal photo o_g. The trained predictor is *frozen* — no weight changes any more. What the planner changes instead are the *actions*. It proposes a sequence of moves, lets the predictor imagine the result step by step (each imagined step feeding the next, since no real future frames exist yet), and scores the imagined ending against the goal:

$$\mathcal{L}_{\text{plan}} = \| \underbrace{\tilde{\boldsymbol{z}}_{t+T}}_{\text{imagined ending, for these actions}} - \underbrace{\text{enc}(o_g)}_{\text{goal photo's features}} \|^2.$$

This is Eq. (13.6). The search (CEM, Section 13.4.1) tries many action sequences and keeps the one with the lowest $\mathcal{L}_{\text{plan}}$. Nothing is learned; the answer is a list of moves, of which the robot performs the first one.

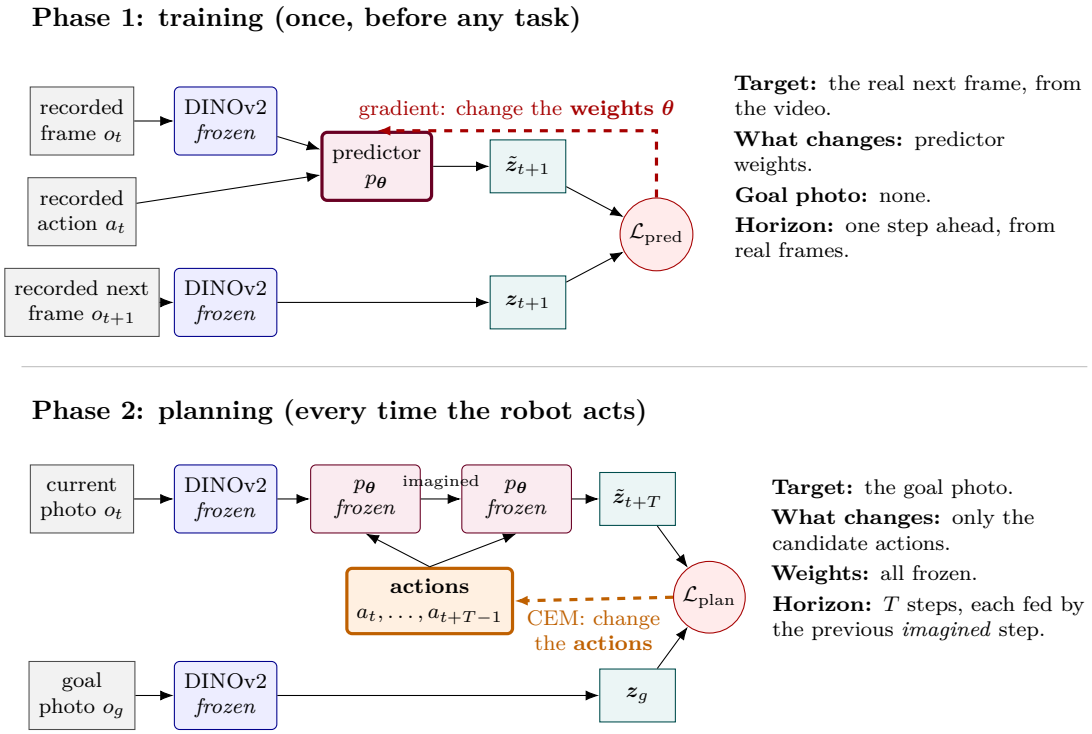


Figure 13.8. The same predictor in its two jobs. *Top, training:* real recorded frames and actions go in, the guess is compared with the real next frame, and the error changes the predictor's weights (red). *Bottom, planning:* everything learned is frozen; the only thing that changes is the list of actions (orange), chosen so that the imagined ending matches the goal photo.

How to read it: Compare the two dashed arrows. In training the error flows back into the network*; in planning it flows back into the* actions*. That is the whole difference.*

Table 13.1. Training and planning in DINO-WM, row by row.

	Training ($\mathcal{L}_{\text{pred}}$)	**Planning / inference** ($\mathcal{L}_{\text{plan}}$)
When	once, before deployment	every time the robot must act
Data	recorded trajectories: frames and actions, no task	the current photo and one goal photo
The question	"what will the scene look like next, after this action?"	"which actions will make the scene look like the goal?"
Target	features of the *real next frame*	features of the *goal photo*
What is changed	predictor weights $\boldsymbol{\theta}$, action encoder $\boldsymbol{\phi}$	the candidate actions only
What is frozen	DINOv2 encoder	DINOv2 encoder *and* the predictor
How	gradient descent on the weights	search over actions (CEM)
Look-ahead	one step, always from real frames	T steps, each from the previous imagined step
Output	a trained world model	a list of actions; the first is executed

Common Confusion

"Is the robot trained to reach the goal?" No. The goal never appears in training. Training only teaches "what happens next". That is exactly why DINO-WM can be given a *new* goal photo without any retraining ("zero-shot planning"): the goal enters only at planning time. **"Is $\mathcal{L}_{\text{plan}}$ used to update the network?"** No. It is only used to rank candidate actions. **"Training predicts one step, but planning imagines T steps — how?"** By chaining: the predicted features of step 1 are fed back in as the input for step 2, and so on. The model was never trained on its own imperfect guesses, so small errors can add up over the chain; this is why planning looks only a few steps ahead and replans after every move (Section 13.5).

13.3.3 Why plan inside frozen features?

Three reasons, all visible in Fig. 13.7:

1. **Pixels are a noisy target.** Predicting every RGB value makes the model spend capacity on texture, lighting and backgrounds it cannot control. Predicting patch tokens asks it to learn only what enc already judged to be stable and meaningful.
2. **The space already means something.** By the DIFT argument of Section 13.1.1, distances in DINOv2 space follow the structure of the scene. So regression in $\mathcal{Z}$ learns the dynamics that matter, not photometric detail.
3. **The encoder never moves.** Because enc is frozen, the predictor is a small network trained once, on *previously collected trajectories*, with no particular task in mind. The same world model then serves any goal you give it later — the paper's "task-agnostic reasoning" claim.

13.3.4 A minimal world model to hold in your hands: CartPole with residual dynamics

DINO-WM makes three design choices:

1. the state is a *latent* (an embedding), not pixels;
2. the dynamics predict a *change* in that latent, given the action;
3. behaviour comes from *planning at test time*.

It wraps them in a Transformer and a ViT-sized encoder. Strip away the scale and the same three choices fit in thirty lines of code — which is what the course's companion case study does: a world model for the classic CartPole task, trained in a *normalised latent space* and used to plan by trying every rollout. It is small enough to write out in full, so it is the chapter's running worked example. Every part of it has a counterpart in the DINO-WM pipeline of Section 13.3.1.

CartPole is the smallest environment in which "predict what happens if I do this" is a real question. The simulator's state has four continuous coordinates: cart position, cart

```python
class Dynamics(nn.Module):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(6, 48),   # 4 latent coordinates + 2 action dims
            nn.SiLU(),
            nn.Linear(48, 4),   # predict the residual Delta z
        )

    def forward(self, z, one_hot_action):
        predicted_change = self.network(
            torch.cat([z, one_hot_action], dim=1))
        return z + predicted_change   # residual: z_{t+1} = z_t + Delta z
```

Listing 13.1. The residual latent transition model of the CartPole case study (companion script, world-model section). Inputs are the concatenation of the 4-dimensional standardized state and a 2-entry one-hot encoding of the binary action; the network predicts only the *change* $\Delta \boldsymbol{z}$, and the forward pass adds it to the current state. The target for training is the encoding of the next recorded observation, so the supervision is free.

velocity, pole angle, and pole angular velocity. The agent's action is binary — push the cart left or push it right — and the task is to keep the pole upright. There is no image here; the "observation" is the 4-vector itself, so the frozen visual encoder of DINO-WM is replaced by a fixed standardization map E that rescales each coordinate to zero mean and unit variance. The standardized vector

$$\boldsymbol{z}_t = E(o_t) \in \mathbb{R}^4$$

is the latent state, and it plays exactly the role of the DINOv2 patch tokens $\boldsymbol{z}_t$ in Eq. (13.3): a fixed, semantically meaningful state space that the dynamics model is not allowed to change.

Two details of Listing 13.1 are worth a slow read.

1. **The one-hot action.** There are only two actions, so the action encoder $\boldsymbol{\phi}$ of Eq. (13.3) is just an indicator vector in $\{0, 1\}^2$. No learned embedding is needed when the actions have no notion of "closer" or "further" worth keeping.
2. **The residual output.** The network predicts a *change* $\Delta \boldsymbol{z}$, and the forward pass returns $\boldsymbol{z} + \Delta \boldsymbol{z}$. Over one time step a physical state barely moves, and an untrained network outputs values near zero — so the model starts out guessing "nothing much changes", which is already roughly right. Training only has to learn how the pole's velocity and angular velocity respond to the push.

This is the same reason residual connections help everywhere in this book: predicting a small change from a good baseline is easier than predicting the whole answer. It also protects the planner from drift: anything the actions cannot affect stays put by default.

Training is the CartPole cousin of Eq. (13.4). Roll the simulator with a sloppy or random policy and record transitions (o_t, a_t, o_{t+1}); standardize each observation once; then minimize the mean squared error

$$\mathcal{L}_{\text{toy}} \;=\; \big\| \underbrace{\boldsymbol{z}_t + \Delta \boldsymbol{z}_{\boldsymbol{\theta}}(\boldsymbol{z}_t, a_t)}_{\hat{z}_{t+1}} - E(o_{t+1}) \big\|^2, \tag{13.5}$$

where $\Delta \boldsymbol{z}_{\boldsymbol{\theta}}$ is the small MLP of Listing 13.1 and $E(o_{t+1})$ is the encoding of the *recorded next observation*. Read the target side of the loss carefully, because it carries this chapter's whole thesis in miniature: the target $E(o_{t+1})$ is not annotated by anyone. The simulator ran, time advanced, and the next observation simply appeared in the log — the case study phrases it as "time acts as the supervisor". Training is fully self-supervised *with respect to the transition targets*; the only ingredients with a supervision question mark are the action labels a_t, which were logged by the controller that drove the pole, and the goal specification, which we return to in Section 13.2.3. Every gradient step of Eq. (13.5) is a regression toward a state the world produced for free.

Planning, with every number written out. Once the world model is trained, controlling the cart is a search.

Step 1. List every plan. Each step is "push left" or "push right", and we look 10 steps ahead, so there are $2^{10} = 1{,}024$ possible plans. Few enough to try them all — no clever search (such as CEM) needed.

Step 2. Imagine each plan. Run each plan through the learned model for 10 steps. Each step feeds on the model's own previous prediction, not on reality — the `predicted_z = model(predicted_z, action)` line of the script.

Step 3. Score the imagined ending. Measure how far the final imagined state is from a target: pole upright, not swinging, cart near the centre. (This is the small-scale version of Eq. (13.6).)

Step 4. Act on the best plan — but only its first push.

Step 5. Look at what really happened, and plan again from there.

Cost per control step: $1{,}024 \times 10 = 10{,}240$ imagined steps, each one pass through a tiny network with 6 inputs. Replanning after every single action is what keeps small errors in the model from adding up.

Worked Example — CartPole: the planning arithmetic

How expensive is exhaustive latent planning here, and where does the cost go? Per control step the planner rolls 1,024 candidates through 10 transition steps: 10,240 calls to a two-layer MLP with 6 inputs — trivial for any hardware, which is the point of planning in a small latent space. Contrast the pixel-space alternative: a $W \times H$ image

rollout would pay WH predicted values per step plus an image distance per candidate, and Section 13.3.1's "pixels are noisy supervision" argument says most of those values are distractors. Now generalize: if the control step is 0.02 s of simulated time and the pole must be balanced for 500 steps, one episode costs $500 \times 10{,}240 \approx 5.1$ million latent transition evaluations — still cheap — while a pixel-level model would not fit the episode into real time at all. The same arithmetic reappears at DINO-WM scale in Exercise Q7: 100 candidates rolled 20 steps, replanned 50 times, is 10^5 predictor calls per episode, feasible only because a call moves no pixels.

The toy is DINO-WM in miniature — piece by piece.

CartPole toy	**DINO-WM**
fixed standardisation E	frozen enc. Neither is trained; both fix what every point of $\mathcal{Z}$ means before dynamics learning starts.
residual MLP + one-hot action	$p_{\boldsymbol{\theta}}$ with $\boldsymbol{\phi}$ (Eq. (13.3)). Same form — past latents in, next latent out, actions as extra input. The toy needs no history ($H = 0$); Push-T needs $H \geq 1$.
$\mathcal{L}_{\text{toy}}$	$\mathcal{L}_{\text{pred}}$ (Eq. (13.4)): squared error against the encoding of the next state actually observed.
hand-weighted distance to a chosen coordinate	$\mathcal{L}_{\text{plan}}$ (Eq. (13.6)) to $\boldsymbol{z}_g = \text{enc}(o_g)$, the embedding of a goal *image* — so the task, too, is specified without human knowledge of which coordinates matter.
try all $2^{10} = 1{,}024$ plans	random search / CEM (Figure 13.10) with K samples.

The last row is the only real difference. The toy can try every plan because each action is binary. Push-T's pusher moves continuously, so a horizon-T plan is a point in $\mathbb{R}^{2T}$ — infinitely many options. CEM exists to approximate, with K samples and elite refitting, what the toy does exactly with 1,024 evaluations. Same loss landscape, different search budget.

The toy also makes Section 13.5's warnings concrete.

- **Drift.** Ten imagined steps are few enough that the residual MLP barely drifts, which is why the case study can act on its plans with confidence. At 50 steps the compounding-error warning of Figure 13.16 applies: pole dynamics are smooth but unforgiving near the tipping point.
- **The goal is a design choice.** The perfectly balanced pose is not actually reachable, so the score is a weighted compromise (pole upright *and* cart centred) that someone had to design. DINO-WM's goal images inherit the same issue: Eq. (13.6) carries whatever blind spots and biases the target representation has.

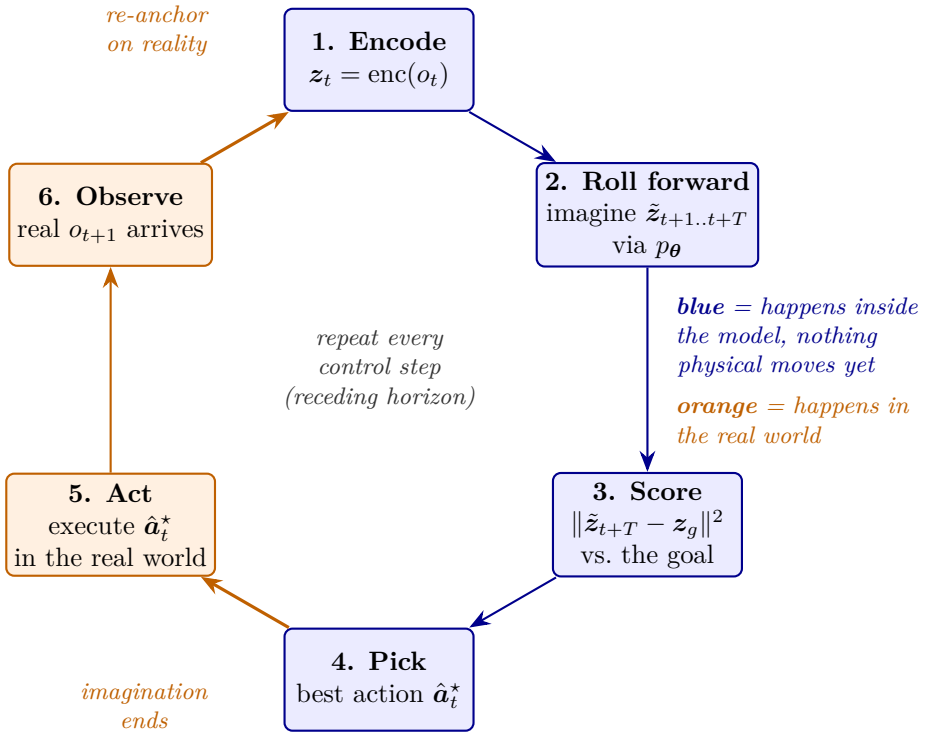


Figure 13.9. The world-model loop, in full: encode the real observation, roll the dynamics model forward to imagine several steps ahead, score the imagined outcome against a goal, pick the best action — then, and only then, leave the model and act in the real world, observe what actually happened, and re-encode. Steps 1–4 (blue) can run many times per control step at negligible cost (Figure 13.5); steps 5–6 (orange) happen exactly once, in reality, and are what the next cycle's Step 1 is anchored to.

How to read it: find the two orange arrows — "imagination ends" and "re-anchor on reality" — those are the only two moments in the whole loop where the model's beliefs are forced to meet the actual world; every other arrow is pure computation.

Key takeaway — DINO-WM in one line

Every design choice in this section removes one thing the dynamics model would otherwise have to get right for free: freezing the encoder removes the risk of the representation collapsing; predicting a latent instead of an image removes the need to render; a residual parameterization removes the need to learn "nothing changes" from scratch. What is left for p_θ to learn is only the part that is genuinely dynamics.

13.4 Planning as test-time optimization

Figure 13.9 is the whole chapter compressed into one cycle. Everything before this section built the pieces; this section is only about how they run, together, at test time.

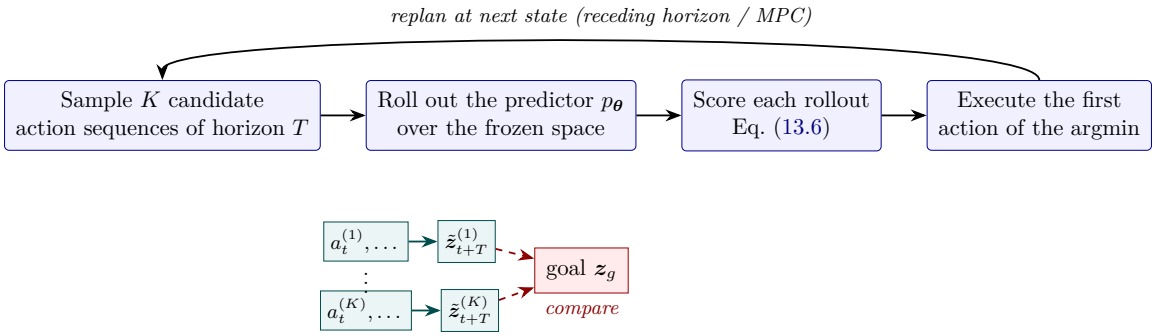


Figure 13.10. The test-time planning loop (Lecture 08, slides 40–41). Top: the receding horizon cycle — sample candidate action sequences, roll each through the predictor, score its terminal embedding against $\boldsymbol{z}_g$, execute the first action of the best sequence, then replan. Bottom: the "roll out" stage unpacked as K parallel imagined trajectories whose terminal points are compared to the goal embedding. Two concrete search rules fit the same box: *random shooting* (keep the single best sample per round) and the *cross-entropy method* (refit the sampling distribution to the elite samples each round).

With a trained $p_{\boldsymbol{\theta}}$, acting reduces to search. There is no policy network; instead, at each control step we imagine many possible futures and execute the first action of the best one — the receding-horizon scheme control theory calls *model-predictive control* (MPC). Concretely, the planner maintains a distribution over action *sequences* of horizon T and iterates:

1. **Sample** K candidate action sequences $(\hat{\boldsymbol{a}}_t^{(k)}, \hat{\boldsymbol{a}}_{t+1}^{(k)}, \dots, \hat{\boldsymbol{a}}_{t+T-1}^{(k)})$, $k = 1, \dots, K$, from the current search distribution (initially broad, e.g. Gaussian around zero motion).
2. **Roll out** each candidate through the frozen encoder's space: start from the true current embedding $\boldsymbol{z}_t$ and repeatedly apply Eq. (13.3), feeding every step its own previous, *predicted* embedding, to obtain the imagined end state $\tilde{\boldsymbol{z}}_{t+T}^{(k)}$.
3. **Score** each candidate by its terminal distance to the goal embedding:

$$\mathcal{L}_{\text{plan}}^{(k)} = \left\| \tilde{\boldsymbol{z}}_{t+T}^{(k)} - \boldsymbol{z}_g \right\|^2, \tag{13.6}$$

   where $\boldsymbol{z}_g = \text{enc}(o_g)$ is the frozen-DINOv2 encoding of a goal image o_g — a single photograph of the scene as it should look when the task is done.
4. **Refit and repeat**: refit the sampling distribution to the best-scoring candidates (*cross-entropy method*, CEM) or simply keep the best sample (*random shooting*); take the argmin and execute only its first action $\hat{\boldsymbol{a}}_t^\star = \arg\min \mathcal{L}_{\text{plan}}^{(k)}$.
5. Observe the real o_{t+1}, re-encode, and replan — closing the loop against the drift of Section 13.5.

Two features of Figure 13.10 deserve emphasis. First, the search is entirely *in feature space*: K imagined trajectories are evaluated with one matrix distance each, no rendering and no physics engine. One predictor step takes about 0.014 s for a batch of 32 on a single GPU, against about 3 s for one step of the simulator; a full CEM plan of 100 samples × 10 rounds takes under a minute (Zhou et al., 2024). Second, the goal is *an image*, not code: specifying a

new task is pointing a camera at the desired scene once. The same world model that was trained without task labels can now, at test time, be aimed at arbitrary goals — this is what the paper means by *zero-shot planning* (Zhou et al., 2024).

What is optimised at test time? No network weights. Once the goal is given, the *only* free numbers in the whole system are the candidate action sequences. That is exactly what "task-agnostic" means for DINO-WM (slides 40–41): the trained parts (enc, $p_{\boldsymbol{\theta}}$, $\boldsymbol{\phi}$) know nothing about the goal. Changing the goal o_g mid-deployment costs one forward pass of the frozen encoder, and the slide's note "actions optimized at test-time" is literally true.

Two extremes of the same loop:

- **CartPole** (Section 13.3.4): two actions, horizon 10. It simply lists all $2^{10} = 1{,}024$ sequences, scores them, and keeps the best — random shooting where K is the whole search space.
- **DINO-WM on Push-T**: continuous pusher motion, so listing everything is impossible. CEM refits $(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ so each new batch of K samples lands where the last batch's elites scored well.

The planning loss Eq. (13.6) and the "do the first action, then replan" rhythm are the same in both. Only the way candidates are proposed changes with the shape of the action space.

13.4.1 How the best action is chosen, worked out by hand

The steps above are compact, so this subsection slows down and does them by hand, with pictures. There are only three ideas: *what "best" means*, *how candidates are tried*, and *why only the first step is carried out*.

In plain words — planning is imagining, then scoring

> DINO-WM never learns "in this situation, push left". Instead, every time it has to act, it *imagines*. It invents many possible sequences of pushes, lets the world model predict what the scene would look like after each sequence, and compares each imagined final scene with the goal photo. The sequence whose imagined ending looks most like the goal wins. The robot performs only its first push, takes a new photo, and imagines again from there. Think of a mini-golf putt: you picture a few shots, choose the one whose imagined ball ends nearest the hole, hit — then look where the ball really went before the next shot.

Idea 1: "best" means "the imagined ending looks most like the goal photo". The planner cannot measure "how far is the T from its target pose" directly: it has no idea where the T is. All it has are DINOv2 features. So it measures distance *between feature grids*: the goal photo is encoded once, $\boldsymbol{z}_g = \text{enc}(o_g)$, and an imagined ending $\tilde{\boldsymbol{z}}_{t+T}$ is scored by $\|\tilde{\boldsymbol{z}}_{t+T} - \boldsymbol{z}_g\|^2$, Eq. (13.6). Because DINOv2 places similar-looking scenes close together, this

number behaves like "how different would the two pictures look?": large when the T is far away or turned the wrong way, zero when the scene matches the goal (Figure 13.11). That is the *only* notion of success the planner has; there is no reward and no hand-written "T-position" anywhere.

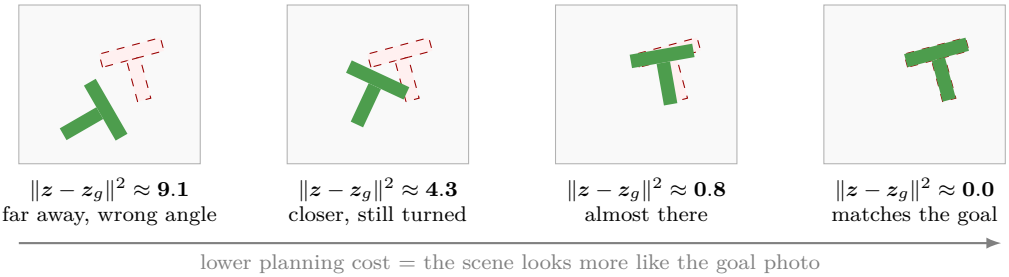


Figure 13.11. What the planning cost measures (illustrative numbers). Each square is one Push-T scene; the red dashed T is the goal pose, the green T is where the block is. The planner compares the DINOv2 features of a scene with those of the goal photo. The further the block is from the goal pose, in position or angle, the larger the distance — so "minimise the distance" means "make the scene look like the goal".

How to read it: Read left to right: the only thing the planner ever sees is the number under each square.

Idea 2: try candidates in imagination and keep the best. To see the mechanics without 196 patch vectors, shrink everything to a toy. Pretend the feature space is 2-D, so a state $\boldsymbol{z}$ is a point on a page, and pretend the learned predictor is very simple: an action moves the point by that action, $\tilde{\boldsymbol{z}}_{t+1} = \tilde{\boldsymbol{z}}_t + a_t$. (In DINO-WM the predictor is a learned transformer, but the planner uses it exactly the same way: as a black box that answers "what comes next?".)

Worked Example — Picking the best of five candidate plans

Current state $\boldsymbol{z}_0 = (0, 0)$, goal $\boldsymbol{z}_g = (5, 3)$, horizon $T = 2$, and each action may move at most one unit per coordinate. The planner samples five two-step candidates, imagines where each ends, and scores it:

candidate	actions (a_0, a_1)	imagined end $\tilde{\boldsymbol{z}}_2$	cost $\|\tilde{\boldsymbol{z}}_2 - \boldsymbol{z}_g\|^2$	
A	$(1,0),\ (1,0)$	$(2,0)$	$3^2 + 3^2 = 18$	
B	$(1,1),\ (1,1)$	$(2,2)$	$3^2 + 1^2 = \mathbf{10}$	best
C	$(0,1),\ (0,1)$	$(0,2)$	$5^2 + 1^2 = 26$	
D	$(1,1),\ (1,0)$	$(2,1)$	$3^2 + 2^2 = 13$	
E	$(-1,0),\ (1,1)$	$(0,1)$	$5^2 + 2^2 = 29$	

B has the lowest cost, so B is the plan. Note three things. (1) Nothing moved in the real world yet: all five rows were computed inside the model. (2) The goal is not reachable in two steps; the planner simply prefers the ending *closest* to it. (3) Only B's *first* action, $a_0 = (1, 1)$, is carried out (Idea 3). Figure 13.12 draws this table.

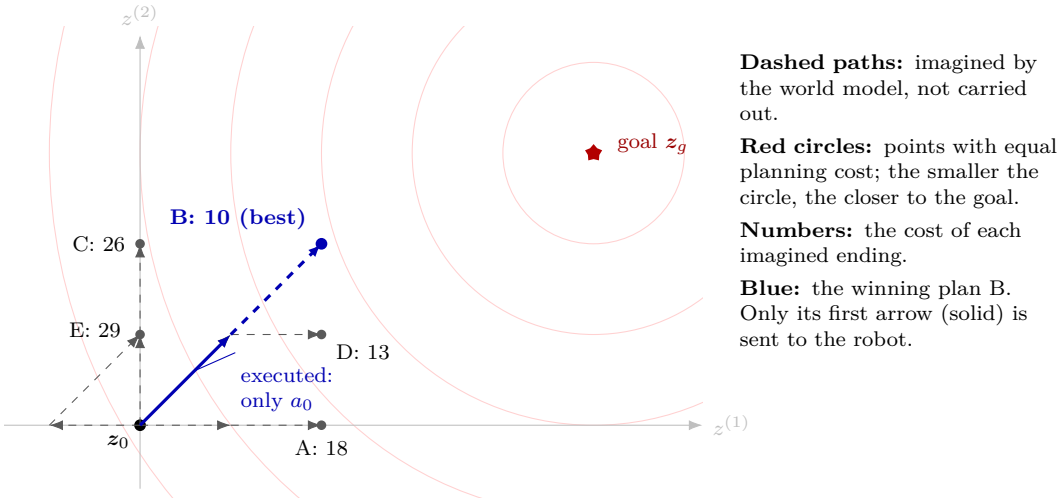


Figure 13.12. The worked example drawn in a pretend 2-D feature space. Each candidate plan is rolled out in imagination (dashed), its end point is scored by its squared distance to the goal embedding, and the plan with the lowest score wins. In DINO-WM the same happens with 196 patch vectors instead of one 2-D point, and with a learned predictor instead of "add the action".

How to read it: Find the end point that sits on the smallest red circle; that plan wins, and only its first step is executed.

How the candidates are proposed: the cross-entropy method. Five random guesses were enough in the toy. For Push-T they are not: a plan is T two-dimensional pusher moves, and even a coarse grid of 10 values per number with $T = 5$ gives 10^{10} possible plans. DINO-WM therefore uses the **cross-entropy method** (CEM), a "guess, keep the best, guess again nearby" search (Zhou et al., 2024):

1. Keep a Gaussian "cloud" over plans (a mean and a spread).
2. Draw N plans from the cloud and score each by imagination (as in the table above).
3. Keep the K best, the *elites*.
4. Move the cloud: new mean = the elites' mean, new spread = the elites' spread.
5. Repeat a fixed number of rounds. The cloud shrinks onto a good plan.

The paper times planning with $N = 100$ samples per round and 10 rounds, i.e. 1,000 imagined rollouts per plan (Zhou et al., 2024).

Worked Example — CEM on one number, three rounds

To see the cloud move, plan a single action a (one number). Suppose the world model, together with the goal, gives the cost $c(a) = (a - 2.6)^2$ — the planner does not know this formula; it can only evaluate it for the actions it tries. Use $N = 8$ samples and $K = 3$ elites.

Round 1. Cloud: mean 0, spread 2. Samples: -2.9, -1.5, -0.4, 0.3, 1.1, 1.8, 2.4, 3.3. Lowest costs: 2.4 (0.04), 3.3 (0.49), 1.8 (0.64). New mean $= (2.4 + 3.3 + 1.8)/3 = 2.5$; new spread $= 0.62$ (the standard deviation of the three elites).
Round 2. Cloud: mean 2.5, spread 0.62. Samples: 1.7, 2.1, 2.3, 2.5, 2.6, 2.8, 3.0, 3.4. Elites: 2.6 (0.00), 2.5 (0.01), 2.8 (0.04). New mean $= 2.63$; new spread $= 0.12$.
Round 3. The cloud is now a narrow bump at 2.63 — within 0.03 of the true best action 2.6, found with only 24 evaluations and without ever seeing the cost formula (Figure 13.13). In DINO-WM the "one number" is a whole plan of $2T$ numbers and each evaluation is an imagined rollout, but the rounds look exactly like this.

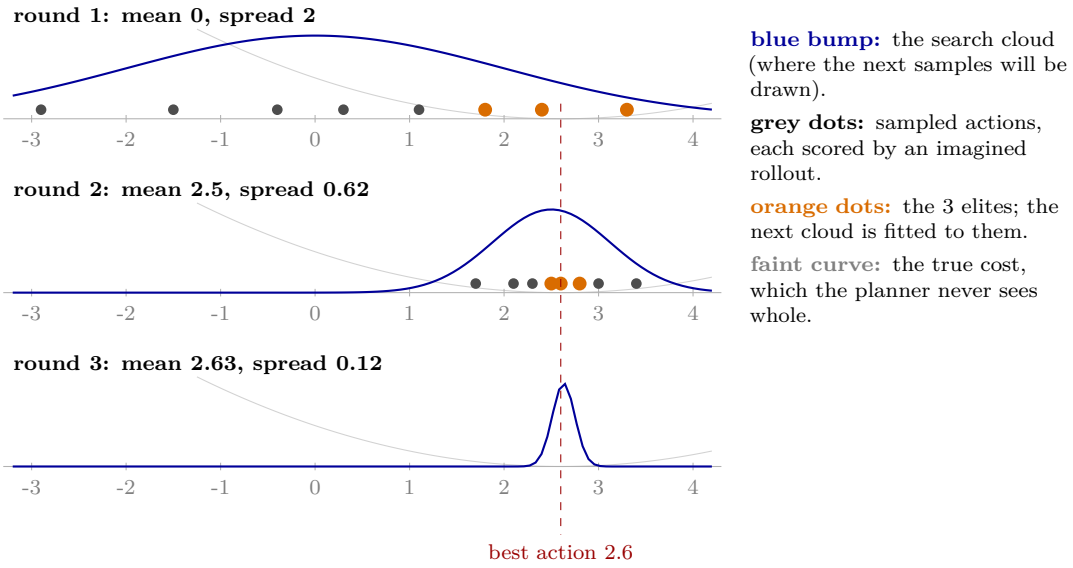
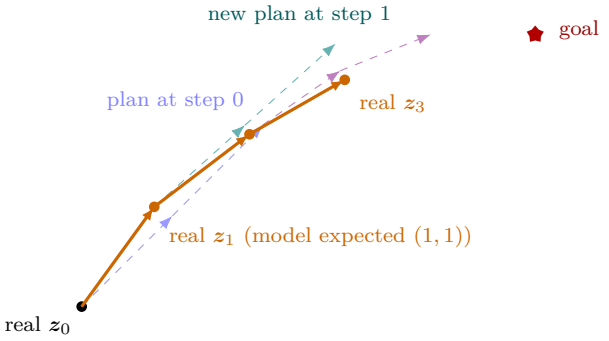


Figure 13.13. The cross-entropy method on the one-number example. Each round draws samples from the blue cloud, scores them by imagination, and refits the cloud to the best three (orange). In three rounds the cloud moves from "anything between -4 and 4" to a narrow bump on the best action.

How to read it: Follow the blue bump downwards: it slides right and narrows, because each new cloud is fitted only to the orange dots of the round above.

Idea 3: do only the first step, then look again. Why not carry out the whole winning plan? Because it was computed from *imagined* states, and every imagined step inherits the model's small errors, which add up the further it imagines (Section 13.5). After one real step the robot can take a fresh photo, which is *exact*, and plan again from there (Figure 13.14). This is **model-predictive control** (MPC), also called a receding horizon: the plan always looks T steps ahead, but that window slides forward with every step taken.

The paper measures what replanning is worth on Push-T: planning once with CEM and executing the whole plan succeeds in 86% of test episodes, while replanning in the MPC loop succeeds in 90% (and on the Wall task the gap is 74% versus 96%). Optimising the actions



Dashed: each step's imagined plan ($T = 2$).

Orange: what really happened — one step per plan.

Each real step lands a little off the imagined one. Instead of following an outdated plan, the robot re-encodes a fresh photo and plans again from where it *actually* is, so the small errors never pile up.

Figure 13.14. Receding-horizon planning (MPC) in the toy space. Each step: plan T steps ahead in imagination, carry out only the first, observe where the robot really ended up, and plan again from there.

directly by gradient descent through the world model, instead of CEM, reaches only 28% on Push-T (Zhou et al., 2024, Table 8). (Strictly, the paper executes the first few actions of each plan before replanning; "the first one" is the limiting case.)

Step by step — choosing one action in DINO-WM

Step 1. Photograph the scene now and the goal once; encode both with the frozen DINOv2: $\boldsymbol{z}_t$ and $\boldsymbol{z}_g$.

Step 2. Draw N candidate plans (sequences of T pusher moves) from the current search cloud.

Step 3. For each plan, imagine: feed $\boldsymbol{z}_t$ and the moves through the predictor $p_{\boldsymbol{\theta}}$, step by step, to get the imagined ending $\tilde{\boldsymbol{z}}_{t+T}$.

Step 4. Score each plan: $\|\tilde{\boldsymbol{z}}_{t+T} - \boldsymbol{z}_g\|^2$ — "how unlike the goal photo would the scene look?".

Step 5. Keep the K best plans; refit the cloud to them; repeat steps 2–5 for a few rounds.

Step 6. Carry out the first move(s) of the final best plan on the real robot. Take a new photo. Go back to step 1.

Nothing in this recipe is trained: the encoder and predictor are fixed; the only things being optimised are the candidate moves.

Worked Example — One round of CEM planning in Push-T

Take a history horizon $H = 1$, planning horizon T, and $K = 100$ candidate sequences. The current state is $\boldsymbol{z}_t$, the DINOv2 embedding of the current arena frame. (i) *Initialize* a Gaussian search distribution over the $2T$ action coordinates with mean $\boldsymbol{\mu}^{(0)} = \mathbf{0}$ (no pusher motion) and diagonal covariance $\boldsymbol{\Sigma}^{(0)}$ wide enough to cover the arena. (ii) *Sample* the 100 sequences $\{\hat{\boldsymbol{a}}^{(k)}_{t:t+T-1}\}$, roll each through $p_{\boldsymbol{\theta}}$, and compute $\mathcal{L}^{(k)}_{\text{plan}}$ from Eq. (13.6).

Figure 13.15. A DINO-WM rollout actually converging on a goal, reproduced from the paper's demonstration (Zhou et al., 2024) (Lecture 08, slide 41). Each panel is one point in time; the solid T is the block's current pose, the translucent T is the goal pose o_g the plan is steering toward. Four frames into the imagined-then-executed plan, the block has visibly closed most of the gap — exactly the behaviour Figure 13.10's sample/score/pick/replan cycle is built to produce.

Source: Lecture 08, slide 41; figure from Zhou et al. (2024).

How to read it: track the solid T across the four panels against the fixed, translucent goal T — this is what "the planning loss went down" looks like in the actual environment, not just in $\mathcal{Z}$.

(iii) *Select elites*: keep the $E = 10$ sequences with smallest loss. (iv) *Refit*: set $\boldsymbol{\mu}^{(1)}$ and $\boldsymbol{\Sigma}^{(1)}$ to the mean and covariance of those elites; most of the mass now lies on sequences that slide the pusher toward the T's long edge. Repeat steps (ii)–(iv) for a handful of iterations — the sampling distribution collapses onto a plan — then execute only $\hat{\boldsymbol{a}}_t^{(1)}$ of the final mean, observe o_{t+1}, recompute $\boldsymbol{z}_{t+1}$ with the frozen encoder, and start over. Because the embeddings of the rollout are predicted, not observed, never trust the imagined trajectory beyond the horizon: the loop's replanning is what keeps *compounding error* (next section) bounded.

Paper Analysis: Zhou et al. (2024) — DINO-WM

Paper: "DINO-WM: World Models on Pre-trained Visual Features enable Zero-shot Planning" (arXiv, Feb. 2025; Zhou, Pan, LeCun, Pinto). **Problem:** world models should be (i) trainable on offline, pre-collected trajectories, (ii) usable for test-time behavior optimization, and (iii) task-agnostic — no per-task retraining. **Insight:** the representation space of a frozen, self-supervised encoder (DINOv2 patch tokens) is a sufficient state space for dynamics learning; never decode to pixels. **Method:** freeze DINOv2; train a small action-conditioned predictor $p_{\boldsymbol{\theta}}$ on histories $(\boldsymbol{z}_{t-H:t}, a_{t-H:t})$ with the L_2 prediction loss of Eq. (13.4); plan with CEM-style test-time optimization of action sequences against Eq. (13.6). **Experiments:** the paper evaluates this on Push-T and related environments with offline data and reports successful zero-shot goal-directed planning from raw goal images; consult the paper for exact success rates, which the lecture does not reproduce numerically. **Limitations:** inherits every property of the frozen features (domain shift of Section 13.5); deterministic posterior mean; no explicit uncertainty, horizon drift, or contact discontinuities modeled. **What to remember:** this chapter's thesis made concrete — a decade of self-supervised representation learning left behind spaces in which dynamics models can be small, task-agnostic, and planned-in directly.

Research Frontier

DINO-WM is the frozen-feature instance of a broader idea usually labeled *latent world models* or *JEPA-style prediction*: predict the future *in a learned representation space*, never in pixels, and let a representation metric — rather than photometric fidelity — define "correct prediction." The family splits along the choice of that representation. *Learned jointly:* the encoder and dynamics are trained together, risking a collapse in which trivial representations satisfy the prediction loss; several chapters of Part III connect this to the collapse-avoidance machinery of Chapters 10–11 (stop-gradient, variance regularization, masking). *Frozen, as here:* collapse is impossible because the encoder cannot move, at the price of inheriting whatever blindness the frozen space has. DINO-WM (Zhou et al., 2024) is the concrete, reproducible instance of the frozen branch relevant to this book; its authors' framing explicitly descends from the JEPA position, but the lecture presents only DINO-WM, and we follow the lecture. A frontier question both branches share: *does the representation space need to encode what can happen, not only what is observed*? A frozen DINOv2 has never seen "this T-block is about to rotate"; its features carry part semantics, not affordances. Whether latent dynamics can be learned well in a space that was never optimized for dynamics is the empirical bet of the frozen-feature program.

Key takeaway — Planning in one line

Once you have a dynamics model, acting does not require learning a second network. It requires imagining several futures cheaply, scoring them against a goal you can also

encode, and committing to only the first step of the best one — because the further you imagine, the less you should trust it (next section).

13.5 Failure modes

Failure Modes

1. Compounding rollout error. During training, Eq. (13.4) always sees *real* inputs $\boldsymbol{z}_{t-H:t}$; at test time, step 2 of the loop feeds the predictor its own outputs. Every step adds a small prediction error, and the drift is roughly *superadditive* in the horizon because later errors compound on earlier ones (Figure 13.16, bottom). Practical consequences: short horizons T, replanning every step, and never executing imagined actions beyond the first. **2. Goal-embedding ambiguity.** The planning loss only sees $\boldsymbol{z}_g = \text{enc}(o_g)$. DINOv2 features are invariant to many things (exact viewpoint, minor lighting), so *many physically different scenes share a similar* $\boldsymbol{z}_g$; the planner can report success in feature space while the real block sits in the wrong pose. A goal image is a coarse specification. **3. DINO feature collapse under distribution shift.** Training DINO-WM on table-top RGB and deploying on a different renderer, different lighting, or a real camera breaks the premise that enc maps semantically similar scenes to similar $\boldsymbol{z}$'s: the frozen embedding of an out-of-distribution image can land arbitrarily, and the learned $p_{\boldsymbol{\theta}}$ — trained on in-distribution embeddings — has no valid dynamics to predict from there. This is the classic frozen-encoder bet going wrong. **4. The frozen-feature benefit is domain-specific.** Section 13.3.1's argument that "frozen DINOv2 already knows what matters" is a statement about *natural-image-like inputs*, where DINOv2's pretraining distribution (142M curated web images, Chapter 11) covers the deployment. The further the scene strays from that distribution — medical imaging, microscopy, industrial sensors, infrared — the weaker the guarantee, and the smaller the benefit of freezing.

The drift of Figure 13.16 is not merely a failure to be lamented; it is the design force behind half the choices in this chapter. Short horizons, receding-horizon replanning, planning with K short parallel rollouts rather than a single long open loop, and the refusal to fine-tune enc (lest its fixed geometry warp under the new loss) are all mitigations of one phenomenon: *imagination decouples from observation at a rate nobody controls.*

The failure that does *not* happen: collapse. After Chapters 10 and 11 you might expect it. Every predictive method without negatives needed a defence — stop-gradients and EMA teachers (BYOL, SimSiam), centring and sharpening (DINO). Here the defence is simpler.

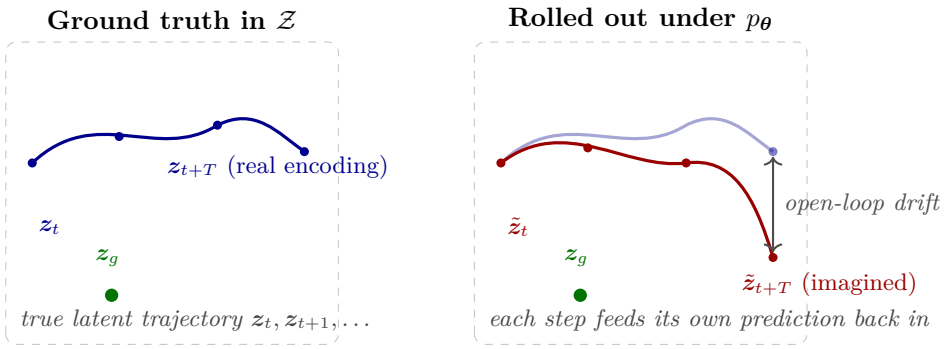


Figure 13.16. Feature-space drift across long horizons (schematic, not measured data). Left: a ground-truth latent trajectory (blue) — the curve through $\mathcal{Z}$ traced by real encoded frames — heads toward a goal embedding $\boldsymbol{z}_g$ (green). Right: an open-loop rollout under $p_{\boldsymbol{\theta}}$ (red) starts from the same $\boldsymbol{z}_t$ but each step consumes its own prediction; small per-step errors accumulate into a visible divergence by step $t{+}T$. This is why Section 13.4's loop executes only the first action and replans from the freshly encoded $\boldsymbol{z}_{t+1}$: replanning re-anchors every step on a real encoding, resetting the drift before it compounds.

Suppose $p_{\boldsymbol{\theta}}$ took the lazy route and always output the same $\tilde{\boldsymbol{z}} = \boldsymbol{c}$, whatever the history and action. Eq. (13.4) would charge it $\|\boldsymbol{c} - \text{enc}(o_{t+1})\|^2$ on every step. The frozen encoder keeps giving *different* targets for different next scenes, so that loss stays large.

Collapse needs *targets that can collapse*, and frozen DINOv2 targets cannot move. Freezing the encoder gets, in one line, what the non-contrastive methods needed extra machinery for. The systems chapter (Chapter 14) makes the same point for proprioceptive targets: a target whose meaning is fixed from outside cannot be bent to make the world easy to predict.

Learning note from study sessions

The two anecdotes in this section — the driver who predicts traffic lights rather than leaves, and the maze-tilter who runs imaginary rolls before touching the puzzle — come from the study session on the lecture's closing block; the study loop's own exam notes single out this last topic as its highest-value question. The same session's counterpart diagnosis is worth keeping: if an exam answer says pixel-based world models "look bad," it has missed the point — the argument is about where the *capacity and the loss* go, not about aesthetics.

Key takeaway — Failure modes, in one line

Nearly every mitigation in this chapter — short horizons, replanning every step, refusing to fine-tune the encoder — defends against the same underlying fact: a model trained on real inputs is asked, at test time, to consume its own predictions, and nothing forces those predictions to stay on the manifold the model was trained on.

13.6 The systems connection: who records the experience?

Where the ingredients come from. Step back and look at the sources:

- the training data for $p_{\boldsymbol{\theta}}$: trajectories the robot itself carried out;
- the actions in them: the robot's own motor commands (proprioception);
- the goal o_g: a picture from the robot's own camera.

Everything is data the *system produced for itself* — exactly Chapter 14's definition. We only preview the link here. The robots of Chapter 14 (a mapping rover, a legged hiker, a quadruped with audio, an autonomous RL arm) apply the same idea, with free supervision from paired sensors, physical laws, or the system's own uncertainty.

World models take that instinct one level up. Instead of using the robot's sensors as *labels for perception*, we use its experience as *labels for dynamics*. And the "extract the action from raw data" branch of Figure 13.3 is a system-level trick of the same kind: recover pseudo-actions from the observations, so the action-conditioned model also counts as SSL.

Key takeaway — The bridge to Chapter 14

> A world model is not a new source of free supervision — it is a new *use* for the free supervision a deployed system already has lying around in its own logs. The next chapter asks the more basic question this one took for granted: where do those logs, and their labels, actually come from?

13.7 Chapter summary

This chapter asked what a self-supervised representation is *for*, and answered: it can become the state space of a simulator. Section 13.2 first drew the boundary the rest of the chapter respects — a world model's *dynamics* objective is self-supervised almost by construction (the next frame labels itself), but whether the whole *system* counts depends on where the actions came from and, separately, where the goal came from. Get that three-way split right and every later claim in the chapter is easy to audit.

DINO-WM (Section 13.3) then made the recipe concrete with two commitments:

1. **Never decode to pixels.** Figures 13.5 and 13.6 show this is about cost *and* about asking a well-posed question — not mere convenience.
2. **Never learn a policy.** Search replaces it (Section 13.4): imagine, score, pick, act, observe, repeat (Figure 13.9).

Freezing the encoder has a price (Section 13.5): anything it never learned to represent is a blind spot, and any imagined rollout longer than a few steps drifts from reality. Almost every engineering choice in the chapter — short horizons, replanning after every step, refusing to fine-tune enc — responds to that one fact.

Table 13.2 collects the chapter's vocabulary in the order it appeared.

Table 13.2. The chapter's working vocabulary, in the order the ideas appeared.

Term	What it means, in plain words
Representation space $\mathcal{Z}$	The output space of an encoder, treated as a state space — useful only if distance in it means something about the scene.
World model	A learned function from (state, action) to a prediction of the next state, trained from logged experience rather than programmed physics.
Action-conditioned prediction	Predicting the next observation given not just the past but also *what the agent is about to do*.
Pseudo-action	An action inferred from raw observation pairs (e.g. by an inverse-dynamics model) when no logged action channel exists, letting action-conditioned prediction train on unlabeled video.
Frozen encoder	A pretrained network whose weights never change during world-model training — its output space is fixed before dynamics learning starts, which is what makes collapse impossible.
Predictor $p_{\boldsymbol{\theta}}$	The small trainable network that maps a latent history and an action (history) to a predicted next latent.
Goal embedding $\boldsymbol{z}_g$	The frozen encoding of a goal image — the whole task specification, expressed in the same units as everything else in the pipeline.
Model-predictive control (MPC)	Plan a sequence of actions, execute only the first one, then replan from the newly observed real state — a receding horizon.
Cross-entropy method (CEM)	A search strategy: sample candidates, keep the best-scoring *elites*, refit the sampling distribution to them, repeat.
Compounding rollout error	The growth of prediction error when a model's own outputs are fed back in as its next input, step after step, with nothing real to correct it.

Key takeaway — Before moving on

Keep the loop of Figure 13.9 in mind for the rest of the book: encode, imagine, score, pick, act, observe. Every system-level case study in Chapter 14 is a different answer to one question this loop leaves open — *where, exactly, do the encoder, the predictor, and the goal come from, if no human is supplying any of them?*

13.8 Check Your Understanding

Check Your Understanding

Level 1 — recall.

1. Define a world model in one sentence, naming the three objects (state, action, prediction) it relates.
2. In DINO-WM, which components are trainable and which are frozen? Why is the frozen part never updated?

3. What object plays the role of the task specification at test time?

Level 2 — comprehension.

4. Why does the lecture call autoregressive next-frame prediction "self-supervised" but reserve judgment about action-conditioned prediction? State the condition under which the latter counts as SSL.
5. Explain in words (no equations) why a deterministic L_2 predictor cannot represent genuinely stochastic outcomes, and what Eq. (13.4) learns in that case.
6. In Figure 13.3, place Diffusion Policy (Chi et al., 2023) on the correct branch and justify the placement.

Level 3 — application.

7. Count: with a planning horizon $T = 20$ and $K = 100$ candidates, how many predictor forward passes does one full evaluation of Eq. (13.6) require (assume a fixed history $H = 1$)? How does this change if you replan at every control step over 50 steps of real execution?
8. Design a pseudo-action extractor for a fixed over-head camera watching a table: what would you train, what are its inputs and outputs, and which branch of the decision tree of Figure 13.3 does it enable?

Level 4 — analysis.

9. A colleague proposes training enc *end-to-end* with $p_{\boldsymbol{\theta}}$, jointly optimizing Eq. (13.4). Give two distinct collapse mechanisms this joint training admits, and one defense from the non-contrastive-learning literature (Chapter 10) that would prevent each.
10. The planning loss (13.6) uses only the *final* embedding. Propose a denser variant that also penalizes $\|\tilde{\boldsymbol{z}}_{t'} - \boldsymbol{z}_g\|$ at intermediate steps t'. Does your denser loss bias the search toward prematurely matching the goal frame? Discuss.
11. Analysis of Figure 13.16: under what condition on the per-step error and the curvature of the true trajectory would you expect drift to be *benign*, and under what condition catastrophic? Argue geometrically.
12. In the CartPole case study of Section 13.3.4, planning enumerates all 1,024 ten-step sequences. Why is exhaustive enumeration valid there but impossible for Push-T, and which ingredient of Figure 13.10 restores tractability in the continuous-action case? What assumption does the enumerative planner make that CEM only approximates?

Level 5 — synthesis.

13. Connect this chapter to Chapter 7: DIFT (Tang et al., 2023) shows emergent *correspondence* in diffusion features, while DINO-WM needs *dynamics* in DINOv2 features. Are these the same property of the representation? If not, which is harder, and what experiment would tell them apart?
14. Position the frozen-encoder bet of DINO-WM on the core trade-off of Part III (structure you inherit from pretraining vs. structure you own by training end-to-

end): in which deployment regime is "frozen" the rational minimax choice, and in which is it guaranteed to underperform?

Solution sketches.

- *Q1.* A learned dynamics model mapping (current state, action or action history) to a prediction of the next state(s), so that futures can be simulated without running the real environment.
- *Q2.* Trainable: predictor $p_{\boldsymbol{\theta}}$ and action encoder $\boldsymbol{\phi}$. Frozen: DINOv2 enc. Freezing keeps the meaning of $\mathcal{Z}$ fixed so the meaning of a point — semantic content of the corresponding scene — cannot drift as training continues.
- *Q3.* A goal *image* o_g, encoded once into the frozen space as $\boldsymbol{z}_g$; Eq. (13.6) turns the image into a scalar objective.
- *Q4.* Unconditioned prediction supervision is free: the next recorded frame is the answer. Action conditioning becomes SSL only if the action channel is *free* too — logged by the system itself or recoverable by an inverse transformation; human-recorded actions are supervision.
- *Q5.* The posterior mean: when outcomes split multimodally the L_2-optimal prediction averages them, landing in feature space where no real state lies. Hence the blurry-seeming rollouts of long horizons.
- *Q6.* The "human-recorded / teleoperated" branch: demonstrations supply $(\boldsymbol{x}, \boldsymbol{a})$ pairs and the policy regresses $\boldsymbol{a}$ on $\boldsymbol{x}$ — supervised imitation, not SSL.
- *Q7.* $K \times T = 2{,}000$ predictor calls per evaluation (100 candidates each rolled 20 steps). Over 50 replanning steps: 100,000 calls — the argument for a small, fast predictor on a frozen feature space.
- *Q8.* Train $I_{\boldsymbol{\phi}}(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}) \to \hat{\boldsymbol{a}}_t$ on whatever motion episodes are available (or on a small proprioceptively logged subset); the inferred $\hat{\boldsymbol{a}}_t$ supervises $p_{\boldsymbol{\theta}}$ on all the unlabeled video — the right-most (recoverable) branch.
- *Q9.* Collapse i: shrink the encoder outputs to a constant so Eq. (13.4) is trivially zero — defense: variance/covariance regularization (VICReg-style). Collapse ii: make enc and $p_{\boldsymbol{\theta}}$ co-contract until both predict each other's drift-addled outputs — defense: stop-gradient on the target branch (SimSiam/BYOL-style).
- *Q10.* Yes. A denser loss rewards *appearing* near the goal early and staying there, which shortcuts the task when the goal frame is reachable by a cheap artifact (e.g. occluding the block with the pusher). The fix is potential-based shaping or keeping the loss terminal-only but discounting replan frequency.
- *Q11.* Benign when the per-step error is small relative to the curvature (the rollout snakes around the truth but stays in-tube); catastrophic when the error grows faster than the dynamics' contraction — e.g. near contact discontinuities in Push-T, where touching the T bifurcates the future.

- *Q12.* The action space is {left, right}, so the space of ten-step plans is the finite set $\{0,1\}^{10}$ of size 1,024: exact enumeration is complete and costs only 10,240 MLP calls. Push-T's actions live in $\mathbb{R}^2$, so horizon-T plans fill $\mathbb{R}^{2T}$ and no finite enumeration exists; CEM restores tractability by sampling K candidates from a Gaussian and refitting the distribution to the elites each round. The enumerative planner implicitly assumes the optimum of the discrete set is enough — that is, that the best available sequence is good — whereas CEM only ever *approximates* the argmin of $\mathcal{L}_{\text{plan}}$ over the continuous space, with no completeness guarantee.
- *Q13.* Not the same property: correspondence needs *static* discriminability of parts; dynamics needs the space to be a *predictable manifold under action*. The separating experiment: freeze both feature sets, train an action predictor on each, and compare zero-shot planning success — the lecture's DINO-WM result already suggests DINOv2 passes the harder test on Push-T-like data while the DIFT evidence is about matching, not motion.
- *Q14.* Frozen is minimax-rational when the deployment distribution is covered by pretraining (natural images, generic manipulation) and per-task data is scarce: you inherit 100M+-image structure for free and never break it. End-to-end wins precisely in the out-of-distribution regimes — new sensor modalities, shifted domains — where the frozen space is not a sufficient statistic and no amount of dataset curation can be re-lived retrospectively.

13.9 Concept Map

Concept Map — how this chapter connects

Backward references. Chapter 1 (the central dogma: structure as free signal) is reapplied — now the "signal" is the next frame itself. Chapter 2 supplies the distance and embedding machinery that Section 13.1 reuses verbatim. Chapter 7's DIFT (Tang et al., 2023) (Section 13.1.1) is the evidence that frozen SSL features are semantically rich enough to plan in. Chapter 11's DINOv2 (Oquab et al., 2024) is the frozen encoder itself; its data engine is why the frozen space can be trusted on natural images. Chapter 10's collapse defenses are the counterfactual of what would go wrong if the encoder were unfrozen (Exercise Q9). The CartPole companion case study (Section 13.3.4) sharpens Chapter 1's dogma into the chapter's refrain: the encoding of the next recorded observation is the training target, so *time acts as the supervisor* — Eq. (13.5) is Eq. (13.4) run through a laundry cycle and shrunk.

Sideways. Chapter 7 also supplies the supervised-imitation contrast — Diffusion Policy (Chi et al., 2023) occupies the "not SSL" leaf of Figure 13.3 precisely because its actions are recorded by a human.

Forward references. Chapter 14 (system-level SSL) explains where the action stream of Figure 13.3 can come from for free: proprioception and multi-rig sensor buses — the case studies there are this chapter's abstract pipeline made flesh on real robots. Part III, especially Chapter 16, revisits the frozen-vs-trained state-space decision as one axis of the field's central trade-off. The research frontier above points from DINO-WM (Zhou et al., 2024) to the broader latent-world-model program the field is now scaling.

CHAPTER 14

System-Level Self-Supervision

A system that observes the world also produces everything it needs to learn about it.

In this chapter

14.1 From Datasets to Systems . 623 • Paired data, and when it counts as self-supervised • Sensor rigs make pairs by default • The definition
14.2 A Zero-Background Guide to 3-D and Robot Words . . . . . 625 • From pixels to voxels: describing space with little cubes • Where am I, and which way am I looking? Coordinates and pose • Keeping track while moving: odometry, drift and SLAM • Looking at a map from a camera: rays, rendering, occlusion • Motion seen in images • The robot's senses
14.3 Maps as Supervision: A Robot That Fixes Its Own Segmentation 636 • The whole idea first, as a story (almost no maths) • From the story to the formulas: why each piece of maths is needed • Background: what you need first • The problem • How it works, step by step • The whole loop on a tiny room, step by step • Not forgetting: the replay buffer • Where to look: uncertainty and planning • What can go wrong
14.4 Geometry as Supervision: Depth, Camera Motion and Flow 654 • The whole idea first, as a story (almost no maths) • From the story to the formulas: why each piece of maths is needed • The details • Background: how a camera turns 3-D into 2-D • The idea: three quantities that must agree • How it works, step by step • The loss, term by term
14.5 Own Experience as Supervision: Where Can I Walk? 667 • The whole idea first, as a story (almost no maths) • From the story to the formulas: why each piece of maths is needed • Background: what you need first • The problem • How it works, step by step
14.6 Other Sensors as Supervision: Teaching Microphones to See Cars 679 • The whole idea first, as a story (almost no maths) • From the story to the formulas: why each piece of maths is needed • Background: what you need first • The problem • How it works, step by step • The losses, term by term
14.7 Interaction as Supervision: A Robot That Practises on Its Own 689 • The whole idea first, as a story (almost no maths) • From the story to the formulas: why each piece of maths is needed • Background: reinforcement learning in five words • The problem • How it works, step by step • The two algorithms, line by line
14.8 The Big Picture: Three Ingredients of System-Level SSL 701
14.9 Chapter summary 702
14.10 Check Your Understanding 703

Chapter roadmap

Before you start. Three ideas from earlier chapters come back here, and each is recapped briefly where it is used: a *pseudo-label* is a training target made by a model rather than by a person (Chapter 12); in *teacher–student distillation* one network's outputs become another network's targets (Chapters 11–12); and *DINO features* are general-purpose image descriptors learned without labels (Chapter 11). Everything from robotics — cameras, camera poses, 3-D maps, optical flow, rewards — is explained from scratch: first in a zero-background guide to 3-D and robot words (§14.2, with a one-page cheat sheet), then again in a *Background* part at the start of each section that needs it.

The idea in one sentence. A robot, a car or a phone measures the world with several sensors at once and keeps measuring it over time, so the same fact is often measured twice — and whenever that happens, one measurement can serve as the label for the other.

The five systems in this chapter, and what each one measures twice:

- *Maps* (§14.3): a single camera frame, and a 3-D map built from many frames of the same room.
- *Geometry* (§14.4): how far away things are, how the camera moved, and how the pixels moved — three quantities that must agree.
- *The robot's own body* (§14.5): what the camera saw ahead, and what the legs felt a few seconds later when they walked there.
- *Another sensor* (§14.6): what the cameras see, and what the microphones hear, at the same moment.
- *Acting* (§14.7): what the robot predicted would happen when it grasped, and what actually happened.

How every section is built. *Background* (the words and pictures you need) → *the problem* → *the idea in plain words* → *how it works, step by step* → *the loss, term by term, with numbers* → *what can go wrong.* If a section feels hard, read only its Background and its plain-words box first, then come back.

Interactive companions. Every system in this chapter has a small web page where you can run it yourself — click pixels, move sliders, watch the losses change. Look for the "try it interactively" box near the end of each section.

What the exam asks of you. Lecture 09 opens (slide 2) by saying the exam has no derivations of losses. You should be able to *explain* a loss and each of its parts, *choose* the right loss for a problem, *spot mistakes* in one, and *implement* it. That is why every loss in this chapter is taken apart piece by piece and checked on small numbers.

Every chapter so far has treated self-supervision as something done to a *dataset*: a fixed pile of images sitting on a disk, from which we invent a task — mask some patches, rotate the image, crop it twice — that produces a target without a human. This chapter changes the point of view. The learner is now a *machine that lives in the world*: a robot, a car, a

phone. Such a machine records sensor data all the time, moves, and acts, whether or not anyone learns from it. The question of the chapter is: *how much can such a machine teach itself, from its own data alone?*

14.1 From Datasets to Systems

14.1.1 Paired data, and when it counts as self-supervised

Start with the idea the lecture starts with (Lecture 09, slide 4). **Paired data** means examples that come in matched twos: a sketch of a handbag together with a photo of that handbag, or a picture of a road together with the same road under different conditions. Pairs like these train *conditional* generators (Chapter 6): give the sketch, get the photo.

Is that self-supervised? Strictly, no. Somebody had to find or make each pair — draw the sketch, match the photos — and that effort is a kind of labelling. The lecture calls such pairs "rather semi-supervised". But there is an exception. If the pairs can be *mined without any manual effort*, they cost nothing and the method counts as self-supervised. Two examples you already know:

- *Augmentations*: two random crops of one image are a pair, made by a program (Chapters 9–11).
- *Super-resolution*: shrink a photo; the small version is the input and the original is the target. Again a program made the pair.

14.1.2 Sensor rigs make pairs by default

Now the pivot of the lecture (slide 5): *most machines that observe the world carry a "rig" of sensors.* The slide shows three: a phone with several cameras, a self-driving car covered in cameras and laser scanners, and a humanoid robot. A rig records all its sensors *on one clock*. So at every instant it produces several measurements of *the same scene at the same time* — a pair (or a triple, or more) that nobody had to make. Figure 14.1 shows what this looks like as data.

New words used from here on — sensors and robots

Sensor. A device that measures something: a camera (light), a depth camera or laser scanner (distance), a microphone (sound), a thermal camera (heat), a motion sensor or IMU (acceleration and turning), wheel or joint encoders (how far each wheel or joint has turned).

Modality. The *kind* of signal a sensor gives: images, depth, sound, heat, motion. "Cross-modal" means from one kind to another.

Rig; synchronised. Several sensors fixed to one body and recorded on one clock, so their readings can be lined up in time.

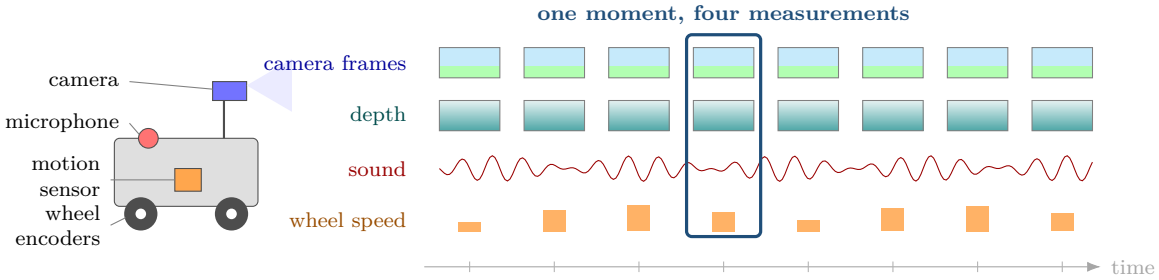


Figure 14.1. What a sensor rig records (after Lecture 09, slides 5 and 44). Each row is one sensor's stream; all rows share one clock. A *column* is one moment of the world seen in four different ways. Nobody made these pairs — the hardware makes them, continuously, for free.

How to read it: Read down one column: camera, depth, sound and wheel speed are all about the same instant. Any row can be a target for a model that reads another row.

Pose. Where something is and which way it faces: a position (x, y, z) plus an orientation (three turning angles). The camera pose is where the camera was, and where it was pointing, when it took a picture.

Proprioception. A robot's sense of its own body: joint angles, motor currents, how fast it is actually moving. Humans have it too — you can touch your nose with your eyes shut.

14.1.3 The definition

Put together, these observations give the lecture's central definition. It appears twice in the deck: as the thesis (slide 6) and again in the summary (slide 43).

Key Concept — System-level self-supervision

On a *system level*, self-supervised learning means **learning from all available data of the system, without any external input or labelling**. The test is not what the loss looks like, but *where the targets come from*. If every target can be traced back to the system's own sensors, its own actions, or physical laws that relate them, the method is system-level self-supervised — even if it uses a mapping program or geometry along the way.

Simplified intuition vs. more precise view

Simplified: a robot's sensors "see the same thing" in different ways, so each sensor can teach the others. **More precise:** the sensors do not record copies of one signal. A microphone and a camera record very different things; what they share is a *fact about the world* — a car drove past on the left — that each sees only partly. Learning across sensors works because of that shared fact. For each system below, ask two questions:

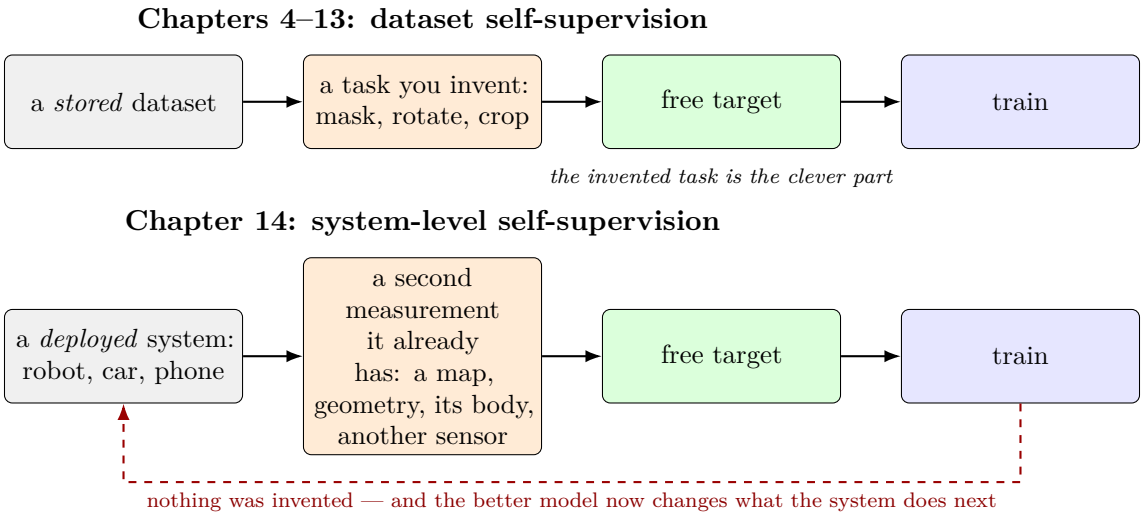


Figure 14.2. The move this chapter makes. Dataset self-supervision invents a task to manufacture a target. System-level self-supervision does not need to invent anything: a deployed system already produces a second, independent measurement of the same world, and that second measurement is the label.

How to read it: The two rows are almost the same. The differences are the orange box — invented versus already there — and the red arrow, which exists only in the bottom row: a system's learning changes what it does, and what it does changes what it learns from next.

which shared fact is used? and *how does the system notice when the sensors disagree for a bad reason?* (A moving car breaks the geometry of §14.4; unfamiliar ground breaks the familiarity test of §14.5.)

Figure 14.3 puts the whole chapter on one page. Keep it nearby: every section fills in one row.

14.2 A Zero-Background Guide to 3-D and Robot Words

This chapter borrows many words from 3-D vision and robotics: *voxel, point cloud, pose, odometry, SLAM, ray casting, proprioception*... If you have never worked with 3-D data or robots, read this section first. It assumes nothing except that you know what a digital photo is. Every word gets an everyday picture, and the most important ones get a diagram. Table 14.2 at the end of the section collects them all on one page, for looking up later.

14.2.1 From pixels to voxels: describing space with little cubes

Pixel. A digital photo is a flat grid of tiny squares. Each square is a **pixel** and stores one colour. That is all a camera gives you: a grid of colours, *flat*, with no information about how far away anything is.

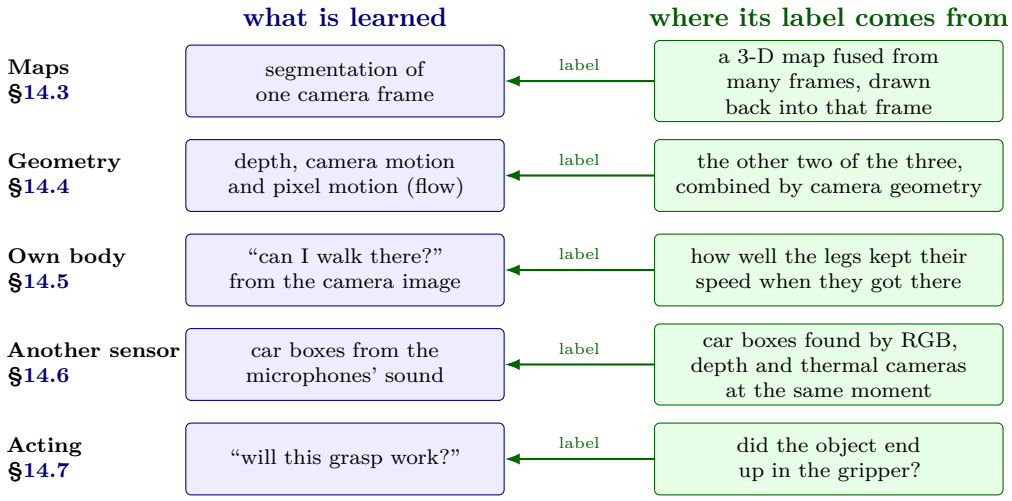


Figure 14.3. The five systems of this chapter, one per row. Left: the model being trained. Right: the second measurement that supplies its label — in every row, something the system was producing anyway.

How to read it: Cover the right column and ask, for each row, "where could a label come from without a person?"; then uncover it.

Depth, depth image, RGB-D. A *depth camera* measures, for every pixel, *how far away* the thing seen in that pixel is — say 2.4 metres. Store those distances in a grid the size of the photo and you get a **depth image**: a picture in which the "colour" of each pixel is a distance (drawn as yellow for near and purple for far in the lecture's figures). A camera that records a normal colour photo and a depth image at the same moment is called **RGB-D** (RGB = red, green, blue, i.e. colour; D = depth). Many phones and robot cameras do this.

Point cloud. A pixel tells you a *direction* (which way you are looking through that square); its depth tells you *how far* along that direction. Direction plus distance is a point in space. Do this for every pixel and you get thousands of 3-D points, a **point cloud**: think of spraying tiny dots of paint onto every surface the camera can see. Figure 14.6 walks through it from above; Figures 14.7 and 14.8 show a real depth image and a real point cloud.

Voxel. A point cloud is just a heap of loose dots. To store knowledge about space in an organised way, we cut space into a regular grid of small cubes, like a 3-D chessboard or a world built from Lego bricks (or Minecraft blocks). Each cube is a **voxel**, short for "volume pixel": it is to 3-D space what a pixel is to a photo (Figure 14.4; real examples in Figure 14.5). A typical voxel is a few centimetres on each side. Like a pixel stores a colour, a voxel stores *something about the little piece of space it covers*: whether it is empty or filled, its colour, or — in this chapter — a tally of which object class (wall, chair, floor) it has been seen as.

3-D map. A map in this chapter is simply the collection of all voxels the robot has seen, each with its stored information. Unlike a single photo, the map stays put as the robot moves around: it is the robot's memory of the room.

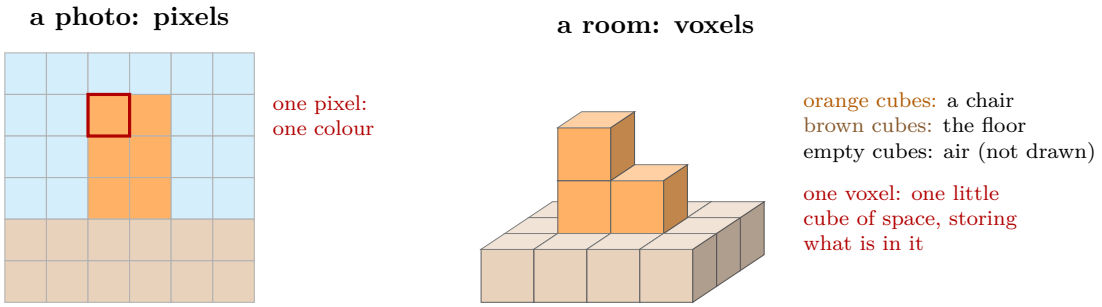
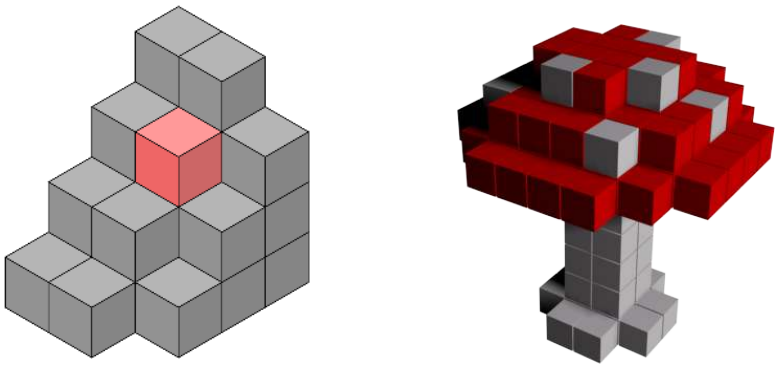


Figure 14.4. A pixel is one small square of a flat picture; a voxel is one small cube of 3-D space. A photo is a grid of pixels; a 3-D map is a grid of voxels, like a world built from Lego bricks. Most voxels in a room are empty air and are simply not stored.



(a) voxels stacked in a grid; one voxel highlighted

(b) a mushroom built only from voxels

Figure 14.5. Real voxel pictures. (a) A stack of voxels with one of them shaded: each cube is one "volume pixel". (b) A 3-D model made entirely of voxels. Every cube stores one thing (here a colour); from far away the cubes add up to a recognisable shape, exactly as pixels add up to a photo. A robot's map stores, in each cube, a class tally instead of a colour.

Source: (a) "Voxels.svg" by Vossman, M. W. Toews and cmglee, CC BY-SA 2.5; (b) "Voxel amanita.png" by Acodered, CC BY-SA 3.0; both via Wikimedia Commons.

14.2.2 Where am I, and which way am I looking? Coordinates and pose

Coordinates and frames. To say where something is, you need a starting point and directions: "3 metres east and 2 metres north of the front door, 1 metre above the floor" is a position $(x, y, z) = (3, 2, 1)$. The chosen starting point and directions are called a **coordinate frame**. The same object has different coordinates in different frames, just as "on the left" depends on who is speaking. Two frames appear all the time: the *world frame*, fixed to the room, and the *camera frame*, which travels with the camera ("2 m straight ahead of me").

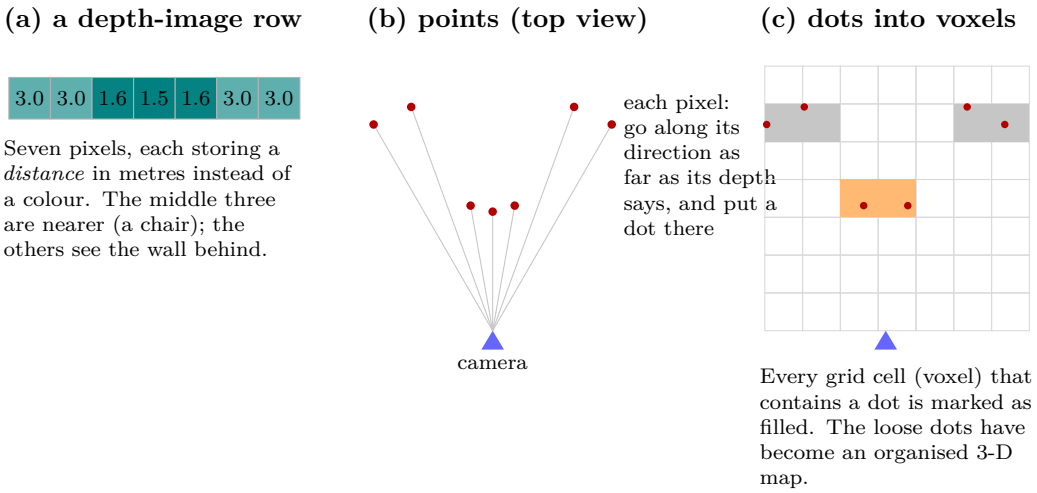


Figure 14.6. From a depth image to a voxel map, drawn from above to keep it flat. (a) A depth image stores distances. (b) Each pixel's distance, along that pixel's viewing direction, gives one 3-D point: together they form a point cloud. (c) Dropping the points into a grid of cubes gives voxels. Doing this for many frames, taken from different places, fills in the whole room.

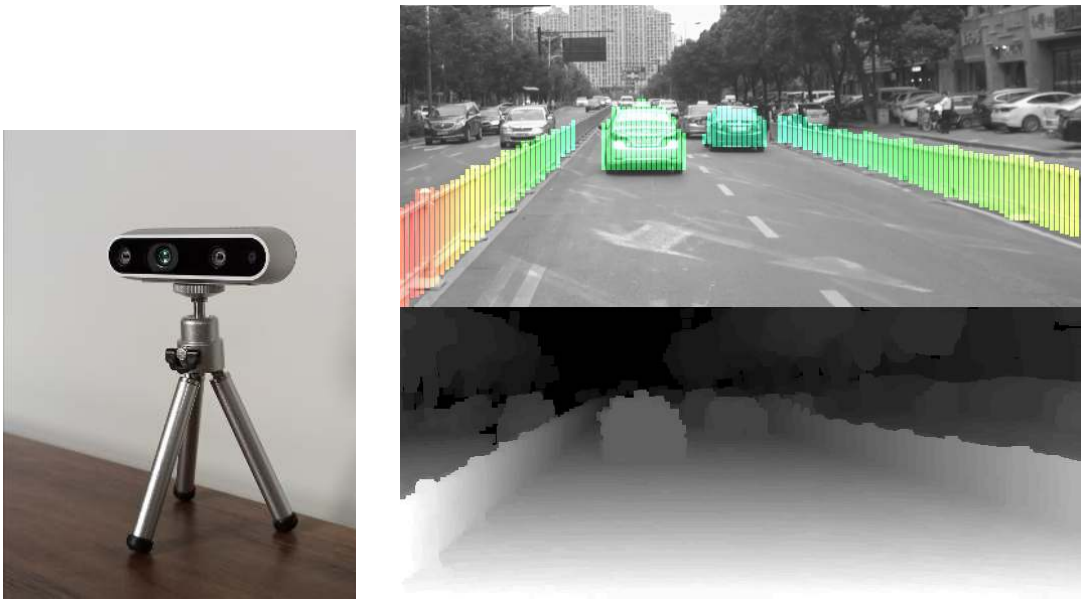
How to read it: Follow one number from (a) — say 1.5 *— to its dot in (b) and its orange cell in (c).*

Pose. A **pose** is *where something is* plus *which way it is facing.* On a shopping-mall map, the "you are here" dot is a position; add an arrow showing which way you face and it is a pose. In 3-D, a pose has six numbers: three for position (x, y, z) and three turning angles — like an aeroplane that can point its nose left or right, up or down, and roll its wings. A *camera pose* says where the camera was and where it was pointing when it took a photo. Knowing the pose is what lets us turn "2 m straight ahead *of the camera*" into "this spot *in the room*".

Trajectory and ego-motion. As the robot drives, its pose changes. The list of poses over time is its *trajectory.* The change of pose between two moments — "I moved 0.5 m forward and turned 10° left" — is the robot's own motion, called **ego-motion** ("ego" = self). In §14.4 it is written $(R, \boldsymbol{t})$: a rotation R (how much it turned) and a translation $\boldsymbol{t}$ (how far it moved).

14.2.3 Keeping track while moving: odometry, drift and SLAM

Odometry. How does a robot know its own pose? The simplest way is to count: wheels turn a known amount per revolution, so counting wheel turns tells you how far you went, and comparing left and right wheels tells you how much you turned. This is **odometry** — the same idea as a car's odometer, or counting your steps with your eyes closed. The sensors that count wheel or joint turns are called *encoders.* A motion sensor (*IMU*, inertial measurement unit) helps too: like your inner ear, it feels acceleration and turning.



(a) an RGB-D camera (b) a street, and its depth image

Figure 14.7. Real depth sensing. (a) An Intel RealSense D435, a small RGB-D camera: the middle lens takes the colour photo, the two outer lenses form a stereo pair that measures depth. This model is the camera used on the real robot of the planning experiment in §14.3 (Zurbrügg et al., 2022). (b) A real street seen by a car's stereo camera. Bottom half: the depth image — bright means *near* (the road just in front of the car, the car ahead), dark means *far*, black means "no measurement". Top half: the photo, with vertical bars coloured by distance (red near, blue far).

Source: (a) "Intel Realsense depth camera D435.jpg" by Marc Auledas, CC BY-SA 4.0; (b) "Stixels.png" by Tino, MIT licence; both via Wikimedia Commons.

Drift. Counting works well for a few metres, then goes wrong. Each wheel slips a little, each small error is added to the next, and the estimated position slowly wanders away from the truth. This is called **drift**. Walk around a dark room counting steps, and after a few laps you will not be where you think you are.

SLAM. **SLAM**, *simultaneous localisation and mapping*, fixes drift by using what the camera sees. Imagine exploring a dark house with a torch. You draw a map as you go (*mapping*), and you keep track of where you are on that map (*localisation*). The two help each other: the map tells you where you are ("I see the red door I drew earlier, so I must be here"), and knowing where you are lets you add new rooms to the map in the right place. When you recognise a place you have already mapped, you can correct all the drift accumulated since you were last there; this moment is called *loop closure* (Figure 14.10). For this chapter, treat SLAM as a black box whose output is *a good pose for every camera frame*.

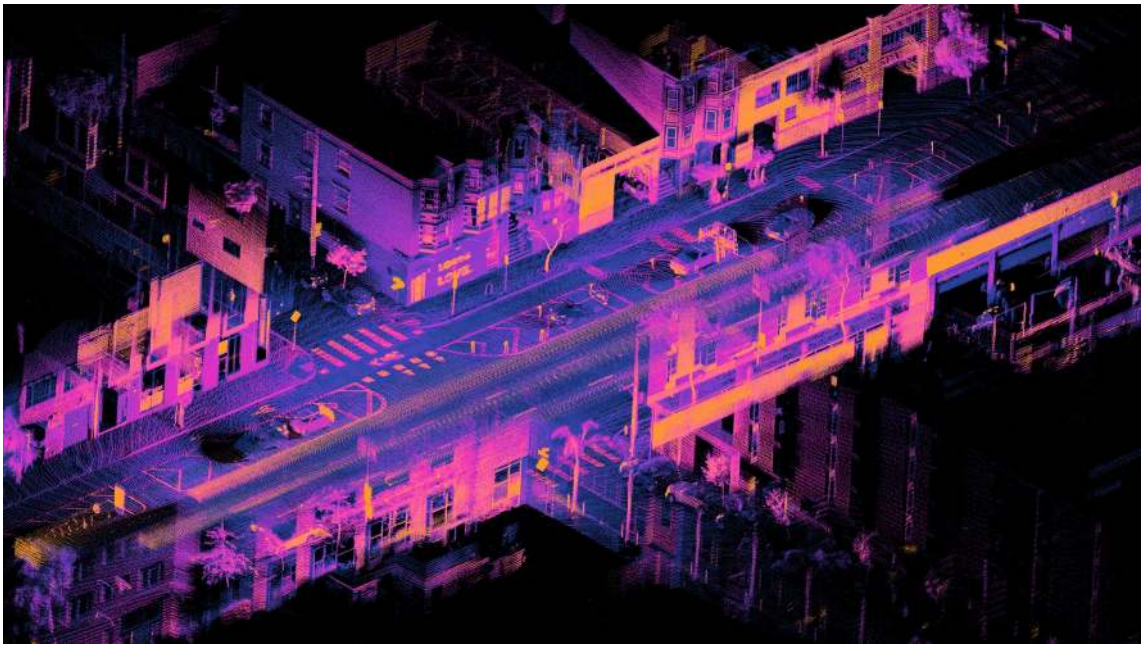
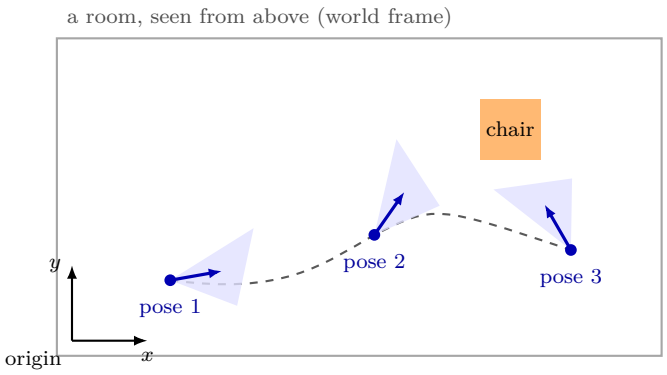


Figure 14.8. A real point cloud: a street crossing in San Francisco, scanned by a LiDAR on a moving car. Nothing here is a photo — every speck is one 3-D point where a laser pulse bounced back, about 23 million of them, collected in 18 seconds. Because the car was moving, each scan had to be placed using the car's pose at that moment, estimated by SLAM; that is why the buildings line up across many scans instead of appearing twice. Colours show how strongly each surface reflected the laser (road markings and signs stand out).

Source: "Ouster OS1-64 lidar point cloud of intersection of Folsom and Dore St, San Francisco.png" by Daniel L. Lu, CC BY 4.0, via Wikimedia Commons.

How to read it: Find the crosswalk stripes and the parked cars: they are made of dots, not pixels. This is what "point cloud" means.



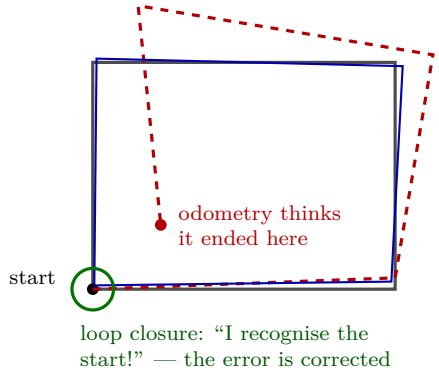
Pose = position (the dot) + facing direction (the arrow). The blue fan is what the camera sees from there.

Trajectory = the dashed path through all the poses.

Ego-motion = the change from one pose to the next: how far the robot moved and how much it turned.

The chair has one position in the room, but it is "ahead-right" in pose 2's camera frame and "ahead-left" in pose 3's.

Figure 14.9. Pose, trajectory and ego-motion, from above. The robot's camera has a different pose at each moment; the chain of poses is its trajectory. Knowing each pose is what lets many photos be combined into one map of the room.



Black: the robot’s true path, one lap of a room.

Red dashed: odometry alone — every corner adds a little error, so the estimate drifts and does not even close the loop.

Blue: SLAM — it recognises places it has seen, so it pulls the estimate back onto the true path.

Figure 14.10. Why robots need SLAM. Counting wheel turns (odometry) drifts; SLAM uses the camera to recognise known places and corrects the drift. Map-based labels (§14.3) need these corrected poses: with drifting poses, the same wall would be drawn twice in two places.

14.2.4 Looking at a map from a camera: rays, rendering, occlusion

Ray. A *ray* is a straight line that starts at a point and goes on in one direction — the path a laser pointer’s beam would take. Every pixel of a camera corresponds to one ray: the line from the camera out through that pixel into the world.

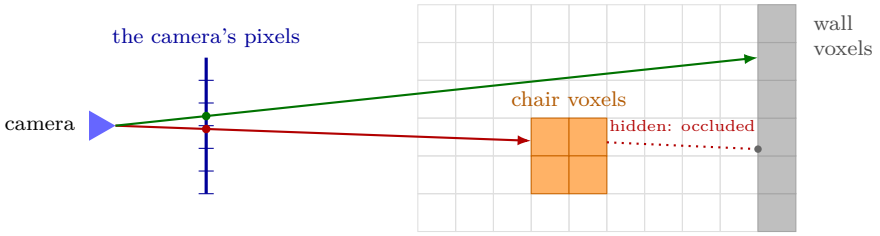
Ray casting and rendering. Suppose you have a voxel map and a camera pose, and you want to know what the camera *would* see. For each pixel, follow its ray out into the map until it hits the first filled voxel, and copy that voxel’s information (its colour, or its class) into the pixel. This is **ray casting**; doing it for all pixels *renders* an image of the map from that pose — like taking a photo of a Lego model (Figures 14.11 and 14.12). In §14.3 the voxels store classes, so the rendered image is a *label image*: a pseudo-label drawn from the map.

Occlusion. A ray stops at the *first* thing it hits; what lies behind is hidden, or **occluded**. A chair occludes the part of the wall behind it. Occlusion matters whenever we compare two views: a spot visible in one photo may be hidden in the next, so it has no match there.

Projection and reprojection. Going from a 3-D point to the pixel where it appears in a photo is called *projection*; it is ray casting run backwards. *Reprojection* means projecting a 3-D point that was measured at one time into a photo taken at another time — as WVN does when it paints the robot’s later footsteps onto an earlier photo (§14.5).

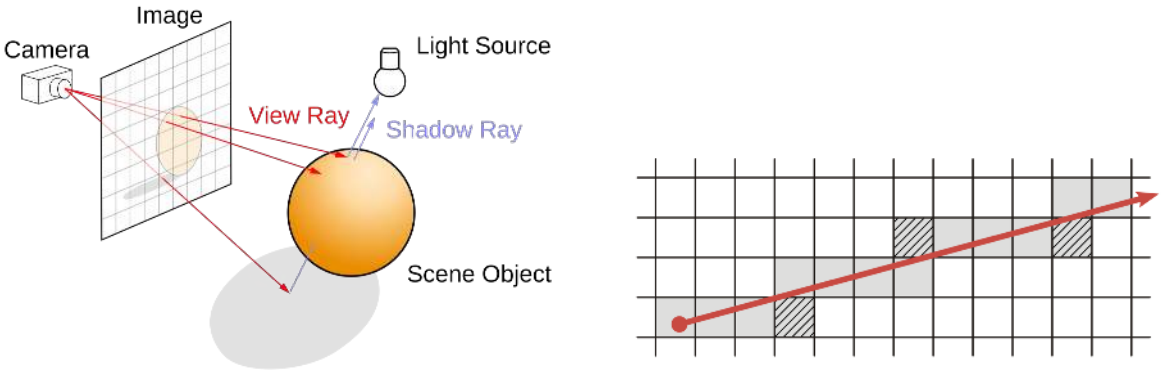
14.2.5 Motion seen in images

Optical flow. Film a scene while moving and compare two frames: everything has slid a little across the picture. For each pixel, the small arrow from where it was to where it went is its **optical flow**.



Red ray: leaves the camera through one pixel, hits a chair voxel first ⇒ that pixel is labelled "chair". The wall behind it is *occluded*. **Green ray:** passes above the chair and hits the wall ⇒ that pixel is labelled "wall". Doing this for every pixel *renders* a whole label image from the map.

Figure 14.11. Ray casting, seen from above. Each pixel of the camera sends a ray into the voxel map; the first filled voxel it meets decides what that pixel shows. This is how a map is turned back into an image — and, in §14.3, into a pseudo-label.



(a) one ray per pixel of the image (b) a ray walking through a voxel grid

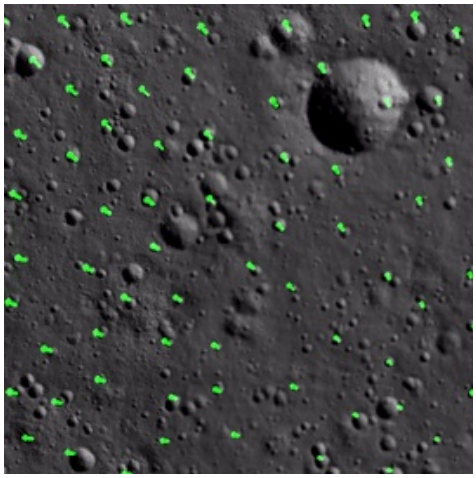
Figure 14.12. Ray casting as it is drawn in computer graphics. (a) From the camera, one *view ray* goes through each square (pixel) of the image and on into the scene; whatever it hits first decides that pixel's colour. (The "shadow rays" towards the lamp are only needed for lighting and play no role in this chapter.) (b) The same idea in a grid, seen from above: the ray passes through the shaded cells one by one; the hatched cells are the filled ones it would hit. The first filled cell along the ray is the one whose label goes into the pixel.

Source: (a) "Ray trace diagram.svg" by Henrik, CC BY-SA 4.0; (b) "Ray in voxel grid.svg" by Phrood, CC BY-SA 3.0; both via Wikimedia Commons.

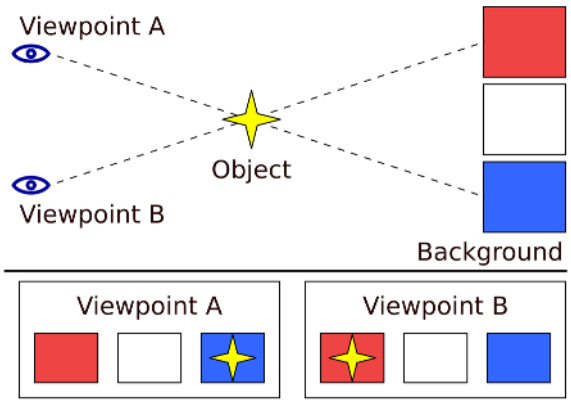
Parallax. Look out of a train window: the fence rushes past, the hills barely move. Near things slide across your view more than far things. This is **parallax**, and it is why flow tells you about depth (Figure 14.33).

Rigid (static) scene. A scene is *rigid* or *static* if nothing in it moves by itself — only the camera moves. Then every pixel's flow is explained by depth and camera motion alone. A passing car breaks this, because it moves on its own.

Figure 14.13 shows both ideas in real pictures.



(a) optical flow on a landing video (zoomed in)



(b) parallax

Figure 14.13. (a) Real optical flow: one frame of a (simulated) video of a spacecraft descending towards the Moon. Each short green arrow shows how far one surface feature moved between this frame and the next — that arrow field is the optical flow. (b) Parallax: seen from viewpoint A, the star (near) sits in front of the blue square; from viewpoint B it sits in front of the red one. The far background did not move, but the near object appears to jump. The size of that jump is what tells a moving camera how far away things are.

Source: (a) One frame of "Optical Flow Moon Landing.gif" by Sassoeffe, CC BY-SA 4.0; (b) "Parallax Example.svg" by Sarang, public domain; both via Wikimedia Commons.

14.2.6 The robot's senses

Table 14.1 lists every sensor that appears in this chapter, with a human equivalent; Figure 14.14 shows some of the hardware.

Table 14.1. The sensors used in this chapter, in plain words.

Sensor	What it measures	Human equivalent / picture
Camera (RGB)	colour of each pixel	your eyes (one eye: no depth)
Depth camera; RGB-D	distance of each pixel; colour + depth together	judging distance with two eyes
Stereo camera	two cameras a known distance apart; depth from how much the two images differ	your two eyes
LiDAR / laser scanner	distances measured by timing laser pulses, many directions per second	a very fast laser tape measure that spins
Thermal camera	heat given off by objects; works in the dark	a snake's heat sense
Microphone array	sound at several places at once; the tiny time differences reveal direction	your two ears locating a siren
IMU (motion sensor)	acceleration and turning speed	your inner ear (balance)
Wheel / joint encoders	how far each wheel or joint has turned	counting your steps; knowing your elbow angle
Proprioception	the robot's sense of its own body: joint angles, motor effort, actual speed	touching your nose with eyes closed; feeling that walking in sand is hard

(a) an IMU board

(b) a rotary encoder

(c) an ANYmal legged robot

Figure 14.14. Real hardware behind the words. (a) An IMU: a coin-sized board whose chips feel acceleration and turning (the robot's "inner ear"). (b) A rotary encoder: it counts how far the shaft through its middle has turned — fixed to a wheel or a joint, it provides odometry and joint angles. (c) An ANYmal, the four-legged robot family used by Wild Visual Navigation (§14.5); its joints are full of encoders, which is where its proprioception comes from.

Source: (a) "SparkFun 9DoF-Razor-IMU-M0" by SparkFun, CC BY 2.0; (b) "Rotary Encoder.jpg" by Projuktiponno, CC BY-SA 4.0; (c) "ANYmal ARCHE 2019 1.jpg" by Auledas, CC BY-SA 4.0; all via Wikimedia Commons.

Table 14.2. Cheat sheet: every 3-D and robot word of the chapter.

Word	**Meaning**	**Everyday picture**
pixel	one square of a photo, one colour	a tile in a mosaic
depth image	a photo where each pixel stores a distance	a "how far" map of the view
point cloud	3-D dots, one per pixel with known depth	paint sprayed onto surfaces
voxel	one small cube of space, storing what is in it	a Lego brick; a Minecraft block
3-D map	all voxels seen so far	the robot's memory of the room
fusion	combining many frames' opinions in each voxel	a tally or vote
coordinate frame	a chosen origin and directions	"3 m east of the front door"
pose	position + facing direction (6 numbers in 3-D)	"you are here" dot + arrow
trajectory	the sequence of poses over time	a path drawn on a map
ego-motion $(R, \boldsymbol{t})$	the robot's own move between two moments: turn R, shift $\boldsymbol{t}$	"I stepped forward and turned left"
odometry	pose from counting wheel or leg motion	a car's odometer; counting steps
drift	slowly growing error of odometry	getting lost while counting steps in the dark
SLAM	build a map and locate yourself in it at the same time	exploring a dark house with a torch and a notebook
loop closure	recognising a known place and correcting drift	"I've been here before!"
ray	the straight line from the camera through one pixel	a laser pointer's beam
ray casting / rendering	making an image of a map from a pose	photographing a Lego model
occlusion	something hidden behind something else	the wall behind a chair
projection	3-D point $\rightarrow$ pixel	where a star lands on your photo
reprojection	projecting a point measured at one time into a photo from another	painting today's footsteps onto yesterday's photo
optical flow	per-pixel arrow: where each pixel moved between frames	things sliding past a car window
parallax	near things shift more than far things when you move	fence vs. hills from a train
rigid / static scene	nothing moves except the camera	a photo walk through an empty museum
superpixel	a patch of neighbouring, similar-coloured pixels	a jigsaw piece
traversability	how easily this robot can move over this ground (0 to 1)	"can I walk here in these shoes?"

14.3 Maps as Supervision: A Robot That Fixes Its Own Segmentation

14.3.1 The whole idea first, as a story (almost no maths)

Start with yourself. You walk around a museum room and look at one sculpture from five places. From one spot, a lamp's reflection makes a bronze statue look white. From the other four spots it clearly looks bronze. Asked afterwards "what colour was it?", you do not say white: four views out of five beat the one odd view. And if someone showed you only that one odd photo, you could still tell them "that white patch is really bronze", because you remember the statue, not the photo. That is the entire idea of this section: *remember the room, not the photos, and let the views vote.*

Now the robot, in four moments (Figure 14.15).

1. **Look from many places.** The robot drives through a new office and takes many photos. Its segmentation network labels every pixel of every photo: wall, cabinet, window… Most labels are right, but a few are wrong — a reflection makes a cabinet door look like a window in one photo.
2. **Put every guess into a 3-D map.** The robot knows where it was for each photo and how far away each pixel was. So every pixel's guess can be put at the right spot in a 3-D map made of small cubes. Each cube collects the guesses of all the photos that saw it.
3. **Count the votes.** Each cube takes the class with most votes. The cabinet cube got "cabinet" from four photos and "window" from the one fooled photo: cabinet wins.
4. **Look back and learn.** From each photo's position, the robot reads the map back into the photo — a cleaner label image, in which the reflection mistake is gone. It trains its network on these cleaner labels, while also rehearsing some old labelled images so it does not forget what it knew.

No person labels anything. The better network then makes a better map next time.

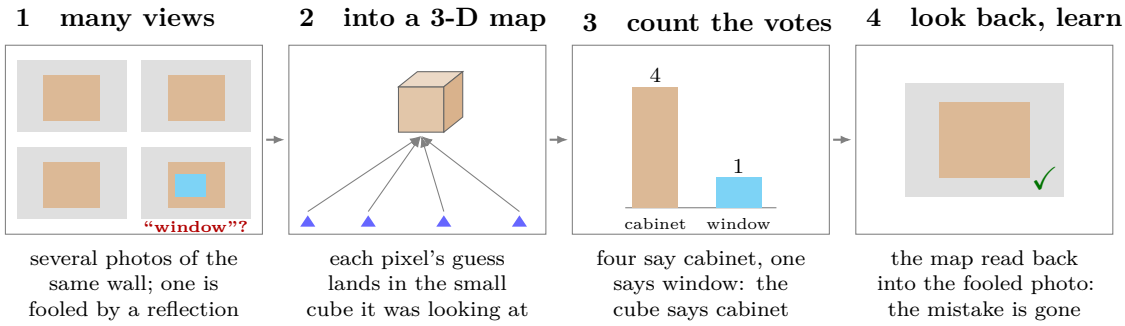


Figure 14.15. Map-based adaptation in four pictures. (1) Many photos of the same place, one fooled by a reflection. (2) Every pixel's guess is put into the small cube of the 3-D map it was looking at. (3) The cube keeps the majority. (4) Reading the map back into the fooled photo gives a corrected label to learn from.

How to read it: Follow the fooled photo (bottom right in picture 1): its "window" vote is out-voted in picture 3 and corrected in picture 4.

In plain words — the technical words, translated

semantic segmentation	labelling every pixel of a photo with what it shows
pose, SLAM	where the camera was and which way it pointed; SLAM is the method that keeps track of it
voxel	one small cube of the 3-D map
fusion	adding each photo's guesses into the cubes (the voting)
ray casting / rendering	reading the map back into a photo: for each pixel, look along its line of sight to the first filled cube
pseudo-label	the label image read back from the map, used as the training target
replay buffer	a box of old labelled images mixed into training so nothing is forgotten
mIoU	the accuracy score: how well predicted and true regions overlap, averaged over classes

Everyday analogy — a jury

One witness can be fooled by the light; a jury of twelve rarely is, as long as they did not all stand in the same spot. *Where it breaks:* if every witness made the same mistake (all of them saw the glass door as a window), the jury agrees on the mistake — and the map does too.

14.3.2 From the story to the formulas: why each piece of maths is needed

Each moment of the story raises a question the computer must answer with a number.

Question 1 (moment 2): "Which cube was this pixel looking at?" The pixel gives a *direction*, the depth camera gives the *distance*, and the camera's pose says *where the camera stood*. Together:

$$\text{point in the room} \;=\; \text{camera position} \;+\; \text{distance} \times \text{pixel's direction},$$

where the direction is first turned by the camera's orientation, and the cube is found by rounding the point down to the cube grid ("$\lfloor 2.5 \rfloor = 2 \Rightarrow$ cube v_3" in the walk-through of §14.3.6). *Why needed:* without it, guesses from different photos could not meet in the same cube.

Question 2 (moment 3): "What does the cube think?"

$$\text{votes}_v(\text{class}) \;\mathrel{+}=\; 1 \quad \text{for every pixel that lands in cube } v,$$

$$\text{label}_v = \text{the class with most votes.}$$

Why this form: counting is the simplest way to let many views out-vote a few. (Real systems add the network's probabilities instead of whole votes, so a confident view counts more than a hesitant one.) *Story numbers:* cabinet 4, window 1 $\Rightarrow$ cabinet.

Question 3 (moment 4): "What label should this pixel get?" Walk from the camera along the pixel's line of sight until the first filled cube, and copy its label. This is *ray casting*; it needs no formula beyond the one of Question 1, run in reverse. *Why needed:* it turns the map back into an image, the form the network learns from.

Question 4 (moment 4): "What should the network learn?" The ordinary per-pixel classification loss, with the map's label as the target:

$$\mathcal{L} \;=\; \text{average over labelled pixels of } \big(-\log P(\text{map's label})\big).$$

Why this form: $-\log P$ is small when the network already gives the map's label a high probability and large when it does not. *Story numbers:* the fooled pixel gave "cabinet" only $P = 0.3$, so its loss is $-\log 0.3 \approx 1.20$; once it learns $P = 0.6$, the loss falls to 0.51. Pixels with no map label are skipped, and each batch also includes old labelled images from the replay buffer.

Question 5: "Did it get better?" Measured by mIoU: for each class, the overlap of predicted and true regions divided by their union, averaged over classes. *Why needed:* a single number to compare the network before and after (e.g. 32 $\rightarrow$ 42 mIoU with curiosity-driven driving in §14.3).

Table 14.3. Map-based adaptation: the story and the maths, side by side.

In the story	In the maths	Why it is needed
"put the guess where the camera was looking"	position + distance × direction; round down to a cube	make guesses from different photos meet
"count the votes"	add one vote per pixel; take the most voted class	many views out-vote a few
"read the map back into the photo"	ray casting	turn the map into a label image
"learn from the cleaner label"	$-\log P$(map's label), plus replay	the training lesson, without forgetting
"did it get better?"	mIoU	one accuracy number

14.3.3 Background: what you need first

Semantic segmentation. A **semantic segmentation** network gives *every pixel* of an image a class: this pixel is wall, that one floor, that one chair (Figure 14.16, left). It is the per-pixel version of image classification. Its quality is usually measured by **mIoU**, the *mean intersection over union*. For one class, take the set of pixels the network *called* "chair" and the set of pixels that really *are* chair; IoU is the size of their overlap divided by the size of their union. It is 1 for a perfect match and 0 when they do not touch. mIoU is the average of the IoUs over all classes, usually written as a percentage.

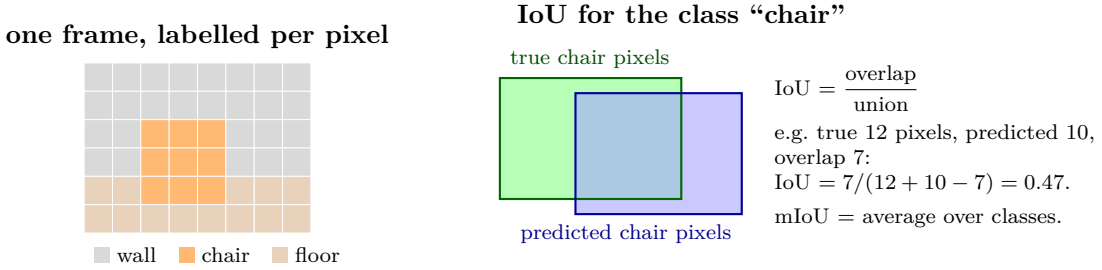


Figure 14.16. Left: semantic segmentation gives every pixel a class. Right: IoU compares the predicted and the true pixels of one class; mIoU averages it over classes. The numbers in this section ("mIoU 32 → 42") are this quantity in percent.

Domain shift. A segmentation network is trained once on a labelled dataset — here *ScanNet*, about 1,500 scans of indoor rooms labelled by people — and then used somewhere else: a particular office with its own furniture, lighting and camera height. The new place does not look like the training data, so the network makes mistakes. This mismatch is called **domain shift**. Fixing it normally needs someone to label images of the new place, which is exactly what we want to avoid.

Pseudo-labels, and why the naive version fails. Recall from Chapter 12: a *pseudo-label* is the model's own prediction, used as a training target. Fine-tuning a network on its own raw predictions mostly teaches it to repeat its mistakes (*confirmation bias*). We need a pseudo-label that is *better* than the prediction it came from. This section's source of better labels is a 3-D map.

Camera pose, odometry and SLAM. To combine many frames of one room, the robot must know where its camera was for each frame: its *pose* (position and viewing direction). Counting wheel turns gives a rough estimate (**odometry**), but small errors add up. **SLAM** — *simultaneous localisation and mapping* — solves two problems together: it builds a map of the surroundings and, at the same time, works out where the robot is in that map. The two help each other (knowing the map helps you find yourself; knowing where you are helps you extend the map). For this chapter, treat SLAM as a black box whose output is *a pose for every camera frame*. Lecture 09 (slides 7–8) shows a modern example, SNI-SLAM++ (an extension of SNI-SLAM, Zhu et al., 2024), building both a colour map and a semantic map of a room from many views, with its estimated camera trajectory drawn next to the true one.

Voxels and fusion. A simple 3-D map cuts space into small cubes called **voxels** ("volume pixels"), a few centimetres on a side. Each voxel keeps a tally: how often has it been seen, and with which class? Every new frame whose pixels look at a voxel adds its opinion to the tally. This adding-up is called *fusion*.

Rendering a label from the map. Given any camera pose, you can turn the map back into an image: from the camera, shoot one ray through each pixel, find the first voxel the ray hits, and write that voxel's winning class into the pixel. This is called *ray casting* or *rendering*, and the result is a label image that came from the map, not from a person.

14.3.4 The problem

A segmentation network, trained on ScanNet, is put on a robot that drives around a new building. Its predictions are poor, because of domain shift. Can the robot improve its own network, while it works, with nobody labelling anything? This is the question of Frey et al. (2022).

In plain words — a map is a vote

> One frame's prediction is one guess. The robot sees the same wall from twenty places, and a map, built with the known camera poses, puts all twenty guesses about each spot of the wall in one place. The majority wins. Then the map is "photographed" from each camera pose to make a label image, which is better than the network's own guess for

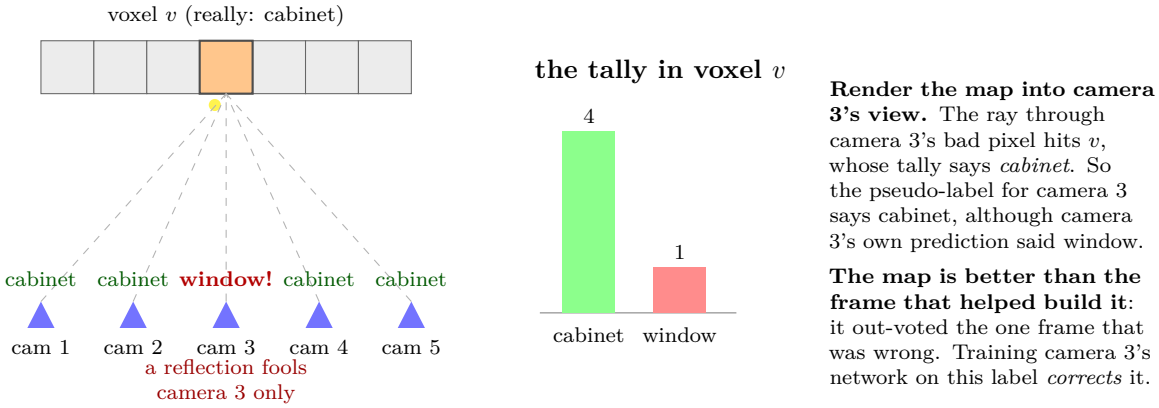


Figure 14.17. Why a map makes better labels than any single frame (the effect the lecture shows on slide 11). Seen from above: a wall made of voxels, and five camera frames look at the same voxel v. Four say "cabinet"; one, fooled by a reflection, says "window". The voxel keeps a tally of all five opinions, so rendering the map back into camera 3 gives the right label.

How to read it: Follow the dashed rays to the orange voxel, count the votes, then read the text on the right. This is the whole trick of the section; the rest is engineering.

that frame, because it carries the vote of all the other frames. Train on those labels, and the network improves — with no person involved.

14.3.5 How it works, step by step

Model at a glance: Continual adaptation from a 3-D map Frey, Blum, Milano, Siegwart & Cadena, 2022

The free label	A 3-D semantic map, fused from many frames with known poses, then rendered back into each camera view.
Why it beats the input	Many views of one spot vote; mistakes that happen in only a few views are out-voted.
Sensors	An RGB-D camera (colour plus depth) and a pose estimate from SLAM. No human labels after deployment.
Loop	predict each frame → fuse into the map → render pseudo-labels → train, mixing in old data from a replay buffer → repeat.
What it buys	A segmentation network that adapts to a new building on its own. Accuracy rises by 9.9% on average over the fixed network.
Main risk	A mistake that *every* view makes (the same object misread from all sides) survives the vote and is then taught back.

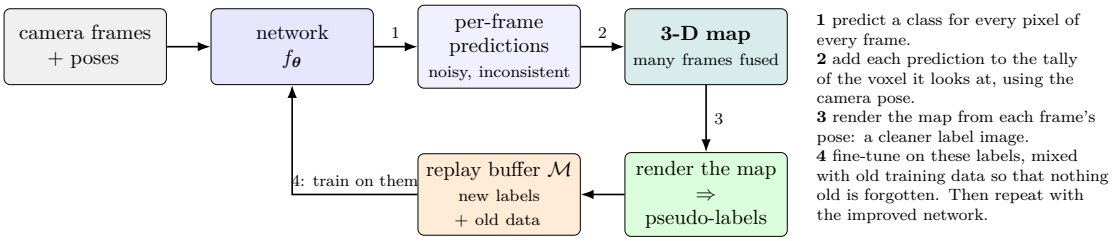


Figure 14.18. The adaptation loop of Frey et al. (2022). Nothing outside the robot is consulted: the supervision comes from the fact that a fused map is more consistent than the single-frame predictions that built it.

How to read it: Follow the numbers 1 to 4. The label gets better *as it goes round the loop (at step 2, the vote), which is the only reason the loop is worth running.*

Step by step — one round of map-based adaptation

Step 1. Predict. The current network labels every pixel of each new RGB-D frame. Many of these labels are wrong.

Step 2. Fuse. Using the depth image and the SLAM pose, each pixel is placed at a 3-D point, and its predicted class is added to the tally of the voxel at that point. After many frames each voxel holds the opinions of all the views that saw it.

Step 3. Render. For each stored frame, the map is ray-cast from that frame's pose; each pixel gets the winning class of the voxel it hits. These rendered images are the pseudo-labels. Because they come from one 3-D map, they agree with each other across views by construction.

Step 4. Train, without forgetting. The network is fine-tuned on the rendered pseudo-labels, mixed with samples from a *replay buffer* (next subsection).

Step 5. Repeat. The improved network makes better predictions, which make a better map, which makes better labels.

Section 14.3.6 below runs these steps by hand on a tiny room, with every vote and every label shown.

14.3.6 The whole loop on a tiny room, step by step

Everything above becomes clear on one small, fully worked example. We shrink the world until every number fits on the page, and follow it through all four steps.

The set-up. Look at the room from above. One wall is made of six voxels, $v_1, \ldots, v_6$. What is really there (which nobody tells the robot) is:

voxel	v_1	v_2	v_3	v_4	v_5	v_6
truth	wall	cabinet	cabinet	window	window	wall

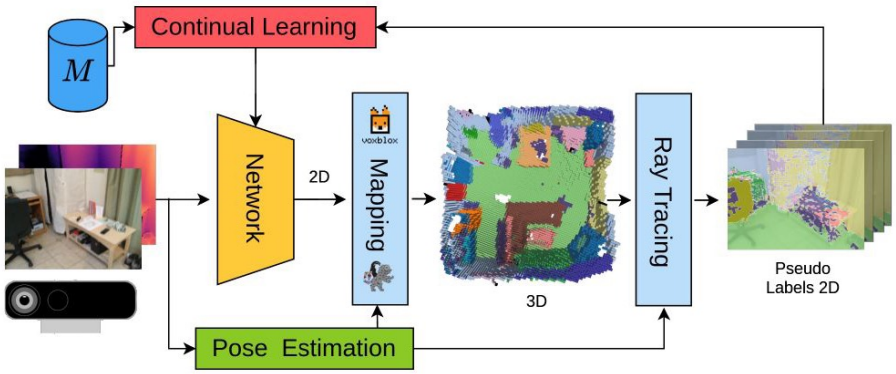


Figure 14.19. The same loop as drawn in the lecture (Lecture 09, slide 10; after Frey et al., 2022). RGB-D frames go through the network (yellow); the 2-D predictions and the pose estimate are fused into a 3-D map ("Mapping"); ray tracing turns the map into 2-D pseudo-labels; "Continual Learning" trains the network on them, drawing old samples from the memory M.

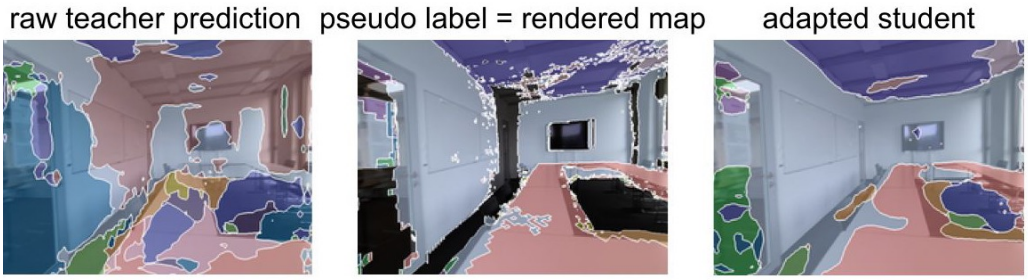


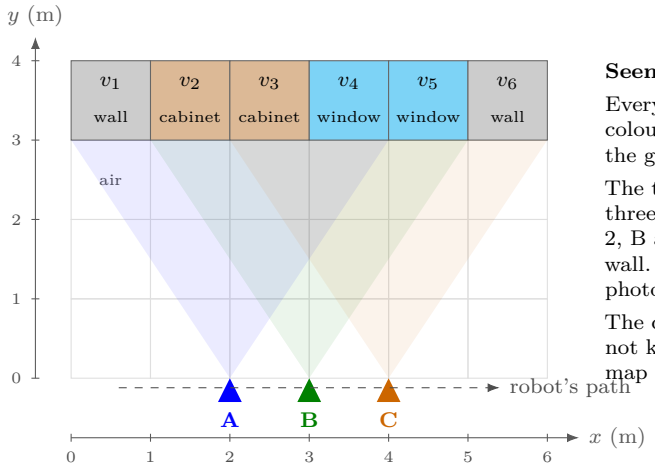
Figure 14.20. One frame, three times (Lecture 09, slide 11; after Frey et al., 2022). Left: the pre-trained network's raw prediction — patchy, with classes smeared across the wall. Middle: the pseudo-label rendered from the map — clean where the map is confident, black where the map has no information, so there is no label there. Right: the network after adaptation, which has learned from the middle picture; its output is far cleaner than the left one, including in regions the map left black.

How to read it: Compare the left and middle pictures first (the map is cleaner than the frame), then the middle and right ones (the student ends up better than its own label, because it also fills in the black holes).

The robot drives past and takes three photos, frames A, B and C, from three poses. Seen from above, a camera photo of a wall is just one *row* of pixels; to keep it tiny, each frame has four pixels, and each pixel looks at one voxel. Frame A sees v_1–v_4, frame B sees v_2–v_5, frame C sees v_3–v_6.

The same room, with real numbers. To see every calculation, give the room coordinates (Figure 14.21). Measure in metres, looking down from above: x runs along the wall, y runs from the robot's path towards the wall. Each voxel is a 1 m cube, so voxel v_i covers x from $i-1$ to i, and the wall voxels all sit in the row y from 3 to 4. The rows $y = 0$ to 3 are empty air. The robot drives along the line $y = 0$ with its camera pointing straight at the wall, and takes photo A at $x = 2$, photo B at $x = 3$ and photo C at $x = 4$.

The camera is the pinhole camera of §14.4, with focal length $f = 6$ pixels. Its four pixels sit at positions $u = -3, -1, +1, +3$ on the image line, measured from the image centre. (A real camera has hundreds of pixels in each direction; four keep the arithmetic small.)



Seen from above, in metres.

Every square is one 1 m voxel. The coloured row is the wall (y from 3 to 4); the grey squares are air.

The three triangles are the camera at the three moments it took a photo: A at $x = 2$, B at $x = 3$, C at $x = 4$, all facing the wall. The shaded fans show what each photo sees: four voxels each.

The colours are the *truth*; the robot does not know them. It must build its own map of this wall.

Figure 14.21. The walk-through's room, drawn from above with coordinates. One wall of six 1 m voxels; the robot's camera takes photos A, B and C from three positions along its path.

Step 1 — predict. The pre-trained segmentation network labels each frame on its own (Figure 14.22). It is mostly right, but makes three mistakes, each for a reason that affects *only one photo*:

- in frame A, a reflection on the cabinet door (v_3) makes it look like glass: *window*;
- in frame C, sun glare on the window (v_4) washes it out: *wall*;
- in frame C, the plain wall at the edge of the picture (v_6) is mistaken for a *door*.

Score so far: 9 of the 12 pixel predictions are right (75%).

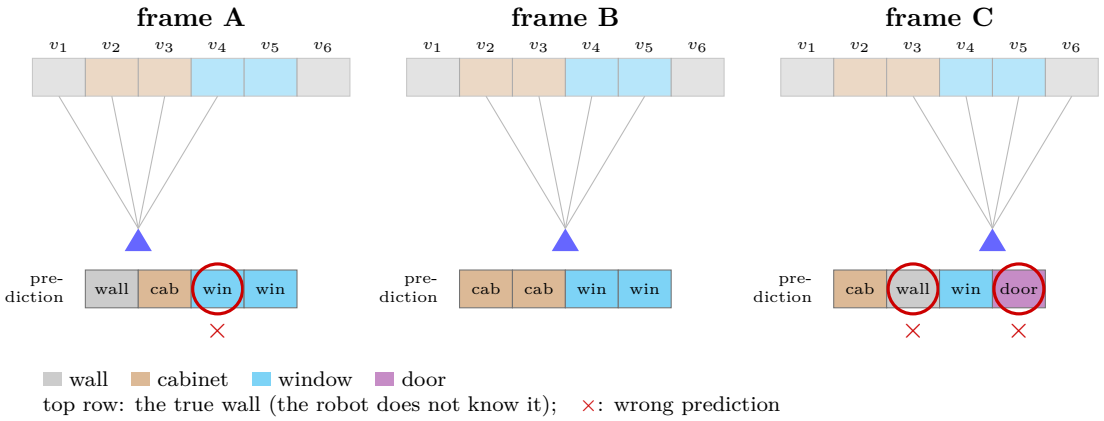


Figure 14.22. Step 1 of the walk-through. Each panel is one photo, seen from above: the camera (blue) looks at four of the six wall voxels, and the network gives each of its four pixels a class. Three predictions are wrong (circled): a reflection in frame A, glare and an edge error in frame C.

How to read it: Follow each grey ray from the camera up to a voxel; the cell at the bottom of that ray is what the network said about that voxel.

Step 2, slowed down. "Fuse the predictions into a map" hides four small calculations: (2a) what each photo actually contains, (2b) where the camera was — the job of SLAM, (2c) which voxel each pixel looked at, and (2d) how a voxel stores its votes. We do each one by hand.

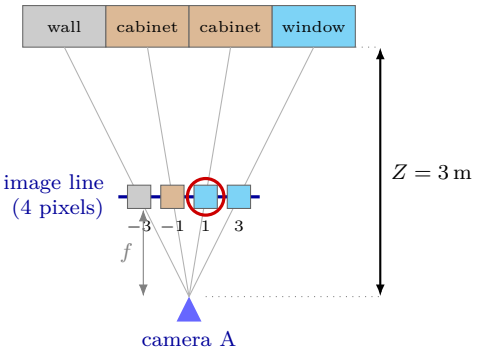
2a — what one frame contains. A frame from an RGB-D camera gives, for each pixel, two things: a *colour* (from which the network predicts a class) and a *depth* Z (measured by the depth sensor: how far away, straight ahead, the surface is). For frame A (Figure 14.23):

pixel position u	-3	-1	$+1$	$+3$
predicted class	wall	cabinet	**window** (wrong)	window
depth Z (m)	3.0	3.0	3.0	3.0

All depths are 3 m because the wall is flat and 3 m straight ahead. Notice what the frame does *not* contain: any idea of where in the room the camera was. A photo alone says "window, 3 m ahead of me, a bit to my right" — not "window at $x = 2.5$ in the room".

2b — what SLAM did: finding where the camera was. To put the three photos into one map, the robot must know each camera's *pose* — here simply its position along the path (all three face the wall). The wheels give a first guess (odometry): they say the robot moved 0.9 m between A and B. But wheels slip, so this is not quite right.

SLAM checks the guess with the camera. Some points on the wall are easy to recognise in both photos — for instance the edge where the plain wall meets the cabinet, at $x = 1$ (the robot does not know this number; it only sees the edge). In photo A this edge appears at



One frame, seen from above. The four small squares on the image line are the four pixels. Each grey line is the pixel's *ray*: the direction it looks in.

Each pixel stores a *colour* (turned into a class by the network — the circled one is wrong) and a *depth* (how far ahead the surface is: 3 m for all four).

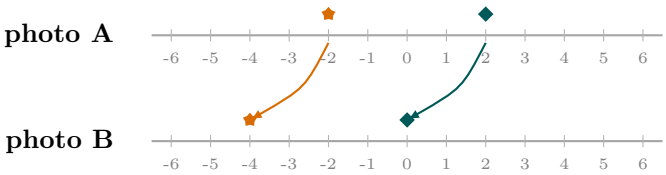
Where exactly a pixel looks is fixed by its position u: pixel u looks u/f metres sideways for every metre forward. So pixel $u = +1$ looks 1/6 m to the right per metre ahead.

Figure 14.23. Step 2a: what one RGB-D frame contains, drawn from above for frame A. Four pixels on the image line, each with a predicted class and a measured depth, and each looking along its own ray.

pixel $u = -2$; in photo B at $u = -4$ (Figure 14.24). The edge seems to have slid 2 pixels to the left, because the camera moved to the right. From Figure 14.33(b), a sideways step t shifts a point at depth Z by $f\,t/Z$ pixels, so

$$t \;=\; \frac{\text{shift} \times Z}{f} \;=\; \frac{2 \times 3}{6} \;=\; 1.0\,\text{m}.$$

A second edge (cabinet to window, at $x = 3$) moves from $u = +2$ to $u = 0$: also 2 pixels, so it agrees. SLAM therefore corrects the odometry guess of 0.9 m to 1.0 m and records the poses A: $x = 2.0$, B: $x = 3.0$, and in the same way C: $x = 4.0$. Real SLAM does this with hundreds of points per image, in 3-D, estimating turns as well as shifts; and whenever it recognises a place seen long ago it corrects the whole path (loop closure, Figure 14.10). But the principle is exactly this: *how far recognisable points slid in the image tells you how far the camera moved.*



Both edges slid 2 pixels left between photo A and photo B.
Shift $= f\,t/Z \Rightarrow t = 2 \times 3/6 = 1.0$ m.

Wheels said 0.9 m (slip); the camera says 1.0 m. SLAM keeps 1.0 m: pose B is $x = 3.0$.

Figure 14.24. Step 2b: how SLAM finds the camera's movement. Two recognisable edges on the wall appear at different pixels in photos A and B. Their shift, together with their depth, gives the distance the camera moved — and corrects the drift of the wheel odometry.

2c — from a pixel to a voxel. Now every pixel can be placed in the room. Take the circled pixel of frame A: $u = +1$, class *window*, depth $Z = 3$ (Figure 14.25).

1. *Relative to the camera*: it looks $u/f = 1/6\,\text{m}$ sideways per metre ahead, and the surface is 3 m ahead, so the point is $x_c = u\,Z/f = 1 \times 3/6 = 0.5\,\text{m}$ to the right of the camera and 3 m in front of it.
2. *In the room*: add the camera's pose from SLAM. Camera A is at $(2.0, 0)$ and faces the wall, so the point is at $(2.0 + 0.5,\ 0 + 3.0) = (2.5,\ 3.0)$.
3. *Which voxel*: with 1 m voxels, the voxel number is the whole-number part of the coordinate: $x = 2.5$ is in column 2 (voxel v_3, which spans $2 \le x < 3$) and $y = 3.0$ is in row 3 (the wall row). So this pixel was looking at v_3.
4. *Vote*: add one *window* vote to v_3's tally.

Doing the same for all twelve pixels: in every frame the four pixels $u = -3, -1, +1, +3$ sit $x_c = -1.5, -0.5, +0.5, +1.5\,\text{m}$ to the side of the camera (all depths are 3 m), so adding each camera's position gives

frame (camera at)	classes predicted	room x of the four points	voxels
A ($x = 2.0$)	wall, cab, **win**, win	0.5, 1.5, 2.5, 3.5	v_1, v_2, v_3, v_4
B ($x = 3.0$)	cab, cab, win, win	1.5, 2.5, 3.5, 4.5	v_2, v_3, v_4, v_5
C ($x = 4.0$)	cab, **wall**, win, **door**	2.5, 3.5, 4.5, 5.5	v_3, v_4, v_5, v_6

Notice the key point: *different pixels of different photos land in the same voxel.* Pixel $u = +1$ of A, pixel $u = -1$ of B and pixel $u = -3$ of C all end up in v_3. That is what makes voting possible — and it only works because SLAM knew the camera positions. With the wrong pose (odometry's 2.9 instead of 3.0 for B), B's pixels would land 0.1 m off, and with small voxels they would vote in the wrong cells.

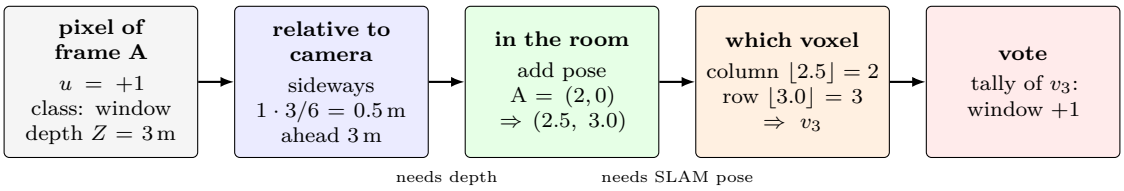


Figure 14.25. Step 2c: the journey of one pixel, from the image to a vote in the map. Depth turns the pixel into a point relative to the camera; the SLAM pose turns it into a point in the room; rounding down turns it into a voxel.

2d — every pixel votes for its voxel. The robot now uses what it knows about each photo: its *pose* (from SLAM) and each pixel's *depth*. Together they say exactly which voxel each pixel was looking at — the grey rays in Figure 14.22. Each pixel adds one vote for its predicted class to that voxel's tally:

voxel	votes received	winner	correct?
v_1	wall (A)	wall	yes
v_2	cabinet (A), cabinet (B)	cabinet	yes
v_3	**window (A)**, cabinet (B), cabinet (C)	cabinet, 2 to 1	yes — A's error out-voted
v_4	window (A), window (B), **wall (C)**	window, 2 to 1	yes — C's error out-voted
v_5	window (B), window (C)	window	yes
v_6	**door (C)**	door	*no* — only one photo saw it

How a voxel stores its label. So what *is* a "labelled 3-D map" inside the computer? Nothing more than a list: for each voxel, one counter per class. The label of a voxel is simply the class with the highest counter right now. Because photos arrive one after another, the label can change as evidence accumulates. Follow v_3 and v_4 as the three photos come in (Figure 14.26):

- after photo A: v_3 has one vote, *window* — so for the moment the map is *wrong* about v_3;
- after photo B: v_3 has window 1, cabinet 1 — a tie;
- after photo C: v_3 has cabinet 2, window 1 — the map now says *cabinet*, and stays right.

The map is not right from the start; it becomes right as more views arrive.

Hard votes versus soft votes. Real systems do not throw away the network's confidence. Instead of adding "1 vote for the top class", they add the network's whole probability vector: a "soft" vote. For v_3, suppose the three frames said

	wall	cabinet	window
A (reflection)	0.1	0.3	0.6
B	0.05	0.85	0.10
C	0.1	0.7	0.2
sum	0.25	**1.85**	0.9

Cabinet wins by an even larger margin, because B and C were *confident* and A was only moderately sure of its mistake. (Probabilistic mapping systems *multiply* the probabilities instead — equivalently, they add their logarithms — which is the correct way to combine independent views. The products here are 0.0005, 0.179 and 0.012: the same winner.) Soft votes also give each voxel a measure of *uncertainty* (how close the top two are), which is what the curiosity planner of this section drives towards.

Step 3 — render: read the map back into each photo. For each frame, send each pixel's ray back into the map and copy the winning class of the voxel it hits (Figure 14.28). The result is one *pseudo-label* image per frame:

- frame A: wall, cabinet, *cabinet*, window — the reflection error is gone;

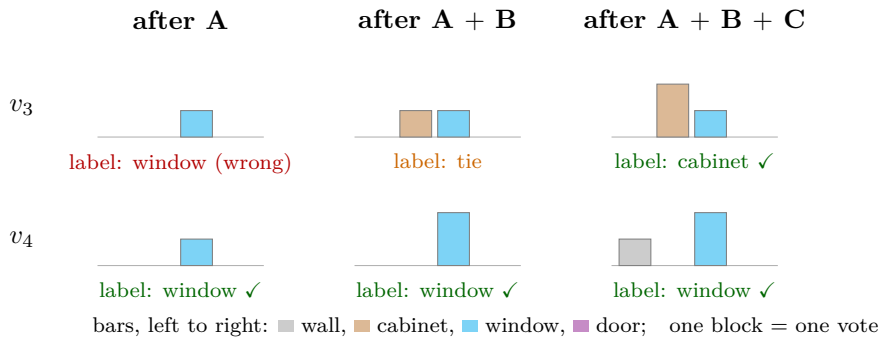


Figure 14.26. Inside the map: each voxel is just a row of counters, one per class, and its label is the largest counter. v_3 starts out wrong (photo A's reflection) and is corrected as photos B and C arrive; v_4's single wrong vote (photo C's glare) never overtakes the two correct ones.

- frame B: cabinet, cabinet, window, window — unchanged, it was already right;
- frame C: cabinet, *window*, window, door — the glare error is gone; the door error stays.

Score: 11 of 12 pseudo-label pixels are right (92%), against 9 of 12 (75%) for the network's own predictions. *The labels are now better than the network that produced them*, and no person was involved.

Ray casting one pixel, by hand. How exactly does "read the map back into the photo" work? Take frame A again, and the pixel $u = +1$ whose own prediction was the wrong *window*. We ask the map, not the network, what this pixel shows (Figure 14.27):

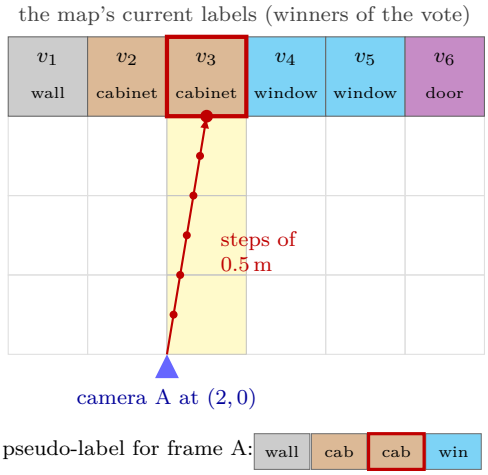
1. *Start at the camera*, at A's pose $(2.0, 0)$, and point along the pixel's ray: 1/6 m to the right for every metre forward.
2. *Walk along the ray in small steps*, here 0.5 m forward at a time, and at each step ask "which voxel am I in, and is it filled?"

forward	position (x, y)	voxel (column, row)	in the map?
0.5	(2.08, 0.5)	$(2, 0)$	empty air — keep going
1.0	(2.17, 1.0)	$(2, 1)$	empty air — keep going
1.5	(2.25, 1.5)	$(2, 1)$	empty air — keep going
2.0	(2.33, 2.0)	$(2, 2)$	empty air — keep going
2.5	(2.42, 2.5)	$(2, 2)$	empty air — keep going
3.0	(2.50, 3.0)	$(2, 3) = v_3$	**filled: stop**

3. *Read the label* of the voxel where the ray stopped: v_3's counters say cabinet 2, window 1, so the label is **cabinet**.
4. *Write it into the pixel*: frame A's pixel $u = +1$ gets the pseudo-label *cabinet*. (As a by-product, the distance walked, 3 m, is the depth the map predicts for this pixel.)

Repeating this for the other three pixels of A gives v_1, v_2, v_4, and the whole pseudo-label image of A: wall, cabinet, *cabinet*, window. Two details matter in practice. A ray that leaves the

map without hitting any filled voxel gets *no* label — those are the black holes in the middle picture of Figure 14.20, and they are simply left out of the training loss. And because the ray stops at the *first* filled voxel, anything hidden behind it (occluded) is correctly ignored.



Ray casting, from above. The red ray belongs to pixel $u = +1$ of frame A. Each dot is one step of 0.5 m. The yellow squares are the voxels the ray passed through: empty air, so it keeps going.

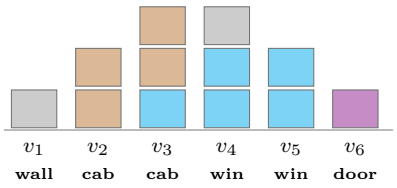
The ray stops in the first *filled* voxel, v_3 (red outline). Its label, cabinet, is copied into the pixel — the red box in the strip below.

The network's own prediction for this pixel was *window*; the map overrules it.

Figure 14.27. Step 3 in detail: ray casting the map into one pixel. Starting at the camera, the ray is followed in small steps through empty voxels until it reaches the first filled one; that voxel's current label becomes the pixel's pseudo-label.

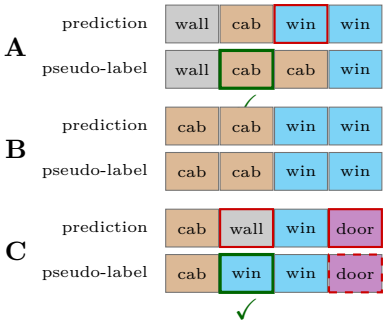
How to read it: Follow the red dots upward: three empty yellow cells, then the cabinet voxel. Compare the red box in the bottom strip with frame A's circled prediction in Figure 14.22.

step 2: the tally in each voxel



Each small square is one vote from one pixel of one photo. The majority colour wins. v_3 and v_4 each have one wrong vote, which loses 2 to 1. v_6 has a single vote, so its error wins by default.

step 3: render and compare



green: a wrong prediction that the map corrected — here the network will learn something.

red dashed: a wrong prediction the map could not fix (only one view).

predictions right: 9/12
pseudo-labels right: **11/12**

Figure 14.28. Steps 2 and 3 of the walk-through. Left: each voxel collects one vote per photo that saw it, and the majority wins. Right: rendering the map back into each photo gives a pseudo-label (lower strip) that fixes two of the three mistakes in the network's own prediction (upper strip).

How to read it: Find voxel v_3 on the left (two brown votes beat one blue), then find frame A's third pixel on the right: its prediction was "window", its pseudo-label is "cabinet".

Step 4 — train: learn where prediction and pseudo-label differ. The network is now fine-tuned with the ordinary per-pixel cross-entropy loss (Chapter 2), with the rendered pseudo-labels as targets. Most pixels already agree with their pseudo-label, so they produce little loss. The learning happens at the two green pixels. Take frame A, pixel v_3: suppose the network's output there was

$$p(\text{wall}) = 0.1, \qquad p(\text{cabinet}) = 0.3, \qquad p(\text{window}) = 0.6.$$

The pseudo-label is *cabinet*, so the loss for this pixel is $-\log 0.3 \approx 1.20$ — large. A gradient step raises $p(\text{cabinet})$ for this kind of reflective cabinet door; if it rises to 0.6, the loss falls to $-\log 0.6 \approx 0.51$. The same happens for the glared window pixel of frame C. The network is being taught *not to be fooled by reflections and glare*, by photos in which the same objects were seen clearly. Each training batch also contains some old labelled ScanNet images from the replay buffer (next subsection), so that nothing it knew before is forgotten.

Step 5 — repeat. In the next round the improved network makes fewer single-photo mistakes, so the votes are cleaner, so the pseudo-labels are better, and so on. And the robot can drive back to look at v_6 from new angles: with three views instead of one, its "door" error would be out-voted too. That is exactly what curiosity-driven planning does (§14.3, "Where to look").

Key takeaway — what the walk-through shows

The map wins because *the same voxel is seen in several photos*, and a mistake caused by one photo's reflection, glare or bad angle rarely happens in the others. Voting in 3-D turns three unreliable opinions into one reliable one; rendering hands that reliable opinion back to every photo as its label. Two limits are visible in the same example: a voxel seen only once (v_6) cannot be corrected, and a mistake that *every* photo made would win the vote and be taught back.

In plain words — try it interactively

Two companion web pages let you run this section's loop yourself.

- **Robot Room Lab** — a virtual RGB-D camera photographs a 3-D room from five poses; you can click any pixel to follow it into the voxel map, watch SLAM match corners and correct the wheels' drift, click voxels to see their vote counts, and cast rays back into each photo. Every photo, depth, pose, vote and ray is computed; only the network is imitated.
  https://claude.ai/artifact/HnzLJc3R3E2gipnFxGptqC
- **The Six-Voxel Room** — the tiny six-voxel example of this section, animated step by step in 3-D.
  https://claude.ai/artifact/FwaDVF8JxGtkdnYy3mehFz

Add `#step3` (or any step number) to the Robot Room Lab link to open that step directly. (Both pages are private until shared from their Share menu.)

14.3.7 Not forgetting: the replay buffer

Training on a stream of new data has a well-known danger. A network fine-tuned only on today's rooms gradually loses what it knew about everything else, and may even get worse on rooms it used to handle well. This is called **catastrophic forgetting**. The simplest remedy is a **replay buffer** $\mathcal{M}$: a store of *old* labelled examples (from the original ScanNet training data) kept alongside the new pseudo-labelled ones. Every training batch mixes the two, so the network is reminded of the old task while it learns the new one (Figure 14.29). In the follow-up work discussed below, the replay buffer holds ScanNet images amounting to 30% of the new training set (Zurbrügg et al., 2022).

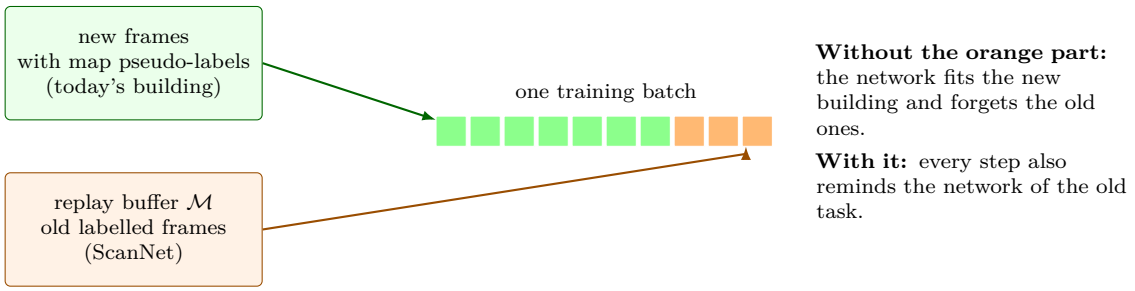


Figure 14.29. Experience replay. Each training batch mixes new pseudo-labelled frames (green) with old labelled frames from a stored buffer (orange).

14.3.8 Where to look: uncertainty and planning

So far the robot's route was given. But the map can also tell the robot *where to go next*. Along with the class tally, the map can store how *uncertain* the network was about each voxel (the lecture's slide 12 shows the mapping interface: depth, colour, and an uncertainty image, next to a 3-D map coloured by uncertainty). A robot that drives towards the uncertain parts collects exactly the views that the network gets wrong — the views that carry the most new information.

This idea was tested by the same group in a follow-up paper, Zurbrügg et al. (2022), which reuses Frey et al.'s map-rendered pseudo-labels and compares ways of choosing where to drive. The main ones:

- *Random*: pick the next viewpoint at random.
- *Exploration*: try to see the whole room (cover every place once).
- *Curiosity*: see the whole room, but prefer viewpoints where the network is uncertain.

In their living-room/kitchen simulation, the pre-trained network starts at 32.1 mIoU. After three rounds of "collect 150 images, then retrain", random viewpoints *lower* it to 27.4,

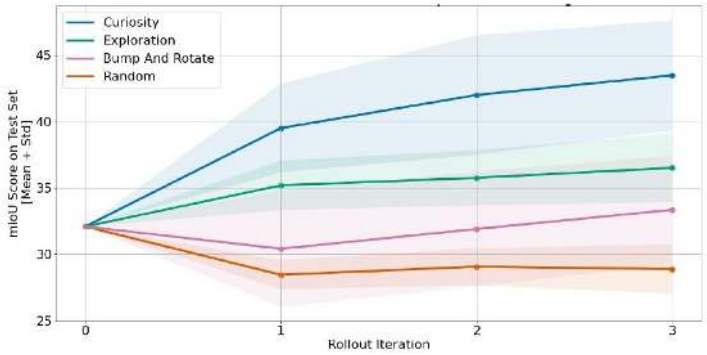


Figure 14.30. Planning significantly influences adaptation (Lecture 09, slide 14; experiment of Zurbrügg et al., 2022). The vertical axis is mIoU on a held-out test set, the horizontal axis the number of collect-and-retrain rounds; shaded bands show the spread over repeated runs. All planners start from the same network (about 32). Curiosity keeps improving; exploration improves and then flattens; random viewpoints make the network worse.

How to read it: Look at where each line ends, not where it starts: the same learning algorithm, fed data chosen in different ways, ends up 15 points apart.

exploration raises it to 34.9, and curiosity raises it to 42.2 — a 10-point gain, and nearly 15 points more than random (Zurbrügg et al., 2022, Table I). The lecture's plot (slide 14, Figure 14.30) shows the same ranking round by round, with one more scripted baseline, "bump and rotate".

Why can random driving make things *worse*? A random route sees a few places many times and misses others. The map is then built from few, similar views, so its vote is weak and its errors are not out-voted; the network is retrained on those errors, and on an unbalanced mix of places. The lesson of slide 13 ("Planning significantly influences adaptation"): *what data the robot chooses to collect matters as much as how it learns from it.*

Results. Adapting with map pseudo-labels raises segmentation accuracy by 9.9% on average over the fixed pre-trained network, both on ScanNet scenes and on the authors' own RGB-D recordings, while keeping what was learned in pre-training (Frey et al., 2022).

Paper Analysis: Frey et al. (2022) and Zurbrügg et al. (2022)

Papers: Continual Adaptation of Semantic Segmentation using Complementary 2D-3D Data Representations (RA-L 2022); Embodied Active Domain Adaptation for Semantic Segmentation via Informative Path Planning (RA-L 2022). **Problem:** a segmentation network fails in a new environment. **Insight:** a 3-D map that fuses many frames is a cleaner label than any single frame. **Method:** fuse predictions into a voxel map with SLAM poses; render pseudo-labels; fine-tune with a replay buffer. The follow-up chooses where to drive by the network's uncertainty. **Results:** +9.9% accuracy on average; curiosity planning reaches 42.2 mIoU versus 27.4 for random viewpoints. **Limitations:** relies on good poses; cannot fix a mistake that all views share. **What to remember:** the map supervises the network, and the map's uncertainty tells the robot where to look next — supervision in a closed loop.

14.3.9 What can go wrong

Failure Modes

Mistakes that all views share. The vote only removes mistakes that some views make and others do not. If the network calls a glass door a window from *every* angle, the map says "window" confidently and the network is taught its own mistake. **Bad poses.** If SLAM places a frame in the wrong spot, its opinions land in the wrong voxels and the map is smeared; the rendered labels are then confidently wrong. **Early winners.** A wrong class that wins a voxel's tally early is trained into the network, which then predicts it again and adds more votes for it. **Forgetting versus stalling.** Too little old data in the batch and the network forgets; too much and it barely adapts. **Curiosity's side effect.** A robot sent to uncertain places may over-collect its hardest cases (shiny surfaces, clutter) and unbalance the training mix.

14.4 Geometry as Supervision: Depth, Camera Motion and Flow

14.4.1 The whole idea first, as a story (almost no maths)

Start with yourself. Close one eye and look at a room: it looks flat, and you cannot tell how far away the chair is. Now move your head a little to the side. The chair slides across your view a lot, the far wall barely moves. Your brain uses exactly this: *near things slide more, far things slide less.* If you know how far you moved your head and how far the chair slid, you know how far away it is. If you know how far away things are and how they slid, you know how far you moved. Each fact pins down the others.

Now the network, in four moments (Figure 14.31). Three small networks are trained together from ordinary video, with nobody labelling anything: one guesses *how far away* each pixel is (depth), one guesses *how the camera moved* between two frames (pose), one guesses *where each pixel slid to* (flow).

1. **Two photos, one small step apart.** The camera takes a photo, moves a little, takes another. A near tree jumps a lot between them; a far house hardly moves.
2. **Guess.** The depth network guesses "the tree is 2 m away, the house 10 m"; the pose network guesses "the camera stepped 0.5 m to the right".
3. **Predict the slide.** From those two guesses, the robot can work out where every pixel *should* be in the second photo — the tree far to the left, the house only a little.
4. **Check against the real photo.** It rebuilds the first photo out of the second photo's pixels, fetched from those predicted places. If the guesses were right, the rebuilt picture looks just like the real one. If not, the mismatch shows exactly where, and all three networks are nudged to fix it. Pixels that no rigid movement can explain — a car that drove away, a spot hidden behind the tree — are simply left out.

After many thousands of frame pairs, the only way to always pass the check is to guess depth, motion and flow correctly (up to one overall scale).

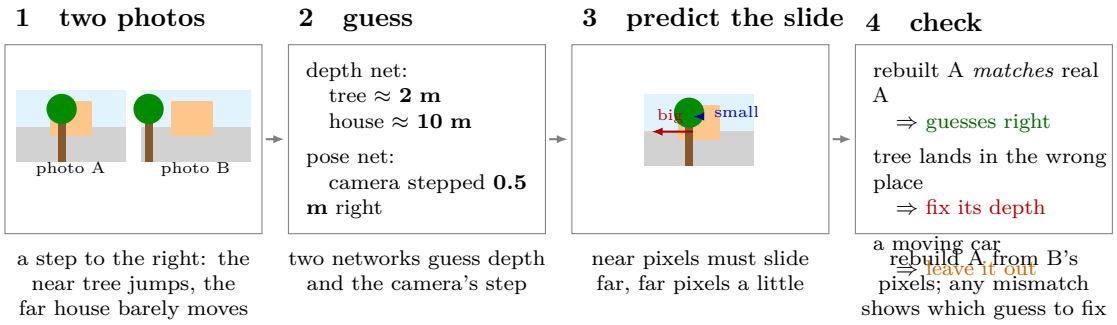


Figure 14.31. DF-Net in four pictures. (1) Two photos a small step apart. (2) The depth and pose networks make guesses. (3) From them, the robot predicts where every pixel should slide. (4) It rebuilds the first photo from the second and checks: mismatches say which guess to fix; pixels no rigid motion can explain are ignored.

How to read it: Compare the tree in the two small photos of picture 1 with the house: that difference in sliding is the whole signal.

In plain words — the technical words, translated

depth	how far away the thing in a pixel is
pose, ego-motion	how the camera moved between the two photos (turn and shift)
optical flow	for each pixel, how far and which way it slid between the photos
parallax	near things slide more than far things when you move
warping	building one photo out of the other's pixels, fetched from the predicted places
photometric loss	how different the rebuilt photo looks from the real one
cross-task consistency	the flow network and "depth + pose" must agree
forward–backward check	going A→B and back B→A should return you to the start
valid mask	the list of pixels the checks are allowed to use (moving or hidden ones are left out)
scale ambiguity	a toy house near by and a real house far away look the same, so absolute size cannot be learned from one camera

Everyday analogy — three witnesses of one car trip

One says how far away the buildings were, one says how fast the car went, one says how fast the view changed. Nobody knows who is right, but their stories must fit together; whenever they don't, someone is corrected. *Where it breaks:* if one witness doubles every distance and another doubles the speed, the stories still fit — that is the scale ambiguity.

14.4.2 From the story to the formulas: why each piece of maths is needed

Question 1 (moment 3): "How far should this pixel slide?" For a sideways camera step t, a point at depth Z slides by

$$\text{slide (pixels)} \;=\; \frac{f \times t}{Z} \qquad (f = \text{the camera's focal length in pixels}).$$

Why this form: dividing by Z is "near slides more, far slides less"; multiplying by t is "a bigger step, a bigger slide". *Numbers* ($f = 500$, $t = 0.5$ m): a tree at 2 m slides 125 px, a house at 8 m only 31 px (Figure 14.32). The general formula, Eq. (14.2), is the same idea for any camera turn and shift.

Question 2 (moment 4): "Does the rebuilt photo look like the real one?" For each pixel, compare its real colour with the colour fetched from where it should have slid to:

$$\mathcal{L}_{\text{photo}} \;=\; \text{average over valid pixels of } \big|\,\text{real colour} - \text{fetched colour}\,\big|.$$

Why needed: it is the only term that looks at the actual pictures; without it the three networks could agree on any nonsense.

Question 3: "Do the flow network and depth + pose agree?"

$$\mathcal{L}_{\text{cross}} \;=\; \text{average of } \big|\,\text{flow network's slide} - \text{slide computed from depth and pose}\,\big|.$$

Numbers (worked example of §14.4): depth + pose say the pixel lands at $(162.5, 20)$, the flow network says $(150, 18)$: they disagree by about 12.7 pixels, so both sides are nudged.

Question 4: "Is the round trip consistent?" Going from photo A to B and back from B to A should return every pixel to where it started; where it does not, the pixel is probably hidden in one photo or moving. That is $\mathcal{L}_{\text{fb}}$, and the same check produces the *valid mask*.

Question 5: "What about a blank white wall?" Every depth looks equally good there, so a small extra term, $\mathcal{L}_{\text{smooth}}$, prefers neighbouring pixels to have similar depth. *Why needed:* it fills in where the pictures give no clue.

Putting it together.

$$\mathcal{L} = \mathcal{L}_{\text{photo}} + \lambda_s \mathcal{L}_{\text{smooth}} + \lambda_f \mathcal{L}_{\text{fb}} + \lambda_c \mathcal{L}_{\text{cross}},$$

where the λ's just say how much each check counts.

Question 6: "Can it learn real distances?" No: doubling every depth *and* the step gives $\frac{f \cdot 2t}{2Z} = \frac{ft}{Z}$ — the same slide, so every check above passes equally. Real distances need an extra clue (a second camera, wheel odometry, an object of known size).

14.4.3 The details

This section needs a little camera geometry. None of it is hard, but all of it is needed, so we build it up from one picture.

14.4.4 Background: how a camera turns 3-D into 2-D

The pinhole camera. Think of a camera as a dark box with a tiny hole. Light from a point in the world travels in a straight line through the hole and lands on the back wall, the

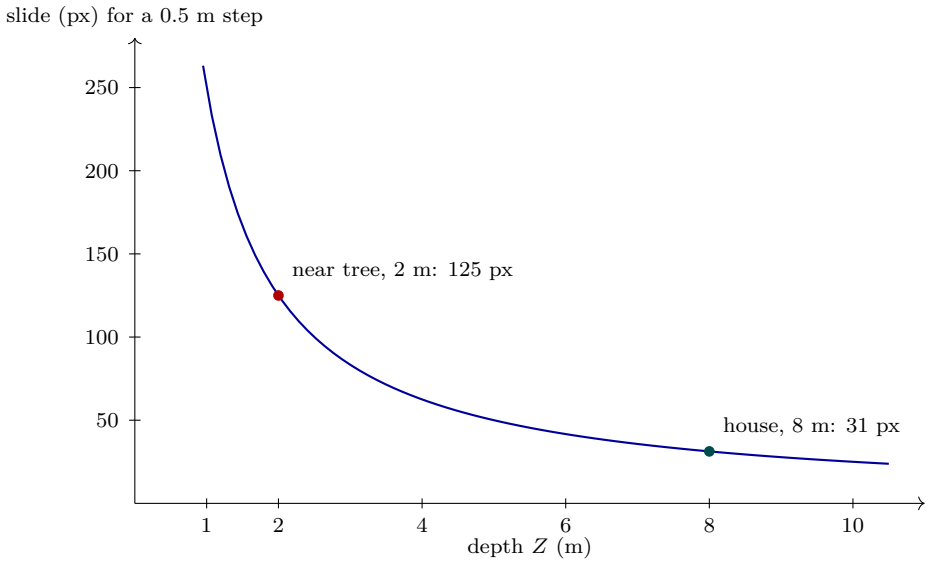


Figure 14.32. The whole of Question 1 in one curve: how far a pixel slides when the camera steps 0.5 m sideways ($f = 500$ px), as a function of its depth. Near things slide a lot, far things little; reading the curve backwards gives depth from slide.

Table 14.4. DF-Net: the story and the maths, side by side.

In the story	In the maths	Why it is needed
"near things slide more"	slide $= ft/Z$	link depth, step and slide
"rebuild A from B and compare"	$\mathcal{L}_{\text{photo}}$	tie the guesses to the real pictures
"the flow net and depth + pose must agree"	$\mathcal{L}_{\text{cross}}$	the triangle's check
"there and back again"	$\mathcal{L}_{\text{fb}}$, valid mask	find hidden or moving pixels
"blank walls: assume smooth"	$\mathcal{L}_{\text{smooth}}$	fill in where pictures give no clue
"double everything, nothing changes"	$f\,(2t)/(2Z) = ft/Z$	why absolute scale is unknown

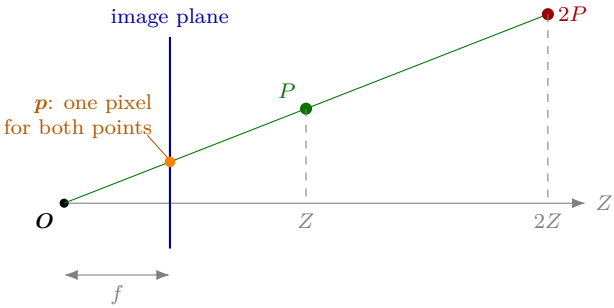
image plane. (Drawings usually put the image plane in front of the hole instead; this flips nothing important and is easier to draw.) The hole is the *camera centre* $\boldsymbol{O}$; the line straight out of the camera is the *optical axis*; the distance from the hole to the image plane is the *focal length* f, measured in pixels.

Put the camera at the origin, looking along the Z axis. A point $P = (X, Y, Z)$ in front of it, at **depth** Z (its distance along the optical axis), lands on the image at

$$\boldsymbol{p} \;=\; \pi(P) \;=\; \begin{pmatrix} f\,X/Z + c_x \\ f\,Y/Z + c_y \end{pmatrix}. \tag{14.1}$$

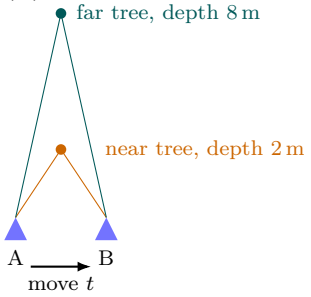
Here π ("project") is the name of this map and (c_x, c_y) is the pixel where the optical axis meets the image, usually near the image centre. (Real cameras can have slightly different focal lengths f_x, f_y for the two directions; we use one f to keep things simple.) The one thing to notice is the *division by* Z: things further away look smaller and sit closer to the centre.

(a) projection divides by depth



P and the point $2P$, twice as far and twice as high, sit on the same ray, so they land on the same pixel: $f\,Y/Z = f\,(2Y)/(2Z)$. **A single photo keeps the direction of each point but loses its distance.**

(b) step sideways: near shifts more



The camera steps $t = 0.5\,\text{m}$ to the right. For a sideways step the image shift of a point is

$$\Delta x = \frac{f\,t}{Z}.$$

With $f = 500\,\text{px}$: the near tree shifts $500 \cdot 0.5/2 = 125\,\text{px}$, the far tree only $500 \cdot 0.5/8 \approx 31\,\text{px}$. **How far each pixel moves tells you how far away it is.**

Figure 14.33. Two facts about cameras. (a) Projection divides by depth, so one image cannot tell a small near object from a big far one. (b) When the camera moves, near points move across the image more than far points (*parallax*) — which is why, looking out of a train window, the fence rushes past while the hills barely move.

How to read it: In (b), compare the angle between the two orange rays with the angle between the two teal rays. A bigger angle is a bigger shift in the image.

Depth map. A *depth map* is an image in which each pixel stores the depth of whatever that pixel sees, usually drawn with colours (yellow near, purple far, in the lecture's figures). A *depth network* predicts a depth map from a single colour image.

Camera motion. Between two video frames the camera moves: it *turns* (a rotation R, a 3×3 matrix) and *shifts* (a translation $\boldsymbol{t}$, a 3-vector). Together, $(R, \boldsymbol{t})$ is the *relative pose* of the two frames, also called the camera's **ego-motion**. A *pose network* takes two frames and predicts $(R, \boldsymbol{t})$.

Optical flow. **Optical flow** is, for every pixel of frame A, a small 2-D arrow that says where that pixel's content appears in frame B. A *flow network* takes two frames and predicts this arrow field. Lecture 09 (slides 16–17) shows it on a drawn scene: the camera moves to the right, so every arrow points left, and the arrows on the near tree and the near grass are much longer than those on the far tree (Figure 14.34). That is Figure 14.33(b) for every pixel at once.

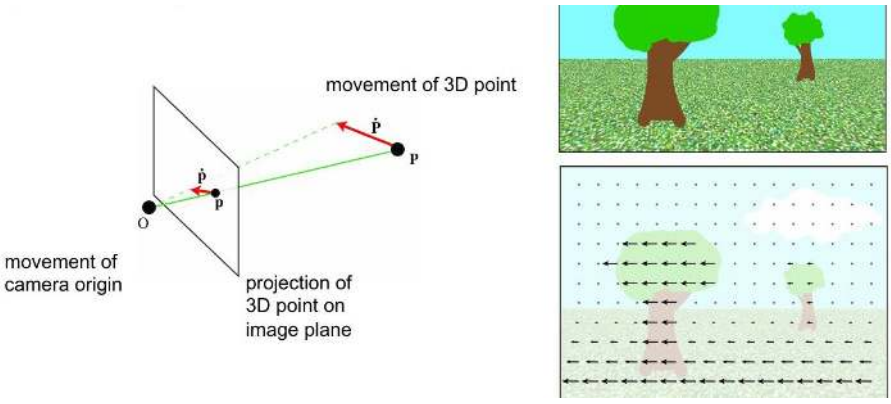


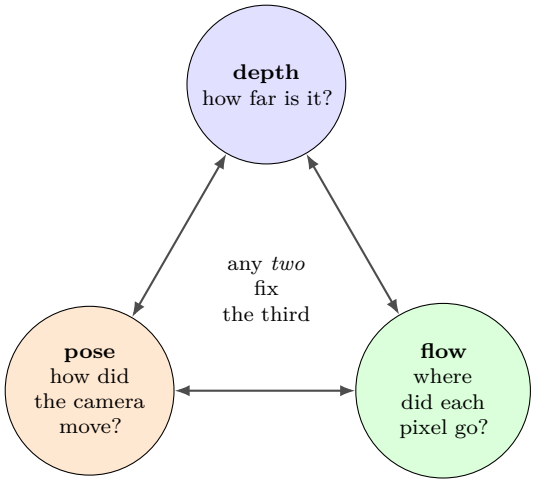
Figure 14.34. Optical flow as projected 3-D motion (Lecture 09, slide 16, after Torralba, Isola & Freeman, *Foundations of Computer Vision*). Left: a scene point P moves by $\dot{P}$ relative to the camera; its image $\boldsymbol{p}$ on the image plane moves by $\dot{\boldsymbol{p}}$ — that 2-D motion is the flow. Right: a scene, and its flow when the camera moves to the right: all arrows point left, and near things (the big tree, the grass at the bottom) move more than far things (the small tree, the cloud).

Occlusion. Some pixels of frame A have no partner in frame B: the thing they show has become hidden behind something else, or has left the picture. Such pixels are called *occluded*. No arrow can be right for them, so any loss that compares A with B must skip them.

14.4.5 The idea: three quantities that must agree

Look again at the formula in Figure 14.33(b): $\Delta x = f\,t/Z$. It ties together three things: the pixel's *depth* Z, the camera's *motion* t, and the pixel's *flow* Δx. Know two, and the third is fixed. This is true for every pixel of every video ever filmed of a still scene, because it is

just geometry. That observation is the key idea of DF-Net (Zou et al., 2018): train a depth network, a pose network and a flow network *together*, and use the fact that they must agree as the training signal. Lecture 09 (slide 18) puts it as: *learn each by aligning to the other two.*



Each circle is predicted by its own network, and none has a ground-truth label. The training signal is **disagreement**: take the predicted depth and the predicted camera motion, compute what the flow *must* be, and penalise the flow network for differing — and use the same comparison to correct depth and pose.

Why this is free. The rule is geometry, not an annotation. It holds for every video of a still scene, so any such video is training data.

Where it breaks. The rule assumes the *scene* stands still and only the camera moves. Moving cars, occlusions, reflections and blank walls break it, which is why these systems carry masks saying "do not trust this pixel".

Figure 14.35. The depth–pose–flow triangle. Depth, camera motion and optical flow are three descriptions of one 3-D motion; fixing any two pins down the third, so three unlabelled networks can supervise each other.

How to read it: Pick any two circles; the arrow to the third is a loss.

In plain words — three witnesses to one event

Three witnesses describe the same car trip: one says how far away the buildings were, one says how the car moved, one says how the view out of the window changed. None of them is known to be right. But their stories must fit together — if the buildings were close and the car moved fast, the view must have changed a lot. Whenever the stories do not fit, at least one witness is wrong, and each is nudged towards a story that fits. After thousands of trips, the only way to always fit is for all three to tell the truth.

Model at a glance: DF-Net — depth and flow from consistency Zou, Luo & Huang, 2018

The free label	Camera geometry. Depth, camera motion and optical flow must agree, so each network's output is a target for the others.
Sensors	A single ordinary video camera. Nothing else.
Networks	A depth network (one image in, depth map out), a pose network (two frames in, $(R, \boldsymbol{t})$ out) and a flow network (two frames in, arrow field out).
Losses	Photometric (rebuild one frame from the other), smoothness, forward–backward consistency, and cross-task consistency (flow agrees with depth + pose).
Escape hatch	Masks that switch the losses off where the geometry cannot hold: moving objects, occlusions, pixels that leave the image.

What it buys	Depth and flow networks trained without a depth sensor and without a single hand-labelled correspondence.

14.4.6 How it works, step by step

Moving one pixel from frame A to frame B. Here is how depth and camera motion together predict where a pixel must go (Figure 14.36). Take pixel $\boldsymbol{p}_A$ in frame A.

1. *Lift it to 3-D.* The depth network says how far away it is, $D_A(\boldsymbol{p}_A)$. Walking that far along the pixel's ray gives the 3-D point P.
2. *Move the point into camera B's view.* The pose network says how the camera moved; in camera B's coordinates the point is at $RP + \boldsymbol{t}$.
3. *Project it into frame B* with Eq. (14.1): this is the pixel $\boldsymbol{p}_B^{\text{rigid}}$ where the point *should* appear if the scene is rigid (nothing in it moves).
4. *Compare.* The arrow from $\boldsymbol{p}_A$ to $\boldsymbol{p}_B^{\text{rigid}}$ is the *rigid flow* — the flow implied by depth and pose. It can be compared with two things: the actual colour of frame B at that spot, and the flow network's own arrow.

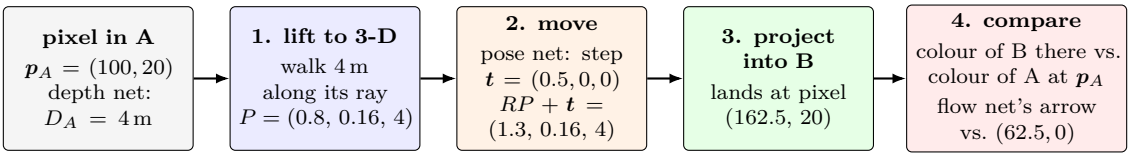


Figure 14.36. How depth and camera motion predict where a pixel goes, with the numbers of the worked example below ($f = 500\,\text{px}$, image centre at $(0, 0)$). Step 4 gives two losses: a *photometric* one (do the colours match?) and a *cross-task* one (does the flow network agree?).

The same steps as one formula. Writing the pixel in *homogeneous coordinates* — just the pixel with a 1 appended, $\tilde{\boldsymbol{p}}_A = (p_x, p_y, 1)^\top$ — step 1 is a matrix product: the *intrinsics matrix* K stores f and (c_x, c_y), and $K^{-1}\tilde{\boldsymbol{p}}_A$ is the pixel's ray direction scaled so that its depth component is 1; multiplying by the depth walks along the ray. Steps 1–3 together read

$$\boldsymbol{p}_B^{\text{rigid}} \;=\; \pi\Big(\underbrace{R\;\overbrace{D_A(\boldsymbol{p}_A)\,K^{-1}\tilde{\boldsymbol{p}}_A}^{\text{step 1: the 3-D point } P}\;+\boldsymbol{t}}_{\text{step 2: in camera B's view}}\Big), \tag{14.2}$$

with π (step 3) from Eq. (14.1). This equation *synthesises flow from depth and pose*: two corners of the triangle produce the third.

Worked Example — One consistency check with numbers

Take a camera with $f = 500\,\text{px}$ and image centre at $(0, 0)$, so $K^{-1}(p_x, p_y, 1)^\top = (p_x/f,\, p_y/f,\, 1)^\top$. In frame A the depth network predicts $D_A = 4\,\text{m}$ at pixel $\boldsymbol{p}_A = (100, 20)$.

Figure 14.37. Flow from depth and pose (Lecture 09, slide 19). A 3-D reconstruction of a scene with the camera positions of a video drawn along the path. For three frames (boxed), the depth maps are shown on the right (yellow near, purple far), and the coloured lines join pixels that show the same 3-D point. Assuming the scene is static, depth plus camera pose is enough to compute these correspondences.

Step 1. $P = 4 \cdot (100/500,\ 20/500,\ 1)^\top = (0.8,\ 0.16,\ 4)^\top$ m.
Step 2. The pose network predicts no turn ($R = I$) and a sideways step $\boldsymbol{t} = (0.5, 0, 0)^\top$ m. So $RP + \boldsymbol{t} = (1.3,\ 0.16,\ 4)^\top$.
Step 3. Project: $\boldsymbol{p}_B^{\text{rigid}} = 500 \cdot (1.3/4,\ 0.16/4) = (162.5,\ 20)$. Depth and pose together claim: the pixel at $(100, 20)$ moves 62.5 pixels to the right. (Check with the shortcut $\Delta x = ft/Z = 500 \cdot 0.5/4 = 62.5$.)
Step 4. Suppose the flow network says the pixel went to $(150, 18)$ instead. The two disagree by $\|(162.5, 20) - (150, 18)\|_2 = \sqrt{12.5^2 + 2^2} \approx 12.7$ pixels — a large cross-task loss. The colours will probably disagree too, since at most one of the two spots in frame B shows the same thing as $\boldsymbol{p}_A$.
Who is wrong? Maybe the flow network (then it is pulled towards the rigid flow), maybe the depth or the pose (then they are pulled towards agreeing with the flow and the colours). Over many pixels and many frames, only the true geometry keeps all the checks happy. But if this pixel belongs to a car that itself drove forward, *no* depth and pose can explain it — the rigid-scene rule is simply false there. The forward–backward check below fails too, and the *valid mask* removes the pixel from the loss: the system withholds supervision rather than learning something wrong.

14.4.7 The loss, term by term

DF-Net combines four terms (Lecture 09, slide 23):

$$\mathcal{L} = \underbrace{\mathcal{L}_{\text{photo}}}_{\substack{\text{rebuild one frame}\\ \text{from the other}}} + \lambda_s \underbrace{\mathcal{L}_{\text{smooth}}}_{\substack{\text{depth changes}\\ \text{smoothly}}} + \lambda_f \underbrace{\mathcal{L}_{\text{fb}}}_{\substack{\text{A}\to\text{B agrees}\\ \text{with B}\to\text{A}}} + \lambda_c \underbrace{\mathcal{L}_{\text{cross}}}_{\substack{\text{flow agrees with}\\ \text{depth+pose}}}, \tag{14.3}$$

where $\lambda_s, \lambda_f, \lambda_c$ are weights that set how much each term counts (Zou et al., 2018). Each term, in words:

- $\mathcal{L}_{\text{photo}}$ — **photometric ("appearance") loss.** If depth, pose and flow are right, then taking, for every pixel of A, the colour found at its predicted spot in B rebuilds frame A. This rebuilding is called *warping*. The loss is the colour difference between the rebuilt image and the real frame A, over valid pixels. In its simplest form: $\sum_{\boldsymbol{p}} V(\boldsymbol{p})\,|I_A(\boldsymbol{p}) - I_B(\boldsymbol{p}')|$, where $\boldsymbol{p}'$ is the predicted partner of $\boldsymbol{p}$ and $V(\boldsymbol{p}) \in \{0, 1\}$ is the valid mask. This is the term that ties everything to the actual images; without it the three networks could agree on nonsense.
- $\mathcal{L}_{\text{smooth}}$ — **smoothness.** On a blank wall every depth looks equally good to the photometric loss (every pixel has the same colour). The smoothness term fills that gap with a sensible guess: neighbouring pixels should have similar depth. It is usually weakened at image edges, where depth really does jump.
- $\mathcal{L}_{\text{fb}}$ — **forward–backward consistency.** Going from A to B and then back from B to A should return you to where you started. For depth, the lecture writes it as "depth of frame A = relative pose $*$ depth of frame B, and vice versa" (slide 21): move B's 3-D points into A's view with the pose, and they must match A's depth. For flow: follow A's arrow to B, then B's arrow back; you should land on the pixel you started from. Pixels where this round trip fails are almost always occluded or moving, so the same check also *produces the valid masks*.
- $\mathcal{L}_{\text{cross}}$ — **cross-task consistency.** The flow network's arrow should equal the rigid flow from Eq. (14.2) — on valid, static pixels. This is the edge of the triangle that couples the flow network to the depth and pose networks.

Moving objects and masks. The rigid rule is a strong claim, and it is false on anything that moves by itself. DF-Net handles this by *detecting* where the rule fails. Rebuild frame B from frame A using only depth and pose, and compare it with the real frame B (and the other way round): on a static scene they match; on a moving car or a pixel that went out of view, they do not. Lecture 09 (slide 20) shows the result: yellow "invalid" regions on the oncoming cars and along the image borders (Figure 14.39). The system supervises itself *only where its physics is allowed to hold*.

The scale problem. Figure 14.33(a) has a consequence for training. Double every depth *and* the camera step: $P \mapsto 2P$, $\boldsymbol{t} \mapsto 2\boldsymbol{t}$. Then $R(2P) + 2\boldsymbol{t} = 2(RP + \boldsymbol{t})$, and projection divides the factor 2 away. Every pixel lands in exactly the same place, so every loss above is unchanged. A single moving camera can therefore learn depth only *up to one unknown scale*: it cannot tell a toy house one metre away from a real house ten metres away. The absolute scale must come from elsewhere — a second camera a known distance away (stereo), a motion sensor, wheel odometry, or an object of known size.

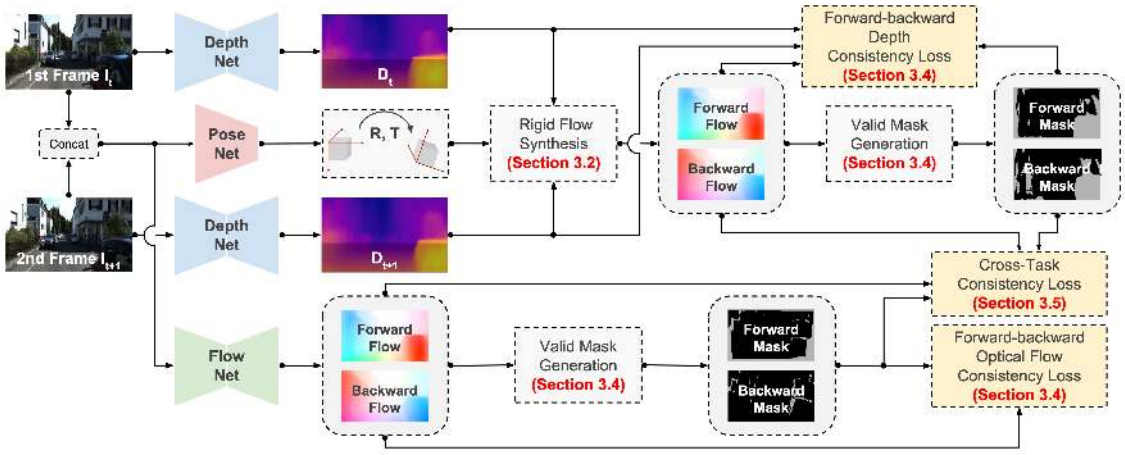


Figure 14.38. The DF-Net architecture (Lecture 09, slide 22; from Zou et al., 2018). Top half: two depth networks (shared weights) give D_t and D_{t+1}, a pose network gives (R, T), and "Rigid Flow Synthesis" (Eq. 14.2) turns them into forward and backward rigid flow. Bottom half: the flow network predicts forward and backward flow directly. "Valid Mask Generation" (forward–backward checks) produces the black-and-white masks. The yellow boxes are the losses: forward–backward depth consistency, cross-task consistency (rigid flow vs. network flow), and forward–backward flow consistency.

How to read it: Find the two flow boxes (top: computed from depth and pose; bottom: predicted by the flow network). The "Cross-Task Consistency Loss" box has arrows from both: that is the triangle's edge.

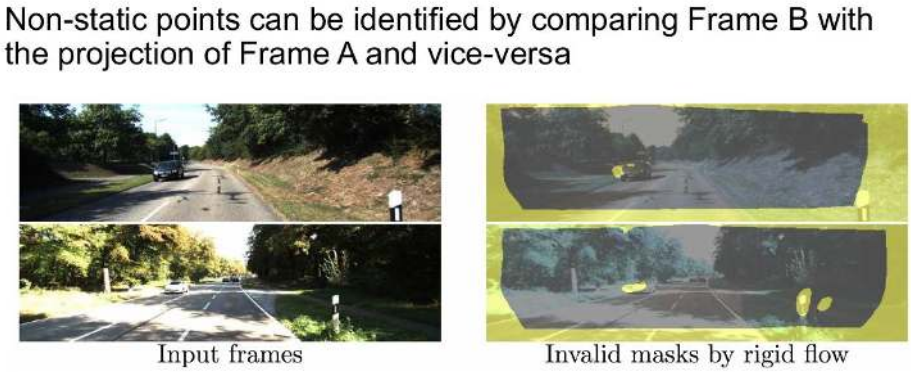


Figure 14.39. Non-static points can be found by comparing frame B with the projection of frame A, and vice versa (Lecture 09, slide 20; from Zou et al., 2018). Left: input frames from a driving video. Right: in yellow, the pixels where depth plus camera motion cannot explain what is seen — the moving cars, and the image borders that leave the view. These pixels are removed from the consistency losses.

Results. Trained on driving videos without any depth labels, DF-Net's depth network produces depth maps from single images that look close to the laser-scanner measurements (Figure 14.40), and training depth and flow together improves both compared with training each alone (Zou et al., 2018).

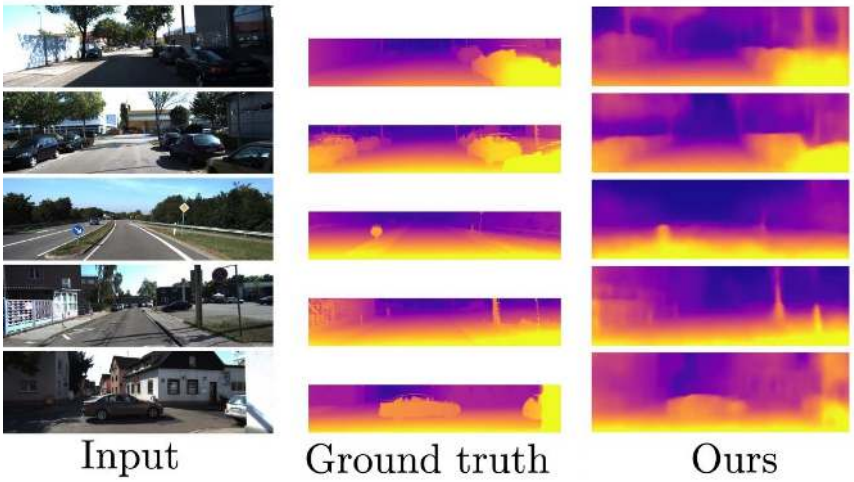


Figure 14.40. Depth from a single image, learned without depth labels (Lecture 09, slide 24; from Zou et al., 2018). Left: input frames from driving videos. Middle: ground truth from a laser scanner, used only for evaluation. Right: DF-Net's prediction. Yellow is near, purple is far.

Simplified intuition vs. more precise view

Simplified: each network supervises the others, so none needs labels. **More precise:** consistency losses only ask for *agreement*, not truth. In principle the three networks could agree on a jointly wrong answer. What prevents this is (1) the photometric term, which ties everything to the actual pixels of real images; (2) the masks, which switch the rules off where they are known to be false; and (3) the sheer number of frames: a wrong answer that fits one pair of frames rarely fits millions. The supervision is not "the triangle knows the truth" but "the only answers that stay consistent across many frames look like the real geometry — up to scale".

In plain words — try it interactively

Depth–Pose–Flow Lab — a camera photographs a street twice; click pixels to see their depth and flow, set the depth and pose guesses yourself and watch the cross-task and photometric losses respond, let a car move to see the valid mask, and scale depth and step together to see the scale ambiguity.
`https://claude.ai/artifact/34NWh78WsjsFGmkBeL1sco`
(Private until shared from the page's Share menu. Add `#step3`, or any step number, to the link to open that step.)

Failure Modes

Scale drift. Because of the scale problem, depth and pose learned from a single camera can slowly rescale the world. **Blank surfaces.** On white walls and sky the photometric term cannot tell depths apart, the smoothness term takes over, and depth becomes a blur. **Masks that are too tight or too loose.** Too tight, and the hard, busy scenes with many moving objects give almost no training signal; too loose, and moving objects teach the networks wrong geometry. **Awkward motion.** When the camera only turns, or moves straight at a distant flat wall, the flow hardly depends on depth, so depth and pose are poorly constrained.

14.5 Own Experience as Supervision: Where Can I Walk?

14.5.1 The whole idea first, as a story (almost no maths)

Before any formula, here is the whole method told as one outing of the robot. If you understand this story, the rest of the section only adds detail.

Start with yourself. Imagine hiking in a forest in new shoes. You look ahead and see a dirt path, dry leaves beside it, a dark muddy patch and a tree. You cannot tell by looking how slippery the mud is. So you walk. On the path your feet feel fine. In the mud you slow down and slide. Next time you see a patch that *looks* like that mud, you already know: slow going. You were never told this by anyone — your feet taught your eyes. That is the entire idea of this section.

Now the robot, in four moments (Figure 14.41).

1. **Look.** The four-legged robot stands at the start of a forest path and takes a photo of the ground ahead. The photo shows the path, leaves beside it, a mud patch on the path, and a tree. At this moment the robot has no idea which of these it can walk on.
2. **Walk and feel.** A person steers the robot forward along the path with a remote control, always asking for the same speed. On the firm path the robot keeps up easily. In the mud its legs slip and it falls behind the requested speed. The robot does not need eyes for this: it simply notices "I was asked to go this fast, and I managed only that fast". Keeping up means *easy*; falling behind means *hard*.
3. **Write it onto the old photo.** The robot remembers where it stood when it took the photo, and where its feet went afterwards. So it can find its footprints *in that earlier photo* and colour them: blue where walking felt easy, red where it felt hard. Now part of the photo has a label — made by the robot's own legs, a few seconds after the photo was taken.
4. **Guess the rest — carefully.** Most of the photo was never walked on: the leaves beside the path, the tree. The robot uses one simple rule. *If a patch looks like ground I*

have walked on, expect it to feel like that ground. If it looks like nothing I have walked on, be careful. The leaves look a lot like the path's edges, so they are guessed easy. The tree looks like nothing it has stepped on, so it is marked "avoid". Unwalked is *not* the same as bad.

The robot then trains its network on these labels, and in a few minutes it can look at a new photo and colour the whole picture by itself.

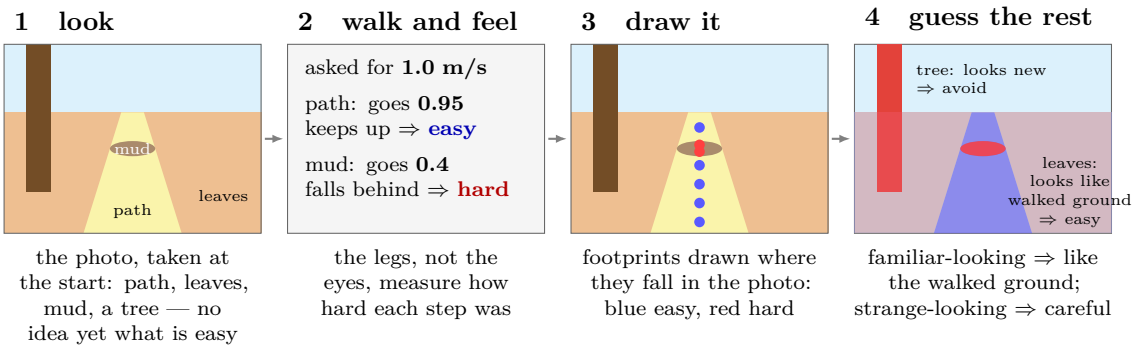


Figure 14.41. The whole of Wild Visual Navigation in four pictures. (1) The robot photographs the ground ahead. (2) It walks there; its legs tell it where it kept up (easy) and where it fell behind (hard). (3) It draws those findings onto the photo it took earlier. (4) For ground it never walked on, it copies what it learned from ground that looks similar, and is cautious about anything that looks new.

How to read it: Follow the mud: unknown in picture 1, felt as hard in picture 2, drawn red in picture 3. Then look at the leaves: never walked on, yet coloured easy in picture 4 — because they look like walked ground.

In plain words — the technical words, translated

traversability τ	a score from 0 (impossible) to 1 (easy) for "how easy did this ground feel to walk on"
velocity error	how much the robot fell behind the speed it was asked to go
sigmoid	a dial that turns "how far behind" into that 0–1 score
reprojection	finding your later footprints in an earlier photo
superpixels / segments	cutting the photo into puzzle pieces of similar colour, so the robot labels pieces, not single pixels
DINO features	a short description of how a puzzle piece looks, from a network trained without labels
autoencoder, reconstruction error	a "have I seen something like this before?" test: small error means familiar, large means new
confidence c	the answer to that test, from 1 (familiar) to 0 (new)
the loss	walked pieces: learn the score the legs measured; unwalked pieces: learn "avoid" *only as much as they look new*

Everyday analogy — a new colleague learning the building

A new colleague does not know which doors in the office are locked. They try a few: some open, some don't. After that, when they see a door that looks like one that opened (same handle, same corridor), they assume it will open. When they see a door unlike any they tried (a heavy steel door with a keypad), they assume it is locked until proven otherwise. They never treat "a door I have not tried" as "a locked door" — only "a door unlike any I have tried". *Where it breaks:* a door can look exactly like the open ones and still be locked; likewise ground can look like the path and be a hidden hole. Looks are only a guess.

The next subsection turns each moment of the story into its formula, one question at a time; after that come the full details.

14.5.2 From the story to the formulas: why each piece of maths is needed

A robot cannot act on sentences like "the mud felt hard" or "the leaves look familiar". Each sentence of the story has to become a *number* the computer can calculate and learn from. That is the only job of the formulas in this section. Below, every formula is introduced by the question from the story that it answers; read the question first, then the formula is just the question written precisely.

Question 1 (moment 2): "How far behind did I fall?" We need one number that is 0 when the robot keeps up perfectly and grows the more it falls behind — whether it is too slow or drifts sideways.

$$v_{\text{err}} \;=\; \tfrac{1}{2}\Big[\; \underbrace{(\bar{v}_x - v_x)^2}_{\text{forward: asked minus managed}} \;+\; \underbrace{(\bar{v}_y - v_y)^2}_{\text{sideways: asked minus managed}} \;\Big].$$

Why each piece: the difference "asked minus managed" is the falling-behind itself; *squaring* makes it positive either way and makes a big stumble count much more than a small wobble; the $\frac{1}{2}$ is only a convention. *Story numbers:* on the path $\frac{1}{2}(1.0 - 0.95)^2 \approx 0.001$ (almost nothing); in the mud $\frac{1}{2}(1.0 - 0.4)^2 = 0.18$ (a lot).

Question 2 (moment 2): "So was it easy or hard, on a 0–1 scale?" Numbers like 0.001 and 0.18 are not a score yet: we want 1 for easy, 0 for hard, and a clear "halfway" point. A *sigmoid* is exactly such a dial (Figure 14.42, left):

$$\tau \;=\; \text{sigmoid}\big(-k\,(v_{\text{err}} - v_{\text{thr}})\big)$$

Read it from the inside out: $v_{\text{err}} - v_{\text{thr}}$ asks "did I fall behind more or less than the halfway amount?"; multiplying by $-k$ flips the sign (more error ⇒ lower score) and sets how sharp the switch is; the sigmoid turns the result into a score between 0 and 1. *Why each piece:* v_{thr} is the amount of falling behind that counts as "half easy" ($\tau = 0.5$); the minus sign turns "more error" into "lower score"; k decides whether the dial moves gently or snaps. *Story numbers* (with $k = 20$, $v_{\text{thr}} = 0.1$): path $\tau \approx 0.88$, mud $\tau \approx 0.17$.

Question 3 (moment 3): "Where are my footsteps in the old photo?" This is pure geometry, and it is the same camera rule as in §14.4: a point at depth Z and sideways offset X appears at pixel $u = c_x + f\,X/Z$. The robot knows where the camera was when it took the photo and where each foot landed later, so it can compute where each footprint appears in the photo. *Why needed:* without it, the score from the legs could not be attached to any pixel.

Question 4 (moment 3): "What label does each puzzle piece get?" A piece of the photo may contain several footprints. Its label is simply the *average* of their scores:

$$\tau_n = \text{average of the footprint scores inside piece } n \qquad \text{(no footprints: "unknown").}$$

Question 5 (moment 4): "Does this piece look like ground I walked on?" We need a number for "how unfamiliar". The autoencoder is trained to rebuild the descriptions (DINO features) of walked pieces only; how badly it rebuilds a piece is that number:

$$\mathcal{L}_{\text{reco}}(\boldsymbol{f}_n) \;=\; \underbrace{\delta_{\tau_n \neq 0}}_{\text{learn only from walked pieces}} \cdot \underbrace{\text{average over the 90 numbers of (rebuilt} - \text{original)}^2}_{\text{how badly it rebuilds this piece}}.$$

Why each piece: the switch δ makes sure the autoencoder only ever learns what *walked* ground looks like — that is what makes a large error mean "unlike anything walked".

Question 6 (moment 4): "How much should I trust that it is familiar?" The error has no natural scale, so the robot compares it with the errors of pieces it *did* walk on: their typical error μ and spread σ. Anything no worse than typical is fully trusted; beyond that, trust falls off smoothly (Figure 14.42, right):

$$c \;=\; \begin{cases} 1 & \text{if the error is at most the typical walked error } \mu, \\ \exp\!\Big(-\dfrac{(\text{error}-\mu)^2}{2\,(2\sigma)^2}\Big) & \text{otherwise (a bell-shaped drop).} \end{cases}$$

Why this shape: the threshold adjusts itself as the robot walks more, because μ and σ are re-measured all the time. *Story numbers* ($\mu = 0.5$, $\sigma = 0.2$): leaves, error 0.6, get $c \approx 0.97$; the tree, error 2.0, gets $c \approx 0$.

Question 7 (moment 4): "What exactly should the network learn?" Two different lessons, one for each kind of piece, added up:

$$\mathcal{L}_{\text{trav}} \;=\; \underbrace{\sum_{\text{walked}} (\text{prediction} - \tau_n)^2}_{\text{learn what the legs measured}} \;+\; \underbrace{\sum_{\text{unwalked}} (1 - c_n)\,(\text{prediction} - 0)^2}_{\text{learn "avoid" only as much as it looks new}} .$$

Why each piece: squaring punishes big mistakes most; the target 0 is the careful default for unknown ground; the weight $(1 - c)$ is the "only if it looks new" from the story — familiar leaves ($c \approx 1$) get weight ≈ 0, the strange tree ($c \approx 0$) gets full weight.

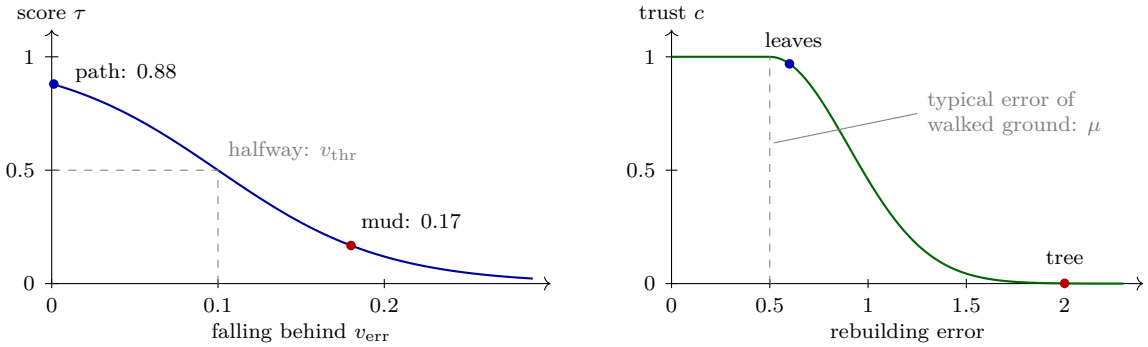


Figure 14.42. The two "dials" of WVN. Left: falling behind is turned into an easy–hard score; the path and the mud from the story sit at the two ends. Right: the familiarity error is turned into trust; up to the typical error of walked ground trust stays at 1, then it drops like a bell curve — the leaves stay trusted, the tree is not.

How to read it: In both graphs, find the story's two examples. Each formula is only a smooth, automatic way of putting them at opposite ends.

Table 14.5. The story and the maths, side by side.

In the story	In the maths	Why it is needed
"I fell behind in the mud"	v_{err}	one number for how far behind
"the mud was hard"	$\tau = \text{sigmoid}(\dots)$	turn that into a 0–1 easy/hard score
"draw my footprints on the old photo"	$u = c_x + fX/Z$	find where each footprint appears in the photo
"this piece of the photo was walked"	τ_n (average)	one label per puzzle piece
"have I seen ground like this?"	$\mathcal{L}_{\text{reco}}$	a number for how unfamiliar a piece looks
"how sure am I that it is familiar?"	c	turn unfamiliarity into trust between 0 and 1
"learn what I felt; be careful only with new-looking ground"	$\mathcal{L}_{\text{trav}}$	the lesson the network is trained on

14.5.3 Background: what you need first

Traversability. **Traversability** is the answer to "can this robot move over this ground?", as a number $\tau \in [0, 1]$: 1 means easy, 0 means impossible. It depends on the robot: a four-legged robot walks through bushes a wheeled one would get stuck in. So there is no universal "traversable" label a person could draw once and for all.

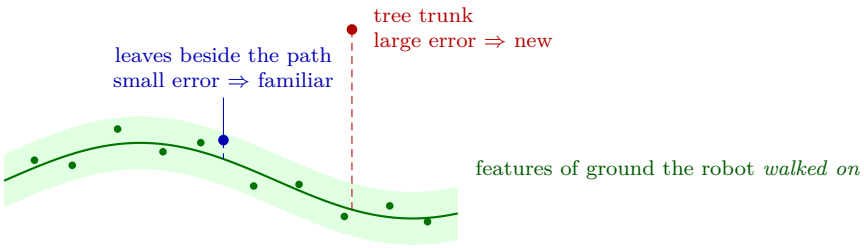
Commanded versus actual velocity. In the system of this section, a person walks next to the robot and steers it with a remote control for a few minutes. The remote control gives a *reference* (commanded) velocity $\bar{\boldsymbol{v}}$ — "go forward at 1 m/s". The robot's own state estimator measures the *actual* velocity $\boldsymbol{v}$, from its joint sensors and motion sensor (this is proprioception). On firm ground the two match. In deep leaves or tall grass the robot struggles and falls behind the command. The *mismatch* is therefore a measurement of how hard the ground is — made by the robot's own body.

The sigmoid. $\text{sigmoid}(x) = 1/(1 + e^{-x})$ squashes any number into $(0, 1)$: large positive x gives nearly 1, large negative gives nearly 0, and $x = 0$ gives exactly 0.5.

Superpixels. Predicting a separate value for each of a million pixels is slow and noisy. Instead, the image is cut into about a hundred *segments* (*superpixels*): patches of neighbouring pixels with similar colour, found by a simple algorithm called SLIC. Everything below is done per segment.

DINO features per segment. Each segment is described by a feature vector $\boldsymbol{f}_n$: the DINO-ViT features (Chapter 11) of its pixels, averaged. DINO was trained without labels on millions of images, so similar looking ground gives similar vectors. In the paper $\boldsymbol{f}_n$ has $E = 90$ numbers.

Autoencoders as familiarity detectors. An *autoencoder* (Chapter 4) squeezes its input through a narrow middle layer and tries to rebuild it. Trained only on one kind of input, it learns to rebuild that kind well and everything else badly. So its *reconstruction error* works as a familiarity test: small error means "looks like what I was trained on", large error means "new to me" (Figure 14.43).



The autoencoder, trained only on the green points, learns to map any input onto the green curve. Its reconstruction error is roughly the length of the dashed line: the distance from the input to the nearest familiar feature.

Figure 14.43. Reconstruction error as a familiarity test, drawn in a 2-D sketch of feature space. An autoencoder trained only on features of walked-on ground (green) rebuilds similar features well (blue: small error) and new ones badly (red: large error).

14.5.4 The problem

A legged robot — the four-legged ANYmal — must walk through a forest, choosing its path from its camera images alone. Nobody will label "forest floor traversability" for this robot, and ready-made datasets of city streets say nothing about moss, mud and leaf litter. Wild Visual Navigation (WVN, Frey et al., 2023) gets its labels from the robot's own body, and learns online, in a few minutes.

In plain words — walk first, label later

The camera cannot tell by looking whether the leaves ahead are firm. But the robot can walk there and *feel* it: if its legs keep up with the command, the ground was easy. A few seconds later, the robot looks back at the photo it took of that ground and writes the answer onto it. Now that photo has a label, made by the robot's body. There is one catch: most of the photo shows ground the robot never walked on, and "I never went there" must not be read as "it is impassable".

Model at a glance: Wild Visual Navigation — labels from the robot's own legs Frey, Mattamala, Chebrolu, Cadena, Fallon & Hutter, 2023

The free label	How well the robot's actual velocity follows the commanded one, measured while it walks over a patch of ground; turned into a score τ and written onto earlier camera images of that patch.
The unusual bit	The label arrives *after* the image it labels. Supervision travels backwards in time along the robot's path.
Sensors	Camera, plus proprioception and the pose estimate. All were on the robot anyway.
Model	A small network on DINO features of each image segment, with two heads: one predicts τ, one reconstructs the features (the familiarity test).
Why it matters	The label is right *for this robot* by construction; no annotator could guarantee that.
What it buys	A traversability predictor learned in the field in under five minutes.

14.5.5 How it works, step by step

Step 1: turn the body's experience into a score. The mismatch between commanded and actual velocity is measured on the ground plane:

$$v_{\text{err}} \;=\; \tfrac{1}{2}\big[(\bar{v}_x - v_x)^2 + (\bar{v}_y - v_y)^2\big]. \tag{14.4}$$

This signal is noisy, so it is first smoothed (with a Kalman filter, a standard noise-smoothing method), and then squashed into a score:

$$\tau \;=\; \text{sigmoid}\big(-k\,(v_{\text{err}} - v_{\text{thr}})\big). \tag{14.5}$$

The threshold v_{thr} is the error that counts as "half traversable" ($\tau = 0.5$), and k sets how quickly the score drops around it. Both are tuned once per robot (Frey et al., 2023). The minus sign makes *more* error give a *lower* score.

Worked Example — From footsteps to a score

Take (for illustration) $k = 20$ and $v_{\text{thr}} = 0.1$. The command is $\bar{\boldsymbol{v}} = (1.0, 0)\,\text{m/s}$.
Firm path: the robot manages $\boldsymbol{v} = (0.95, 0)$. Then $v_{\text{err}} = \frac{1}{2}(0.05)^2 = 0.00125$ and $\tau = \text{sigmoid}(-20 \cdot (0.00125 - 0.1)) = \text{sigmoid}(1.98) \approx 0.88$.
Deep leaves: the robot only manages $\boldsymbol{v} = (0.4, 0)$. Then $v_{\text{err}} = \frac{1}{2}(0.6)^2 = 0.18$ and $\tau = \text{sigmoid}(-20 \cdot 0.08) = \text{sigmoid}(-1.6) \approx 0.17$.
The lecture's plot (slide 25, Figure 14.45, left) shows exactly this on a real run: whenever the blue actual velocity falls well below the black reference, τ drops to red.

Step 2: write the score onto past images. The score in Step 1 describes the ground *under the feet now*. The network to be trained, however, looks at *camera images*, which

show the ground *ahead*. The bridge is the pose estimate. The robot remembers where its feet have been, with a score for each spot, and it has stored earlier images together with the pose they were taken from. Each footprint is projected into those earlier images (the same projection as Eq. 14.1), painting the path the robot later walked, with its scores, onto the photo (Figure 14.44 and Figure 14.45, right). Every segment that the painted path touches gets a label τ_n: the average score of the path inside it. Every segment the path does not touch gets $\tau_n = 0$, which here means *unknown*, not *bad* (Frey et al., 2023).

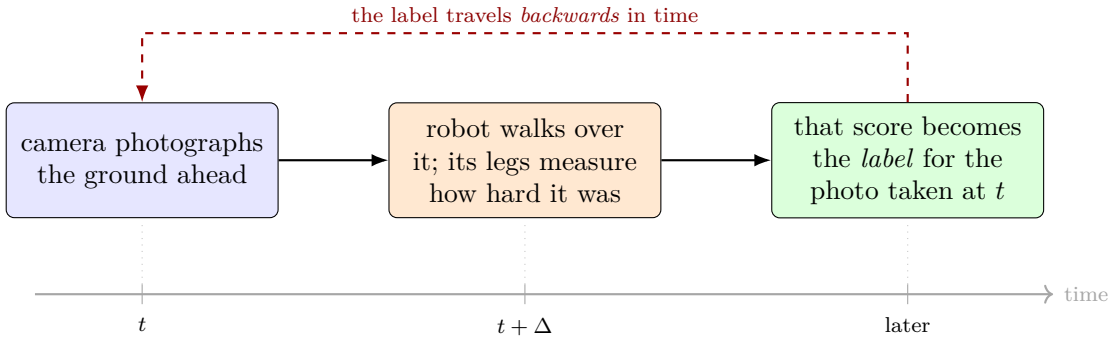


Figure 14.44. Labels from the robot's own body. The label for a camera image is produced by the robot's legs, seconds after the image was taken.

How to read it: The red arrow is the whole idea: evidence collected at $t + \Delta$ labels an observation made at t.

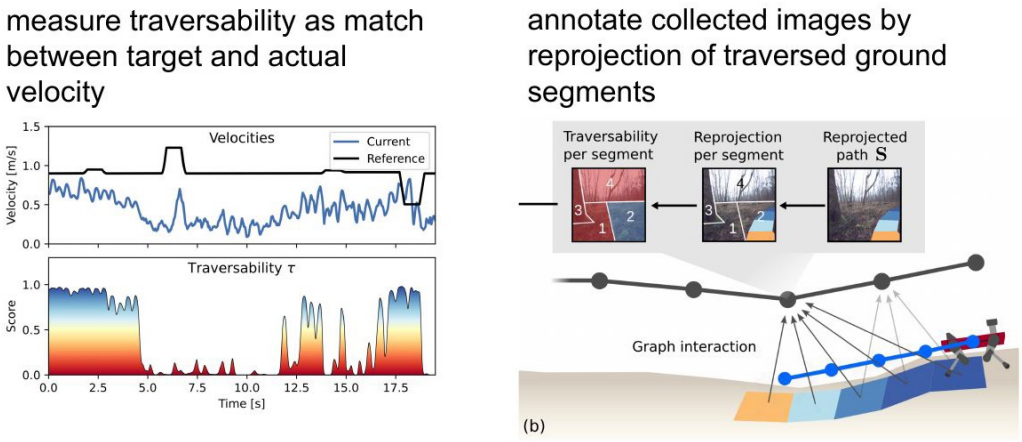
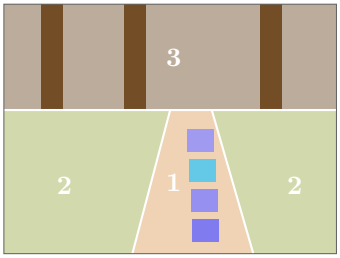


Figure 14.45. The two halves of WVN's label generation (Lecture 09, slide 25; from Frey et al., 2023). Left: commanded (black) and actual (blue) velocity over 20 seconds, and the score τ below it — high (blue) while the robot keeps up, dropping to red where it falls behind. Right: the robot's recent footprints (coloured patches on the ground, with their scores) are reprojected into an earlier camera image ("Reprojected path **S**"), then spread over the image segments that the path touches ("Traversability per segment"). Segments 1 and 2 get labels; segments 3 and 4 do not.

Step 3: the problem of the places you did not go. After Step 2, a typical image has labels on a thin strip — the walked path — and nothing everywhere else (Lecture 09, slide 26: "we know nothing" outside the strip). What should the network learn about the rest? Two obvious answers are both wrong:

- *Treat unlabelled segments as untraversable* (target 0). Then the leaf-covered ground right next to the path, which is just as easy, is taught to be impassable. The robot becomes afraid of anything it has not literally stepped on, stops exploring, and never collects evidence to the contrary.
- *Ignore unlabelled segments.* Then the network only ever sees the walked path, which the operator chose because it was walkable. It has never seen a tree trunk labelled as bad, so it may happily call one traversable.

WVN's answer (slides 27–28) uses the familiarity test: for an unlabelled segment, *if it looks like ground we walked on, don't push it to* 0*; if it looks new, push it to* 0.



segments of one image;
blue: the path walked later

What the robot knows. It walked along the path (1) and measured a good score there. It never set foot on the leaves beside the path (2) or on the tree trunks (3).

What the loss does. Unwalked segments are pushed towards 0 with weight $1 - c$, where c is the familiarity confidence. The leaves look like the path ($c = 0.97$, weight 0.03) and are left almost alone; the trunks look new ($c \approx 0$, weight ≈ 1) and are firmly taught "not traversable".

segment	what we know	τ_n	confidence c	what the loss asks
1 path	walked, score measured	0.9	—	predict 0.9
2 leaves beside path	not walked, looks familiar	(none)	0.97	almost nothing
3 tree trunks	not walked, looks new	(none)	≈ 0	predict 0, full weight

Figure 14.46. The three kinds of segment WVN must handle (after Lecture 09, slides 26–28; numbered for this example). Segment 1 was walked on and has a measured score. Segments 2 were not walked on but look like segment 1 to the familiarity test. Segment 3 was not walked on and looks like nothing the robot has walked on.

How to read it: Read the right-most column. Only segment 3 is taught "no", and only because it looks new*, not because it is unlabelled.*

Step 4: the familiarity test — reconstruction loss. The autoencoder $f^{\boldsymbol{\theta}_r}_{\text{reco}}$ is trained to rebuild the DINO features of *walked* segments only (slide 27):

$$\mathcal{L}_{\text{reco}}(\boldsymbol{f}_n) \;=\; \underbrace{\delta_{\tau_n \neq 0}}_{\substack{\text{only walked}\\ \text{segments}}} \cdot \underbrace{\frac{1}{E}\sum_{e=1}^{E} \left\| f^{\boldsymbol{\theta}_r}_{\text{reco}}(\boldsymbol{f}_n)_e - \boldsymbol{f}_{n,e} \right\|^2}_{\substack{\text{rebuilding error, averaged}\\ \text{over the } E \text{ feature numbers}}} . \tag{14.6}$$

Here $\boldsymbol{f}_{n,e}$ is the e-th of the $E = 90$ numbers in segment n's feature vector, and $\delta_{\tau_n \neq 0}$ is 1 for walked segments and 0 otherwise (the lecture's note: "only where we have data"). After training, walked-looking segments have a small error and new-looking ones a large error.

The error is then turned into a *confidence* $c \in [0, 1]$ that a segment is familiar. WVN fits a bell curve (mean μ, spread σ) to the errors of the walked segments in the current batch; a segment whose error is below μ gets $c = 1$, and above it, c falls off like the bell curve: $c = \exp(-(\mathcal{L}_{\text{reco}} - \mu)^2/(2(k_\sigma\sigma)^2))$, with a tuning factor k_σ ($= 2$ in the paper) (Frey et al., 2023). With $\mu = 0.5$ and $\sigma = 0.2$: the leaves beside the path, error 0.6, get $c = \exp(-0.01/0.32) \approx 0.97$; a tree trunk, error 2.0, gets $c = \exp(-2.25/0.32) \approx 0.001$.

Step 5: the traversability loss, term by term. The traversability head $f^{\boldsymbol{\theta}_t}_{\text{trav}}$ predicts τ for each segment. Its loss has one term for each case (slide 28):

$$\mathcal{L}_{\text{trav}}(\boldsymbol{f}) \;=\; \underbrace{\delta_{\tau_n=0} \sum_n (1 - c(\boldsymbol{f}_n)) \left\| f^{\boldsymbol{\theta}_t}_{\text{trav}}(\boldsymbol{f}_n) - 0 \right\|^2}_{\text{unwalked: "assume 0", but only as much as it looks new}} \;+\; \underbrace{\delta_{\tau_n \neq 0} \sum_n \left\| f^{\boldsymbol{\theta}_t}_{\text{trav}}(\boldsymbol{f}_n) - \tau_n \right\|^2}_{\text{walked: predict the measured score}} . \tag{14.7}$$

Reading it piece by piece:

- $\delta_{\tau_n=0}$ and $\delta_{\tau_n \neq 0}$ are switches: each segment falls in exactly one of the two sums.
- *Second term:* on walked segments, an ordinary squared error between prediction and measured score. This is the self-generated label doing its job.
- *First term:* on unwalked segments, a squared error towards 0 ("assume zero traversability on strange-looking terrain") — multiplied by $1 - c$. For familiar-looking segments $c \approx 1$, the weight is ≈ 0, and the term does nothing: the network is free to generalise from similar walked ground. For new-looking segments $c \approx 0$, the weight is ≈ 1, and the network is taught caution.

The two networks share their hidden layers and are trained together: $\mathcal{L} = w_{\text{trav}}\mathcal{L}_{\text{trav}} + w_{\text{reco}}\mathcal{L}_{\text{reco}}$ (Frey et al., 2023).

Worked Example — The traversability loss on three segments

Use the segments of Figure 14.46, and suppose the network currently predicts 0.8 for the path (1), 0.7 for the leaves beside it (2) and 0.6 for the tree trunks (3).

- Segment 1 (walked, $\tau_1 = 0.9$): $(0.8 - 0.9)^2 = 0.010$.
- Segment 2 (unwalked, $c = 0.97$): $(1 - 0.97) \cdot 0.7^2 = 0.03 \cdot 0.49 \approx 0.015$.
- Segment 3 (unwalked, $c = 0.001$): $(1 - 0.001) \cdot 0.6^2 \approx 0.360$.

Almost the whole loss, and so almost the whole gradient, is the message "stop calling tree trunks traversable". The leaves beside the path, which the robot never touched, keep their optimistic 0.7, because they look like the path. *Mistake to spot:* with a constant weight 1

instead of $1 - c$, segment 2 would add 0.49 and the network would be pushed hard to call easy ground impassable.

Common Confusion

"No label" is not "negative label". The heart of WVN's loss is refusing to read "the robot never went there" as "the robot cannot go there". Familiarity is a property of how a segment *looks* (is its feature close to walked features?), and the loss uses it to decide how much an unwalked segment should be pushed towards 0. Both naive choices — all-negative and ignore — fail, in opposite directions.

Results. Starting from an untrained network, WVN learns to separate traversable from untraversable terrain after less than five minutes of walking with an operator, in forests and parks it had never seen, with no human labels (Frey et al., 2023). The robot can then use its own predictions to walk on its own (Figure 14.47).

Figure 14.47. WVN at work (Lecture 09, slide 29; from Frey et al., 2023). Middle: the robot's height map of the surrounding forest, with the robot and its walked footprint (blue). Bottom left: the operator and the robot. Right: the camera image and the learned prediction overlaid on it — the leaf-covered ground is marked as traversable, the tree trunk is not.

Paper Analysis: Frey et al. (2023) (Wild Visual Navigation)

Paper: Fast Traversability Estimation for Wild Visual Navigation (RSS 2023). **Problem:** traversability for a legged robot in nature, with no labels available. **Insight:** velocity-tracking error is a label the robot's body produces for free, and projecting past footprints into past images turns it into image labels. **Method:** SLIC segments; averaged DINO-ViT features per segment; τ from Eq. (14.5); a small network with a traversability head and a reconstruction head; confidence from the reconstruction error weights the "assume 0" term. **Losses:** Eq. (14.6) and Eq. (14.7). **Results:** useful traversability after under five minutes of in-field training. **Limitations:** the label is right for *this* robot, gait and day; familiarity inherits DINO's blind spots. **What to remember:** the robot's own experience is its teacher — as long as "no data" is never silently read as "negative data".

In plain words — try it interactively

Terrain Lab — play the robot's walk and watch the speed error become τ (with k and v_{thr} as sliders), see the footsteps painted back into the earlier photo, click segments to see their familiarity confidence c, and read the loss segment by segment.
`https://claude.ai/artifact/45fhx12mF76MktfUU2gqUN`
(Private until shared from the page's Share menu. Add `#step3`, or any step number, to the link to open that step.)

Failure Modes

The body, not the ground. A tired motor or a slipping foot raises the velocity error on easy ground; the label then describes the robot, not the terrain. **Painting in the wrong place.** Scores land on the right image segments only if the poses are right; drift or grass between camera and ground puts a score on the wrong patch. **Familiar but dangerous.** The familiarity test knows only how things look. A drop-off covered in the same leaves as the path looks familiar and is not pushed to 0. **The familiar set moves.** As the robot walks new ground, the set of walked features grows and shifts; the confidence must be refitted all the time, which WVN does batch by batch.

14.6 Other Sensors as Supervision: Teaching Microphones to See Cars

14.6.1 The whole idea first, as a story (almost no maths)

Start with yourself. You go bird-watching with a friend. Every time a bird sings, your friend points at it and names it: "robin, in that tree". After a few weekends you notice you can do it yourself *with your eyes closed*: the song alone tells you which bird and roughly where. Your friend's eyes were the teacher; your ears became the student. Nobody gave you a book of bird songs.

Now the car, in four moments (Figure 14.48).

1. **The cameras see the cars.** A car drives through a city with a colour camera, a heat (thermal) camera and a depth camera, each running an ordinary car detector: the three *teachers*. Each draws boxes around the cars it finds, with a confidence score.
2. **Three opinions become one.** The teachers' boxes overlap and sometimes disagree. Boxes that clearly describe the same car are merged, keeping the most confident one; a car only one teacher saw is kept too. The result is one list of car boxes for this moment.
3. **The microphones hear the same moment.** Eight microphones on the roof record one second of sound around that instant. Engines and tyres make patterns in the sound, louder in the microphones that face them.
4. **The student learns to answer from sound.** A fourth network, the *student*, gets only the sound and must output the same car boxes the teachers agreed on. Over thousands of moments it learns. Then the cameras can be switched off — at night, in fog, in a tunnel, the student still "hears" where the cars are.

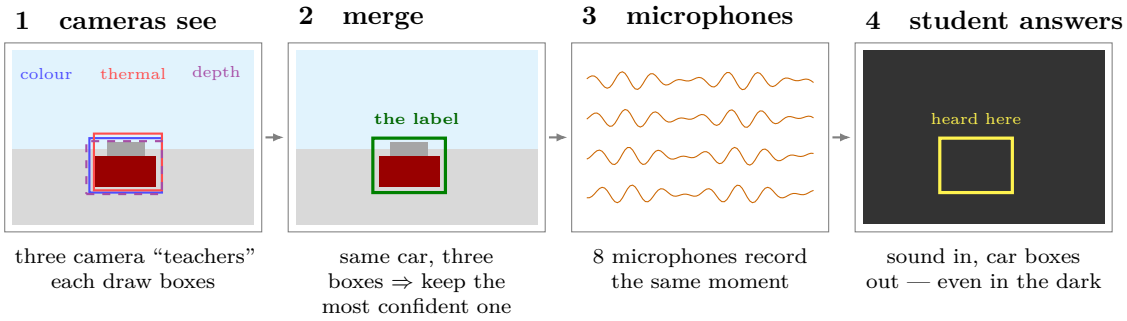


Figure 14.48. MM-DistillNet in four pictures. (1) Three camera teachers find the car. (2) Their overlapping boxes are merged into one label. (3) The microphones record the same moment. (4) The student learns to produce that label from sound alone, so it works when the cameras cannot see.

How to read it: The green box of picture 2 is the only thing that passes from the cameras to the student.

In plain words — the technical words, translated

teacher / student	the networks that give the answers / the network that learns to copy them
object detection, box, score	finding objects as boxes, each with how sure the detector is
IoU	how much two boxes overlap (overlap area ÷ total area)
NMS (non-maximum suppression)	merging overlapping boxes: keep the most confident, delete its near-copies
pseudo ground truth	the merged boxes, used as the student's label
spectrogram	a picture of sound: time left to right, pitch bottom to top, brightness = loudness
focal loss	the "how wrong was the student" number, with easy cases turned down
attention map, MTA loss	where inside the network the teachers "look"; the student is nudged to look there too

Everyday analogy — learning bird songs from a friend

Your friend's eyes teach your ears. *Where it breaks:* if your friend always mistakes one bird for another, you learn the same mistake — the student inherits whatever all its teachers get wrong; and a bird that never sings can never be heard, however well you learn.

14.6.2 From the story to the formulas: why each piece of maths is needed

Question 1 (moment 2): "Are these two boxes the same car?" We need a number for "how much do two boxes overlap":

$$\text{IoU} \;=\; \frac{\text{area where they overlap}}{\text{area they cover together}}.$$

It is 1 for identical boxes and 0 for boxes that do not touch. NMS treats two boxes with IoU > 0.5 as the same car and keeps only the more confident one. *Example:* two 10×10 boxes shifted by 2 pixels overlap in $8 \times 10 = 80$ pixels out of 120: IoU ≈ 0.67, so they are merged.

Question 2 (moment 4): "How wrong was the student?" For each candidate box, the student gives a probability p_t to the right answer (car or no car, from the merged label). The basic "how wrong" number is

$$-\log p_t \qquad \text{(small when } p_t \approx 1,\ \text{large when } p_t \approx 0\text{)}.$$

Question 3: "Why not let every candidate count equally?" A detector checks thousands of candidate boxes per image, almost all of them obviously empty road. Their many small losses would drown out the few hard, interesting cases. So the loss is turned down for easy cases:

$$\mathcal{L}_{\text{focal}} \;=\; -\;\underbrace{\alpha}_{\text{balance}}\;\underbrace{(1-p_t)^{\gamma}}_{\approx 0 \text{ when already confident}}\;\underbrace{\log p_t}_{\text{how wrong}}\;.$$

Numbers ($\alpha = 0.25$, $\gamma = 2$): an easy case with $p_t = 0.9$ gives 0.00026; a hard one with $p_t = 0.1$ gives 0.47 — 1800 times more (Figure 14.52).

Question 4: "Can the student also learn *how* the teachers look?" Each network has internal maps of where it pays attention. The three teachers' maps are multiplied (places all three look at stay bright), and the student's map is compared with that product by a divergence (the KL divergence: 0 when the two maps have the same shape):

$$\mathcal{L}_{\text{MTA}} \;=\; \beta \sum_{\text{layers}} \text{KL}(\text{student's map} \,\|\, \text{teachers' combined map}).$$

Why needed: the boxes say *what* the answer is; the maps say *where to look*, a richer hint for learning. The total is simply $\delta\,\mathcal{L}_{\text{focal}} + \omega\,\mathcal{L}_{\text{MTA}}$.

Table 14.6. MM-DistillNet: the story and the maths, side by side.

In the story	In the maths	Why it is needed
"these boxes are the same car"	IoU > 0.5, NMS	merge three opinions into one label
"how wrong was the student?"	$-\log p_t$	the basic mistake size
"don't drown in easy cases"	$(1-p_t)^{\gamma}$ in the focal loss	let the hard cases teach
"look where the teachers look"	KL to the product of attention maps	a richer signal than boxes alone

14.6.3 Background: what you need first

Object detection. An **object detector** looks at an image and outputs a list of *boxes*. Each box comes with a class ("car") and a *confidence score* between 0 and 1 (how sure the detector is). Detectors usually produce several overlapping boxes for one object, and those have to be cleaned up.

IoU of two boxes, and non-maximum suppression. The overlap of two boxes is measured with the same IoU as in Figure 14.16: overlap area divided by union area. **Non-maximum suppression** (NMS) cleans up a list of boxes: sort the boxes by score; keep the

best one; delete every remaining box that overlaps it by more than a threshold (IoU above 0.5 here); repeat with the next surviving box (Figure 14.49). What is left is one box per object.

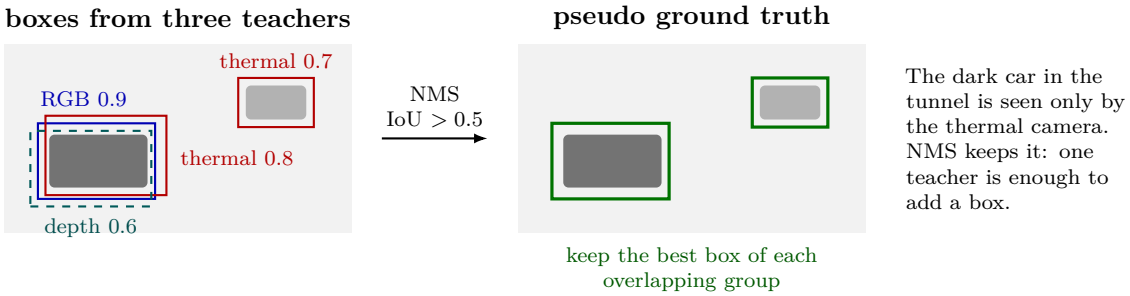


Figure 14.49. Non-maximum suppression merges the three teachers' detections. The three boxes around the left car overlap heavily, so only the best (RGB, 0.9) survives. The right car was found by one teacher only and is kept. The result is the pseudo ground truth for the audio student.

Teachers and students. As in Chapters 11–12, a *teacher* network's outputs become targets for a *student* network. Here the teachers and the student read *different sensors*.

Thermal and depth cameras. A thermal camera sees heat, so it works in darkness and picks out warm engines and people; a depth camera measures distance. Each is a different "view" of the same street.

Sound as an image. A microphone records air pressure over time — a wiggly line. A *spectrogram* turns one second of that line into an image: time along one axis, pitch (frequency) along the other, brightness for loudness. A car appears as a characteristic smear of engine and tyre noise. With an array of eight microphones at different places on the car's roof, a sound from the left reaches the left microphones slightly earlier and louder; eight spectrograms, stacked like the colour channels of an image, therefore carry information about *where* a sound comes from.

14.6.4 The problem

Cameras fail in darkness, fog, glare and tunnels; microphones do not. Could a car learn to *detect other cars from sound alone*, using the fact that its cameras saw those same cars at the same moment? This is the question of MM-DistillNet (Rivera Valverde et al., 2021). The rig is a car with a camera unit and eight microphones on its roof (Lecture 09, slide 31); every passing car is an event that is both seen and heard.

In plain words — sight teaches hearing

Three "eyes" (a colour camera, a thermal camera and a depth camera) each run a car detector. Where they find cars, their boxes are merged into one answer. A fourth network, which only *hears*, is trained to output the same boxes from the sound recorded at that moment. Nobody labelled anything: the eyes' answer is the label for the ears. Afterwards, the ears work alone — including in the dark, where the eyes would have failed.

Model at a glance: MM-DistillNet — hearing where the cars are Rivera Valverde, Hurtado & Valada, 2021

The free label	Boxes from RGB, thermal and depth car detectors, merged by NMS, for the same moment the microphones recorded.
Why it is self-supervised	No one labels the new audio task. (The teachers are ordinary detectors trained earlier, each on its own existing dataset.)
Sensors	Training: camera unit plus eight microphones. Use: the microphones alone.
Losses	Focal loss on the boxes, plus the multi-teacher alignment (MTA) loss, which makes the student's internal "where am I looking" maps match the teachers'.
The point	The student does not just copy its teachers; it works where they cannot see.

14.6.5 How it works, step by step

Step by step — one training example

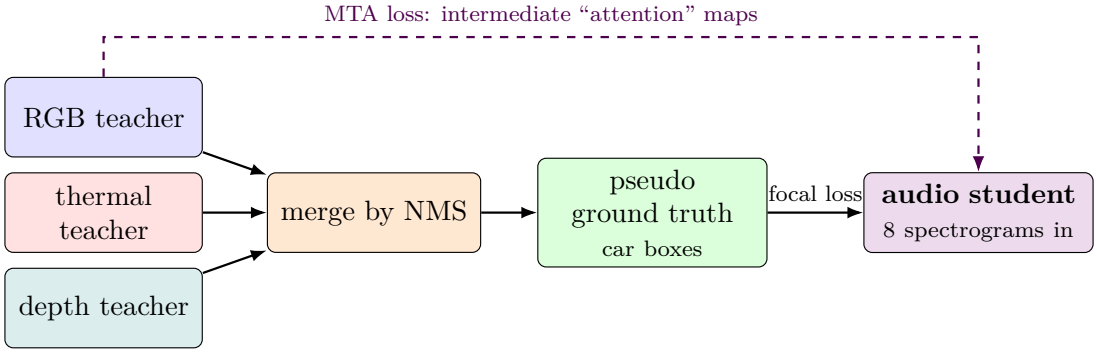
Step 1. Record. At one instant the rig stores an RGB image, a thermal image, a depth image, and one second of sound from each of the eight microphones, centred on that instant.

Step 2. Teachers detect. Three car detectors (all of the EfficientDet type, each already trained for its own camera) output boxes with scores.

Step 3. Merge. NMS with IoU threshold 0.5 turns the three lists into one: the pseudo ground truth (Figure 14.49).

Step 4. Student predicts. The eight spectrograms are stacked into one 8-channel input; the audio student, a detector with the same design as the teachers, predicts car boxes *in the camera image's coordinates*.

Step 5. Learn. Focal loss compares the student's boxes with the pseudo ground truth; the MTA loss compares its internal maps with the teachers'. Only the student is updated.



Why this counts as self-supervision. Nobody labelled the audio. Three sensors that happen to be pointed at the same scene vote on where the cars are, and their agreement becomes the target for a fourth sensor that cannot see at all.

What you get. A detector that works in darkness and in tunnels, where some teachers would fail. The student ends up *complementary* to its teachers, not a cheaper copy.

What you inherit. Every error the teachers share. If all three miss a car, the student is trained to miss it too, and nothing in the pipeline can notice.

Figure 14.50. MM-DistillNet in one picture. Sensors that are easy to interpret supervise one that is not, using only the fact that they record the same moment. Two things pass from teachers to student: the merged boxes (focal loss) and the teachers' internal maps of where they look (MTA loss).

How to read it: Follow left to right. The student never sees an image; it only receives targets.

The audio student starts from a network pre-trained on a simple pretext task: from the eight spectrograms, *count* how many cars the teachers see. This gives it a useful starting point before the harder box-prediction task (Rivera Valverde et al., 2021).

14.6.6 The losses, term by term

Focal loss: cross-entropy that ignores the easy cases. Recall cross-entropy for one prediction: if the model gives probability p_t to the correct answer, the loss is $-\log p_t$. It is small when p_t is near 1 and large when p_t is near 0. A detector scores tens of thousands of candidate boxes per image, and almost all of them are easy background (“no car here”). Each easy one adds a little cross-entropy, and together they drown out the few hard, informative cases. The **focal loss** fixes this by multiplying by a factor that vanishes for easy cases (slide 33):

$$\mathcal{L}_{\text{focal}} = -\underbrace{\alpha}_{\substack{\text{balance of}\\ \text{rare vs. common}}}\ \underbrace{(1-p_t)^{\gamma}}_{\substack{\text{small when already}\\ \text{confident}}}\ \underbrace{\log p_t}_{\text{cross-entropy}}, \tag{14.8}$$

with $\alpha = 0.25$ and $\gamma = 2$ in MM-DistillNet (Rivera Valverde et al., 2021). With $\gamma = 0$ and $\alpha = 1$ it is plain cross-entropy; the lecture's summary is that focal loss is “a variant of cross-entropy”, weighted by the model's own confidence (Figure 14.52).

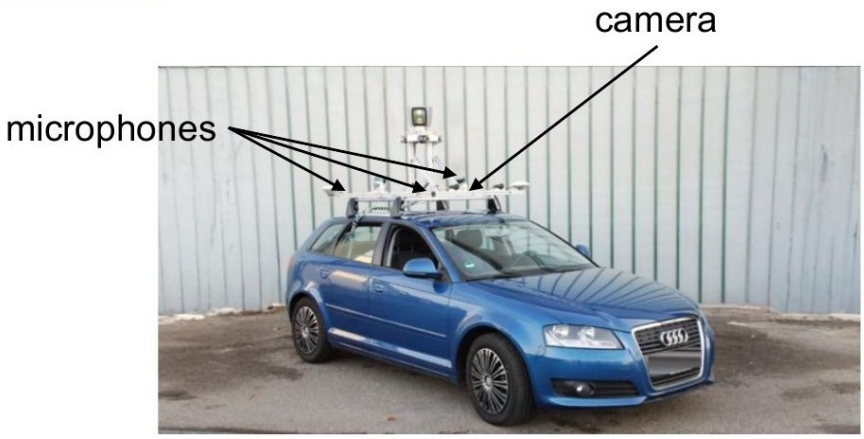
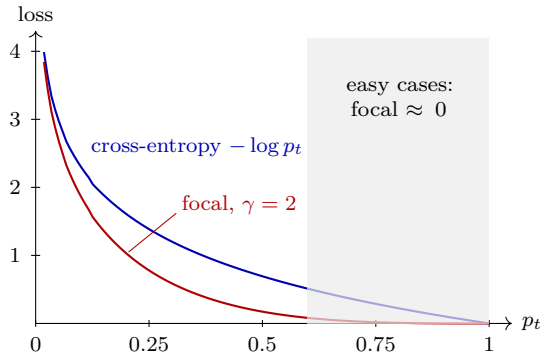


Figure 14.51. MM-DistillNet as drawn in the paper (Lecture 09, slides 32 and 34; Rivera Valverde et al., 2021). Three teachers (RGB, thermal, depth), each with feature layers P_3–P_5 and a class-and-box head, feed non-maximum suppression; the result supervises the audio student at the bottom, which reads spectrograms. The "MTA" blocks compare the student's P_3–P_5 layers with the teachers'.



	$p_t = 0.9$ (easy)	$p_t = 0.1$ (hard)	hard/easy
cross-entropy $-\log p_t$	0.105	2.30	22×
$(1-p_t)^2$	0.01	0.81	
focal ($\alpha = 0.25$)	0.00026	0.47	1800×

Figure 14.52. Focal loss versus cross-entropy (the plot leaves out α). For confident, correct predictions (grey region) the focal factor $(1-p_t)^2$ makes the loss almost zero; for hard ones it barely changes it. In the table, a hard example counts 22 times as much as an easy one under cross-entropy, but 1800 times as much under the focal loss.

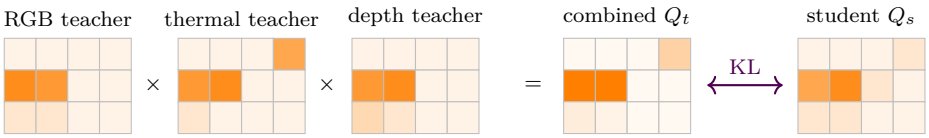
MTA loss: look where the teachers look. A detector's intermediate layers are grids of feature values. Averaging a layer's values over its channels gives one number per location: a *heat map* of where the network is "paying attention". This is called an *attention map*. The **multi-teacher alignment** (MTA) loss makes the student's attention maps match the teachers' at three layers $j \in \{P_3, P_4, P_5\}$ (slide 33):

$$\mathcal{L}_{\text{MTA}} = \underbrace{\beta}_{\text{weight}} \sum_j \underbrace{D_{\text{KL}}\left(\frac{Q_s^j}{\|Q_s^j\|_2} \,\middle\|\, \frac{Q_t^j}{\|Q_t^j\|_2} \right)}_{\substack{\text{how different are the student's and} \\ \text{the teachers' attention maps at layer } j}} , \tag{14.9}$$

with $\beta = 0.5$. Piece by piece:

- Q_s^j is the student's attention map at layer j.
- Q_t^j is the *combined* teachers' map. MM-DistillNet multiplies the three teachers' maps together, location by location (Rivera Valverde et al., 2021). A place all three teachers look at gets a high value; a place only one looks at gets a lower but non-zero value, so a car seen by only one sensor still counts a little (Figure 14.53).
- Dividing by $\|\cdot\|_2$ puts both maps on the same scale, so only the *shape* of the map — where it is bright — is compared, not its overall size.
- D_{KL} (Kullback–Leibler divergence, Chapter 2) measures how different two distributions are; it is 0 when they are equal.

The two losses do different jobs. The focal loss says *what the answer is* (which boxes). The MTA loss says *how to get there* (which parts of the input to attend to), a richer signal than the final boxes alone. The total is a weighted sum, $\mathcal{L} = \delta\,\mathcal{L}_{\text{focal}} + \omega\,\mathcal{L}_{\text{MTA}}$.



Bright = strong activation. All three teachers light up the car on the left, so the product is brightest there. Only the thermal teacher lights up the top-right cell (a car in a dark tunnel); the product keeps a weaker trace of it. The student, which only hears, is pushed to make its own map look like the combined one.

Figure 14.53. The idea of the MTA loss on a tiny 3×4 attention map (illustrative values, normalised to the brightest cell). Multiplying the teachers' maps rewards agreement; the KL term pulls the student's map towards the result.

Results. The authors recorded their own dataset: over 113,000 time-synchronised frames of RGB, depth, thermal and audio, from 24 drives through a city, including tunnels and streets with trams (Rivera Valverde et al., 2021). The audio-only student detects several cars at once from sound, even while its own car is moving, and outperforms earlier sound-based detectors

(Figure 14.54). Its limits are Chapter 12's: it can only be as good as what its teachers agree on, and it learns only what sound can reveal.

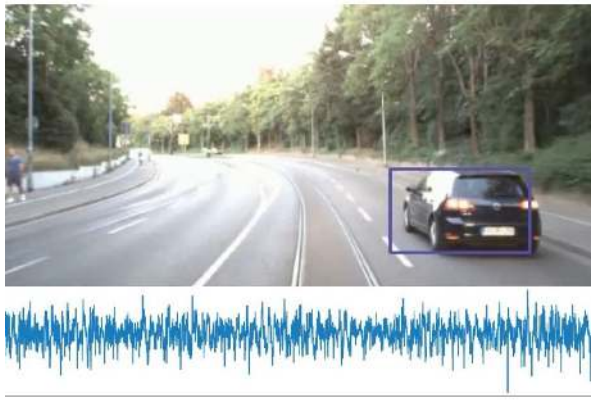


Figure 14.54. A car found by listening (Lecture 09, slide 35; from Rivera Valverde et al., 2021). The box is predicted by the audio student from the sound (waveform below); the camera image is shown only so that we can check it.

In plain words — try it interactively

Sound Lab — see the three camera teachers' boxes, step through non-maximum suppression with the real IoU values, look at the eight microphones' spectrograms, play with γ in the focal loss, compare the teachers' attention maps with their product, and switch to night.

`https://claude.ai/artifact/MQCLkNUPjdtndtLijAkuw6`

(Private until shared from the page's Share menu. Add `#step3`, or any step number, to the link to open that step.)

Failure Modes

Shared blind spots. If all three teachers miss the same car (say in heavy rain), the merged label misses it and the student is trained to miss it. **Some things make no sound.** A parked car is silent; the student cannot hear it, and forcing it to predict such boxes teaches it to invent detections. **Timing.** Every pair assumes the sound and images are perfectly synchronised; a small time offset shifts the target boxes relative to the sound that caused them.

14.7 Interaction as Supervision: A Robot That Practises on Its Own

14.7.1 The whole idea first, as a story (almost no maths)

Start with yourself. Think of a toddler learning to pick up blocks, alone in a playroom. They reach for a block. Either it is in their hand or it is not — nobody has to tell them which. When it slips, they try again, a bit differently. When they are done with a block, they toss it away, and it lands somewhere new — a fresh challenge, no adult needed. And they are most drawn to the tricky blocks they are not sure about, because that is where there is most to learn.

Now the robot, in four moments (Figure 14.55).

1. **Try.** The robot drives to an object and picks one of several ways to grasp it.
2. **Check.** It closes the gripper and feels whether it is holding something: yes (1) or no (0). That answer is the label — free, and always correct.
3. **Learn and choose the next try.** The robot has six small predictors that each guess how likely a grasp is to work. After each try, all six are corrected by the outcome. Next time, it favours grasps that look likely to work *and* grasps the six predictors disagree about — trying those settles the argument.
4. **Reset by dropping.** Instead of waiting for a person to put things back, it drives a random path and drops the object somewhere new. Dropping never fails, and each drop creates a new practice situation.

The same trick teaches it where to drive: every driving step costs a little, and reaching an object it can grasp pays it back.

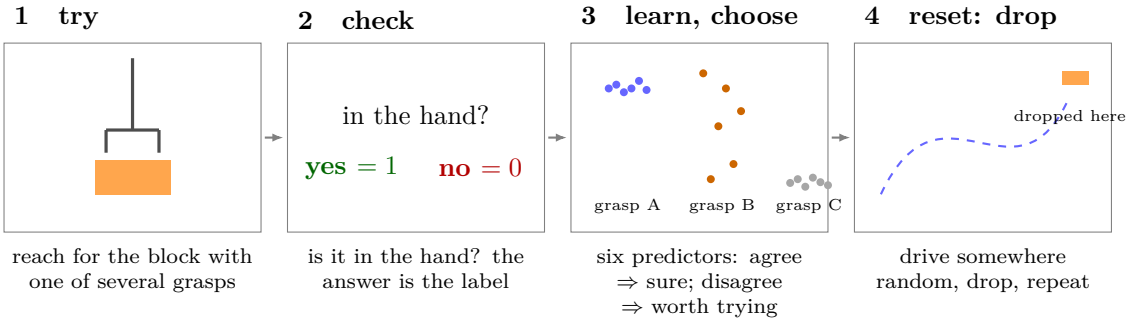


Figure 14.55. ReLMM in four pictures. (1) Try a grasp. (2) Feel whether it worked: the free label. (3) Update the six predictors; next, favour grasps likely to work and grasps they disagree about (B). (4) Drop the object somewhere random: the free reset.

How to read it: In picture 3, grasp A's dots agree (sure), grasp C's agree on "unlikely", grasp B's are spread out — that spread is what makes B worth trying.

In plain words — the technical words, translated

reinforcement learning	learning by trying and seeing what happens
reward	a number for how good the outcome was (here: 1 if the object is in the gripper)
policy	the network that decides what to do next
reset, pseudo-reset	putting the world back for the next try; here done by the robot itself by dropping the object
ensemble	several networks trained side by side; their disagreement shows what is still unknown
uncertainty bonus	extra preference for trying things the ensemble disagrees about
SAC	a standard reinforcement-learning method, used here to learn where to drive
relabelling	re-scoring past drives with the grasp model's latest opinion, so each drive teaches more

Everyday analogy — a toddler in a playroom

Try, feel, toss, try again — and go for the puzzling blocks. *Where it breaks:* a toddler also learns from a parent's help, and a robot practising alone can break things; ReLMM still needed someone to change batteries and free the odd stuck object.

14.7.2 From the story to the formulas: why each piece of maths is needed

Question 1 (moment 2): "Did the grasp work?" Just one bit:

$$r_g = 1 \text{ (object in the gripper)} \qquad \text{or} \qquad r_g = 0 \text{ (not)}.$$

Why needed: this is the label, and the world produces it for free.

Question 2 (moment 3): "How wrong was each predictor?" A predictor that said "this grasp works with probability p" is scored by cross-entropy against what happened:

$$\text{loss} \;=\; -\big[\, r_g \log p + (1 - r_g) \log(1 - p) \,\big].$$

Numbers: it said $p = 0.9$ and the grasp worked: loss $= -\log 0.9 \approx 0.11$ (small). It said 0.9 and the grasp failed: loss $= -\log 0.1 \approx 2.3$ (large) — a big correction.

Question 3 (moment 3): "Which grasp should I try next?" Each candidate grasp is scored by what the six predictors think:

$$\text{score} \;=\; \alpha \times \underbrace{\text{their average}}_{\text{likely to work}} + \beta \times \underbrace{\text{their spread}}_{\text{they disagree}}.$$

Numbers ($\alpha = \beta = 1$): grasp A, average 0.80, spread 0.02: score 0.82. Grasp B, average 0.53, spread 0.29: score 0.82 — a tie, although B is less likely to work, because trying B teaches more. Grasp C, average 0.10, spread 0.02: score 0.12, tried least often (about 20%). With $\beta = 0$ the robot would only repeat what it already believes.

Question 4: "How do I turn scores into a choice?" Not always the top score — some randomness keeps it exploring: grasps are picked with probability growing with exp(score).

Question 5: "How is driving rewarded?" Every driving step costs 1, and a successful grasp pays it back:

$$r_n \;=\; r_g - 1 \qquad \text{(with relabelling: } r_n = \text{predicted chance of a grasp from here} - 1\text{)}.$$

Why this form: the -1 per step rewards reaching a graspable object *quickly*; relabelling gives credit to a step that got close even if no grasp was tried.

Table 14.7. ReLMM: the story and the maths, side by side.

In the story	In the maths	Why it is needed
"is it in my hand?"	$r_g \in \{0, 1\}$	the free label
"correct each predictor"	cross-entropy with r_g	learn from every try
"likely to work, or worth learning about"	α·average + β·spread	choose the next grasp
"keep some randomness"	probability $\propto$ exp(score)	keep exploring
"each step costs, success pays back"	$r_n = r_g - 1$ (or relabelled)	learn where to drive
"toss it somewhere new"	pseudo-reset (no formula)	practise without a person

14.7.3 Background: reinforcement learning in five words

Reinforcement learning (RL) learns by trial and error. Five words are enough for this section (Figure 14.56):

Observation. What the robot senses now, e.g. a camera image.

Action. What it does: drive, turn, move the arm, close the gripper.

Reward. A number that says how good the outcome was: 1 for a successful grasp, 0 otherwise.

Policy. The rule, usually a network, that picks an action from an observation. Training improves the policy to collect more reward.

Episode and reset. One attempt at the task. Classic RL assumes that after each attempt the world is *reset* to a fresh starting state — objects put back, robot returned to start.

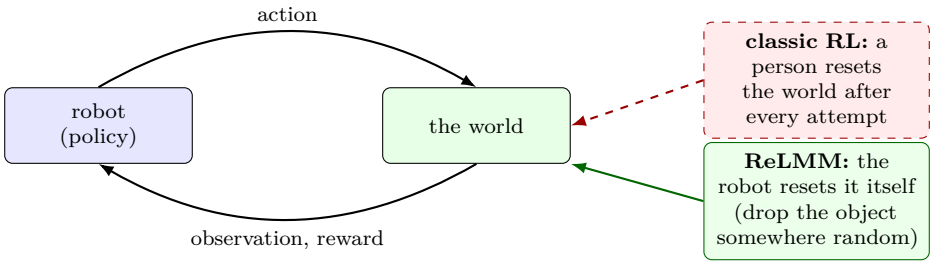


Figure 14.56. The reinforcement-learning loop. The robot acts, and the world answers with a new observation and a reward. On a real robot the hidden cost is the reset: in classic set-ups a person restores the scene after every attempt. That person is the "annotator" that ReLMM removes.

Ensembles and uncertainty. Train several networks (an **ensemble**) on the same data, each from a different random start. Where the data were clear, they agree; where the data were thin, they disagree. The spread of their answers (the standard deviation) is therefore a cheap measure of how unsure the model is.

Soft actor-critic (SAC). A standard RL algorithm for choosing continuous actions (such as "drive this far, turn this much"). For this chapter it is a black box that improves a policy from recorded observations, actions and rewards.

14.7.4 The problem

RL promises that a robot can learn from the results of its own actions. On real robots the promise usually breaks at two points: someone must *define and measure the reward*, and someone must *reset the scene* after every attempt. The second is the hidden annotator: hours of a person picking up objects and putting them back. ReLMM (Sun et al., 2022) removes both. Its task is *room cleanup*: a mobile robot with an arm must drive around a room, find objects on the floor and pick them up (the robot: slide 37, Figure 14.57).

In plain words — practise, check, and scatter the toys again

> The robot tries to pick something up. Whether it worked is easy to check — is the object in the gripper? — so the reward is free. Then, instead of waiting for a person to reset the room, the robot drives somewhere random and simply drops the object: dropping always

works, and it scatters the objects for the next attempt. The robot also tries grasps it is unsure about, because those teach it the most. It can practise like this for days.

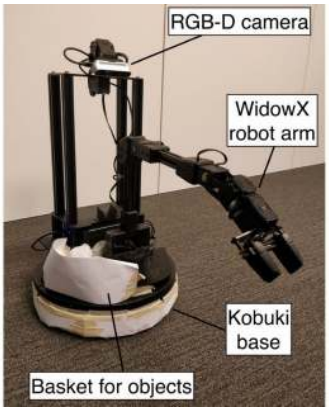


Figure 14.57. The ReLMM robot (Lecture 09, slide 37; from Sun et al., 2022): a Kobuki mobile base, a WidowX robot arm, an RGB-D camera on a mast, and a basket for collected objects.

Model at a glance: ReLMM — learning by doing, without a person in the loop Sun, Orbik, Devin, Yang, Gupta, Berseth & Levine, 2022

The free label	The outcome of acting: after a grasp, is the object in the gripper?
The free reset	The robot drives a random path and drops what it holds, re-scattering the room without a person.
What drives exploration	Uncertainty: an ensemble of grasp predictors, and a bonus for grasps they disagree about.
Structure	Two policies: grasping (where exactly to grasp) and navigation (where to drive), trained together.
What it buys	Room cleanup learned in about 40 hours of real-world practice, after a short warm-up phase.

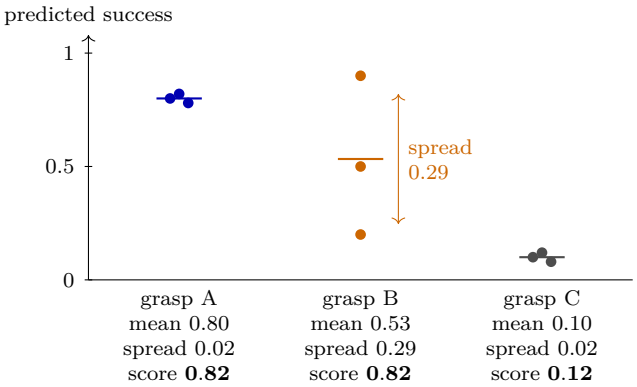
14.7.5 How it works, step by step

Learning to grasp. The grasp model $G(o_g, a_g)$ takes the camera view in front of the robot, o_g, and a candidate grasp a_g (where to put the gripper), and predicts the probability that the grasp succeeds. After the grasp, the robot checks whether the object is in its hand: success $r_g = 1$ or failure $r_g = 0$. The loss is the plain cross-entropy between the predicted probability and this outcome (slide 38: "the loss here is simply cross-entropy between predicted grasp and outcome"). The world is the labeller.

Choosing which grasp to try. ReLMM trains an *ensemble* of $M = 6$ grasp models $G^1, \ldots, G^M$ (Sun et al., 2022) and scores each candidate grasp by

$$\tilde{G}(o_g, a_g) \;=\; \alpha \underbrace{\mathbb{E}_i\big[G^i(o_g, a_g)\big]}_{\substack{\text{average predicted} \\ \text{success}}} + \beta \underbrace{\sigma_i\Big(G^i(o_g, a_g)\Big)}_{\substack{\text{disagreement between} \\ \text{ensemble members}}}, \tag{14.10}$$

where $\alpha, \beta \geq 0$ are weights, $\mathbb{E}_i$ averages over the ensemble members and σ_i is their standard deviation. Grasps with a high score are more likely to be tried (the robot samples grasps with probability proportional to $\exp \tilde{G}$). As slide 38 says, sampling *trades off high predicted success and high uncertainty.* Why reward disagreement? If the models disagree, whatever happens proves one side wrong — the attempt is informative either way (Figure 14.58).



Each dot is one ensemble member's prediction; the bar is their average. With $\alpha = \beta = 1$ the score is mean + spread.

A is a safe bet. C is a safe *no*. B is a coin toss the models argue about — and its score ties with A, because trying it will settle the argument. C is tried least (about 1 time in 5).

Figure 14.58. How the score of Eq. (14.10) ranks three candidate grasps (three ensemble members drawn for clarity; ReLMM uses six). Standard deviations are computed over the members: for B, the values $0.9, 0.2, 0.5$ have mean 0.533 and standard deviation 0.287.

The grasp score, slowly. If the formula above feels abstract, here it is one idea at a time, with the numbers of Figure 14.58.

Step by step — how the robot picks its next grasp

Step 1. Ask several advisers, not one. Instead of one network, the robot keeps several small networks (the *ensemble*) that were trained on slightly different data. Think of them as advisers who each guess "how likely is this grasp to work?". For grasp B three advisers say 0.9, 0.2 and 0.5.

Step 2. Average: "how good does it look?" Add the guesses and divide by how many there are: $(0.9 + 0.2 + 0.5)/3 \approx 0.53$. For A the average is 0.80, for C 0.10. If the robot looked only at this number it would always pick A.

Step 3. Spread: "how much do they argue?" Look at how far the guesses are from their average: 0.9 is 0.37 above, 0.2 is 0.33 below, 0.5 is 0.03 below. The typical size of these gaps (the *standard deviation*) is about 0.29. For A the advisers

almost agree $(0.80, 0.82, 0.78)$: spread 0.02. A large spread means *nobody really knows*.

Step 4. Score = "looks good" + "worth finding out".

$$\text{score} = \alpha \times \text{average} + \beta \times \text{spread}.$$

α and β are two dials: how much the robot cares about success right now, and how much about learning. With both set to 1: A $= 0.80 + 0.02 = 0.82$, B $= 0.53 + 0.29 = 0.82$, C $= 0.10 + 0.02 = 0.12$. Grasp B catches up with A *only* because of the argument about it.

Step 5. Pick with a little randomness. The robot does not always take the top score; it picks each grasp with a chance that grows with its score (proportional to e^{score}). Here that is about 40% for A, 40% for B and 20% for C. So the safe bet and the open question are tried equally often, and even the unlikely grasp gets an occasional try.

Step 6. Try it, and the argument settles. Suppose B is tried and the object is lifted (label 1). Each adviser moves towards 1, say half way: $0.9 \to 0.95$, $0.2 \to 0.6$, $0.5 \to 0.75$. Now the average is 0.77 and the spread only 0.14. Had it failed (label 0) they would move towards 0: $0.45, 0.10, 0.25$, average 0.27, spread again 0.14. *Either way the spread halves*: the robot has learned something whatever the outcome (Figure 14.59). That is why disagreement is rewarded — a try where everyone already agrees would teach almost nothing.

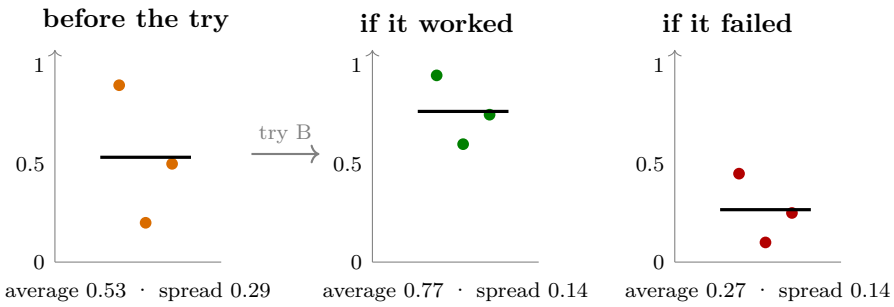


Figure 14.59. Why trying the grasp the advisers argue about is worth it. Left: three advisers' guesses for grasp B (the bar is their average). Middle and right: after one try, each adviser moves half way towards the outcome. Whether the grasp worked or failed, the guesses move closer together — the spread halves — so the robot learns either way.

How to read it: Compare how far apart the three dots are before and after: that shrinking gap is "information gained".

Everyday analogy — choosing where to eat

Your friends all agree the pizza place is good (high average, small spread): a safe choice. They argue fiercely about the new noodle bar — two love it, one hated it (middle average, large spread): going there once will tell you who was right, so it is worth a visit. Everyone agrees the kebab shop is bad (low average, small spread): you rarely bother. *Where it breaks:* if all your friends share the same blind spot, they agree even when they are all wrong, and the spread no longer shows what you do not know.

Resetting without a person. After a grasp attempt, ReLMM runs a *pseudo-reset* (slide 39):

1. check whether an object is in the hand;
2. drive a random path;
3. drop the object;
4. drive another random path.

This works because of a simple physical fact the lecture highlights (slide 45): *dropping is easier than grasping, and generates randomness.* Dropping never fails, and wherever the object lands becomes a new, random practice situation. So the robot keeps producing varied training data with nobody in the room.

Learning where to drive. Grasping only works if the robot first drives to an object. The navigation policy (trained with SAC) is rewarded through the grasping module, in a nested loop (slide 40, Figure 14.60):

1. take a navigation step;
2. if a grasp now looks likely to succeed, try it;
3. reward the navigation step. Each step costs -1, and a successful grasp cancels it: $r_n = r_g - 1$. So the policy learns to reach graspable objects in as few steps as possible. With *relabelling*, the reward instead uses the *expected* grasp success, $r_n = p[\text{grasp} \mid o] - 1$, computed from the grasp ensemble for every stored observation — so a step that led to a good grasping position earns credit even when no grasp was tried;
4. update the navigation policy with SAC.

Relabelling squeezes more learning out of each drive: the same recorded experience is re-scored with the current grasp model.

14.7.6 The two algorithms, line by line

The lecture shows ReLMM as two short algorithms (slides 37 and 40). They are the whole system in pseudocode, and they fit together like two loops, one inside the other (Figure 14.61):

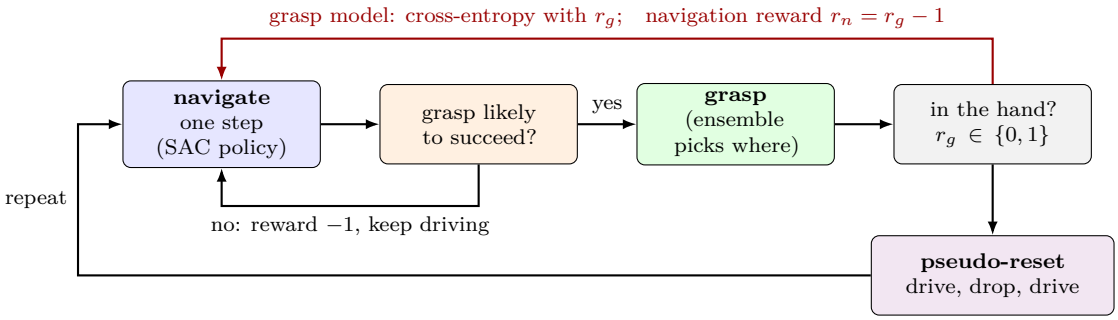


Figure 14.60. One cycle of ReLMM (after Lecture 09, slides 38–40). Every signal is produced by the robot itself: the grasp outcome labels the grasp model and rewards navigation, and the pseudo-reset prepares the next attempt.

- **Algorithm 2 (the outer loop) is the robot's day**: drive a step, maybe try to grasp, reward the driving, reset, repeat.
- **Algorithm 1 (the inner loop) is one grasping session**: at the current spot, try up to a few grasps, learn from each outcome, and report back whether one worked.

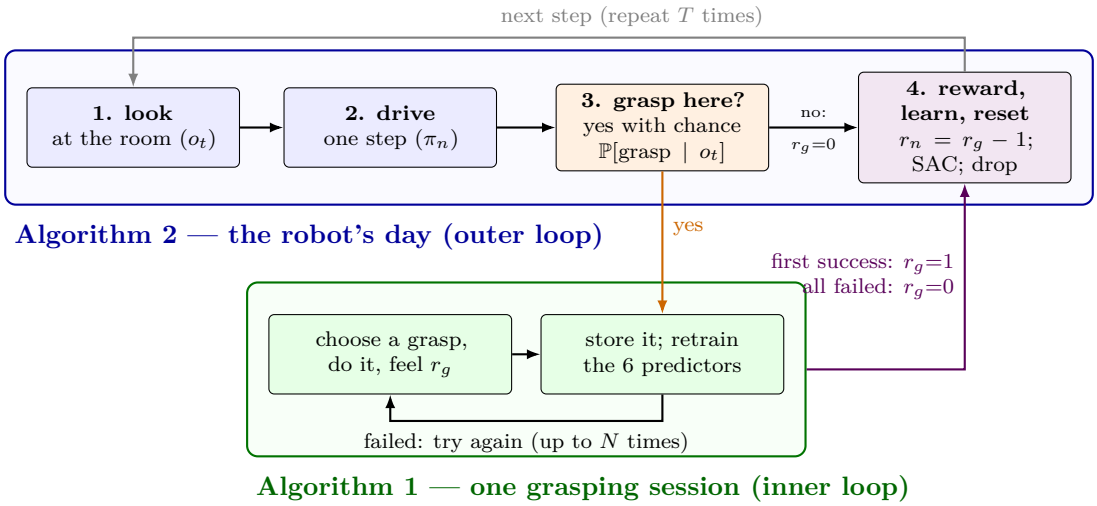


Figure 14.61. How ReLMM's two algorithms fit together. The outer loop (blue, Algorithm 2) drives the robot. When a grasp looks likely from where it stands, it calls the inner loop (green, Algorithm 1), which tries grasps and reports success (1) or failure (0). That report becomes the reward for the driving step.

How to read it: Find the single arrow from the green box to the violet one: the grasp outcome r_g is the only thing the grasping session hands back to the navigation.

Algorithm 1: one grasping session. Its inputs (Figure 14.62) are the $M = 6$ grasp predictors $G^1, \ldots, G^M$; the grasp memory $\mathcal{D}_g$ (every try so far, kept for retraining); N, how many tries are allowed at this spot; and a switch *st* ("stationary") for practice mode.

Algorithm 1 TrainGrasp($G^1, ..., G^M, \mathcal{D}_g, N, st$)

1: **for** t = 0, . . . , N steps **do**
2: Get grasp observation $\tilde{o}$.
3: Sample $a_g \sim \pi_g(\cdot|\tilde{o})$. // see Equation 2
4: Perform grasp a_g, receiving $r_g = 0$ or 1.
5: Store $(\tilde{o}, a_g, r_g)$ in $\mathcal{D}_g$.
6: Update $G^1, ..., G^M$ on $\mathcal{D}_g$
7: **if** st =True, Drop object if held.
8: **elseif** $r_g = 1$ and st =False, **return** r_g
9: **return** 0

Figure 14.62. Algorithm 1, TrainGrasp, as shown in the lecture (Lecture 09, slide 37; from Sun et al., 2022).

line	says	in plain words
1	for N steps	up to N tries at this spot
2	get grasp observation $\tilde{o}$	look closely at the object in front (camera view)
3	sample $a_g \sim \pi_g(\cdot \mid \tilde{o})$	choose a grasp with the score rule of the previous pages: likely to work, or worth finding out
4	perform a_g, receive $r_g = 0$ or 1	do it, and feel whether the object is in the gripper — the free label
5	store $(\tilde{o}, a_g, r_g)$ in $\mathcal{D}_g$	write "what I saw, what I did, what happened" into memory
6	update $G^1, \ldots, G^M$ on $\mathcal{D}_g$	retrain the six predictors on the memory (cross-entropy with the outcomes)
7	if st: drop object if held	practice mode: put it straight back down and try again at the same spot
8	else if $r_g = 1$: return r_g	normal mode: first success ⇒ stop and report 1
9	return 0	all tries failed ⇒ report 0

Why practice mode? At the very start the predictors know nothing, and a robot that cannot grasp at all gets no useful reward for driving. So first the robot practises on one object in place: grasp, drop it right there, grasp again — the *stationary curriculum*.

Algorithm 2: the robot's day.

Algorithm 2 ReLMM (with Stationary Curriculum)

1: Hyperparameters: M, N_{st}, N_{grasp}, *relabel*
2: Init: function estimators $\pi_n, G^1, \ldots, G^M$.
3: Replay buffers $\mathcal{D}_n = \{\}, \mathcal{D}_g = \{\}$
4: TrainGrasp($G^1, .., G^M$, D_g, N_{st}, True)
5: **for** $t = 0, \ldots, T$ steps **do**
6: | Get navigation observation o_t
7: | Sample $a_n \sim \pi_n(\cdot|o_t)$ and perform a_n
8: | **if** *uniform*() $\leq \mathbb{P}[\text{grasp}|o_t]$ **then**
9: | | r_g =TrainGrasp(G^1,.., G^M,D_g,N_{grasp},False)
10: | **else** $r_g = 0$
11: | **if** *relabel* **then**
12: | | Navigation reward $r_n = \mathbb{P}[\text{grasp}|o_t] - 1$
13: | **else** $r_n = r_g - 1$
14: | Get next navigation observation o_{t+1}
15: | Store (o_t, a_n, r_n, o_{t+1}) in $\mathcal{D}_n$.
16: | Update π_n with $\mathcal{D}_n$ using SAC.
17: | Pseudo-reset
18: **end for**

Figure 14.63. Algorithm 2, ReLMM with the stationary curriculum, as shown in the lecture (Lecture 09, slide 40; from Sun et al., 2022).

line	says	in plain words
1	hyperparameters $M, N_{\text{st}}, N_{\text{grasp}}$, relabel	settings: 6 predictors, number of warm-up tries, tries per grasping session, relabelling on or off
2	init $\pi_n, G^1, \ldots, G^M$	start with an untrained driving policy and six untrained grasp predictors
3	$\mathcal{D}_n = \{\}$, $\mathcal{D}_g = \{\}$	two empty memories: one for driving, one for grasping
4	TrainGrasp($\ldots, N_{\text{st}}$, True)	warm-up: Algorithm 1 in practice mode (learn a basic grasp first)
5	for T steps	the rest of the day, step by step
6	get navigation observation o_t	look at the room
7	sample $a_n \sim \pi_n(\cdot \mid o_t)$ and perform	choose a driving step with the policy and drive it

Lines 8–17, the rest of each step:

line	says	in plain words
8	if uniform() $\leq \mathbb{P}[\text{grasp} \mid o_t]$	roll a die: try grasping with a chance equal to how likely the predictors think a grasp from here would work — mostly where it looks promising, occasionally elsewhere
9	$r_g =$ TrainGrasp$(\ldots, N_{\text{grasp}}, \text{False})$	run a grasping session (Algorithm 1, normal mode)
10	else $r_g = 0$	no grasp tried this step
11–13	$r_n = \mathbb{P}[\text{grasp} \mid o_t] - 1$ (relabel) or $r_g - 1$	reward for this driving step: every step costs 1; a grasp (actual, or expected with relabelling) pays it back
14	get o_{t+1}	look again after the step
15	store (o_t, a_n, r_n, o_{t+1}) in $\mathcal{D}_n$	remember "where I was, what I did, what it earned, where I ended up"
16	update π_n with $\mathcal{D}_n$ using SAC	improve the driving policy from that memory
17	pseudo-reset	if holding something: drive a random path and drop it — a new situation, no person needed

Worked Example — a few steps of the robot's day, traced

Step 1. The robot is in an empty corner. The predictors think a grasp from here would work with chance $\mathbb{P} = 0.05$. The die shows $0.62 > 0.05$: no grasp (line 10, $r_g = 0$). Reward: $r_n = 0 - 1 = -1$ (with relabelling $0.05 - 1 = -0.95$).
Step 2. It drives next to a toy: $\mathbb{P} = 0.55$. The die shows $0.31 \leq 0.55$: a grasping session starts (line 9). Algorithm 1: try 1 fails ($r_g = 0$, stored, predictors retrained); try 2 succeeds ($r_g = 1$) $\Rightarrow$ report 1. Reward: $r_n = 1 - 1 = 0$ — the step that reached a graspable toy is not penalised.
After step 2. Pseudo-reset (line 17): it drives a random path and drops the toy. Both memories have grown, both networks were updated, and no person touched anything.
The lesson for the driving policy: steps that end near graspable objects cost nothing, all others cost 1 — so over many days it learns to head for objects quickly.

Results. After a short grasping warm-up (a "curriculum" of easy grasps), ReLMM learns navigation and grasping together in about 40 hours of real-world training, in several room layouts, and keeps improving beyond 50 hours (Sun et al., 2022). Honestly counted, "no human" still allows a little help: someone changes the battery every five hours or so and occasionally frees an object stuck in a corner. In the authors' simulated ablations, removing the grasp uncertainty bonus ($\beta = 0$) makes learning much worse, which shows that the ensemble's disagreement is doing real work.

Table 14.8. Three ingredients of system-level self-supervision (Lecture 09, slides 43–46).

Ingredient	Question to ask	Where it appears in this chapter
Paired modalities	Which other sensor saw the same world at the same time?	pose + RGB + depth → map labels (Frey et al., 2022); pose + camera + legs → traversability (Frey et al., 2023); RGB + thermal + depth → audio (Rivera Valverde et al., 2021)
Physical laws	What must be true in the world, whatever the network thinks?	depth, pose and flow must agree (Zou et al., 2018); dropping is easier than grasping and creates randomness (Sun et al., 2022)
Uncertainty	Without curated data, what does the system not know yet?	where to go: map uncertainty (Zurbrügg et al., 2022); what is unknown: reconstruction error (Frey et al., 2023); which action teaches most: ensemble disagreement (Sun et al., 2022)

In plain words — try it interactively

Grasp Lab — watch the robot reset its own room by dropping objects, move the α and β sliders of the ensemble score, try grasps and see each outcome train the six predictors until they agree, and re-score a drive with relabelling.
`https://claude.ai/artifact/M3rsYwtqtbcAw4Bntv6AcZ`
(Private until shared from the page's Share menu. Add `#step3`, or any step number, to the link to open that step.)

Failure Modes

Resets that are not random enough. The random drop only reaches places the robot can drive to; objects that end up in corners are never practised on. **Overconfident ensembles.** If the members are too similar, they agree even where they are all wrong, the bonus vanishes, and exploration stops. **Relabelling can flatter itself.** Navigation is rewarded for *predicted* grasp success; if the grasp model is too optimistic about some spot, navigation learns to drive there. **Safety.** A robot that practises on its own can break things; "no external supervision" does not mean "no external risk".

14.8 The Big Picture: Three Ingredients of System-Level SSL

The lecture ends (slides 43–46) by sorting the five systems by three ingredients. Each system uses one or two of them.

Paired modalities (slide 44). Every system that trains one signal from another needs a rig: camera and pose for the map, camera and legs for traversability, microphones and cameras

for the audio student. This is the observation of slide 5 — rigs are the normal case — turned into supervision.

Physical laws (slide 45). Some supervision comes from rules the world enforces: rigid geometry ties depth, pose and flow together; gravity makes dropping easy. These rules do not drift the way a learned teacher does, so the systems rely on them most where their learned parts are least trustworthy — while still masking out the places where a rule does not apply (moving cars).

Uncertainty (slide 46: "in absence of curated data, uncertainty estimation is important"). A curated dataset contains only examples someone chose. A robot in the world meets things its training never covered, so it must know what it does not know: to decide *where to go* (curiosity planning), *what is unknown* (WVN's familiarity test) and *which action is informative* (ReLMM's ensemble). At system level, uncertainty is the link between learning and acting.

Research Frontier

Where does system-level SSL stop being self-supervised? SLAM systems, physics simulators and pre-trained teachers all contain human-engineered knowledge; the definition of slide 6 (no external input or labelling *while the system runs*) allows them, a stricter reading would not. Open problems include keeping maps and replay buffers useful over months of operation, combining sensors that disagree about timing, learning without rigid geometry (soft robots, fluids), and benchmarks that score a *learning loop* rather than a fixed model.

14.9 Chapter summary

Every system in this chapter answers the same question differently: *what does this machine already measure twice?* Once you have that second measurement, the rest is ordinary supervised learning.

Table 14.9. The five systems, side by side. The middle column is where the supervision comes from; it is different in every row, while the training machinery is ordinary.

System	Where the free label comes from	What breaks it
Map-based adaptation (§14.3)	a fused 3-D map out-votes the single frames that built it	mistakes all views share survive the vote and are taught back
Geometric consistency (§14.4)	depth, pose and flow must agree: any two fix the third	moving objects and blank surfaces; scale is never recovered
Own body (§14.5)	the legs' velocity tracking, painted onto earlier images	labels only where the robot walked; familiar-looking danger
Cross-modal distillation (§14.6)	three cameras voting on the same moment, for the microphones	errors the teachers *share* are inherited silently
Autonomous RL (§14.7)	the observed outcome of a grasp; dropping as a free reset	resets that are not random enough; unsafe exploration

Two patterns run through all five rows.

Redundancy is the resource. None of these systems gets information from nowhere. Each uses the fact that the world was measured more than once — across frames, across sensors, across time, or across the depth/pose/flow triangle — and that measurements of one fact must agree. Where they disagree, there is a gradient.

The typical failure is a shared error. Combining measurements removes mistakes they do *not* share. It does nothing about mistakes they all share, which is why every row of Table 14.9 can fail the same way: the system becomes confidently, consistently wrong, and has no way to notice.

Key takeaway — the question to carry out of this chapter

Ask of any machine: *what does it already measure twice?* A map and a frame. Depth and flow. A camera and a pair of legs. A microphone and a thermal camera. A predicted grasp and a closed gripper. Wherever two independent measurements of one fact exist, one of them can label the other — and no annotator is needed.

14.10 Check Your Understanding

Check Your Understanding

Level 1 (recall).

1. State the lecture's definition of system-level self-supervision and name which slide introduced it.
2. Paired data for conditional generation is "strictly rather semi-supervised." Under what condition do pairs count as SSL?
3. In the Frey et al. system, name the four steps of the loop that turns per-frame predictions into adapted weights.

Level 2 (comprehension).

4. Why are map-rendered pseudo-labels usually cleaner than any single frame prediction they were fused from? Which kind of mistake can the map *not* fix?
5. Slide 14's plot shows random viewpoints *hurting* adaptation. Give a mechanism for why.
6. In DF-Net, what role do the "invalid masks by rigid flow" play in the loss (14.3)?
7. In WVN's traversability loss (14.7), what would go wrong if the weight $(1 - c(\boldsymbol{f}_n))$ were replaced by a constant 1? And if the whole first term were deleted?

Level 3 (application).

8. Redo the worked example of §14.4 with depth $D_A = 8\,\text{m}$ instead of 4 m. Where does the pixel land in frame B, and how does the answer relate to Figure 14.33(b)?

9. A planetary rover with stereo cameras and a robot arm should learn which rocks it can pick up. Which of the five mechanisms apply, and which labels are free?
10. MM-DistillNet trains the student with both focal and MTA losses. Argue what each adds that the other cannot.
11. Compute the focal loss ($\alpha = 0.25$, $\gamma = 2$) and the cross-entropy for $p_t = 0.5$. By what factor does the focal factor shrink this example?

Level 4 (analysis).

12. Show that the DF-Net losses are unchanged if all depths and the camera translation are multiplied by the same number $s > 0$, and name one way a system could recover absolute scale without labels.
13. ReLMM's score (14.10) adds the ensemble standard deviation. Contrast this with Thompson sampling over the same ensemble (pick one member at random and follow it) — when do they choose differently?
14. The chapter says physical laws "do not drift". Give a case within one of the five systems where the law is nevertheless the wrong model, and how the system defends itself.

Level 5 (synthesis).

15. For each ingredient of Table 14.8, describe the smallest sensor set that provides it. Can a single camera plus wheel odometry provide all three?
16. Argue for or against: "a system that uses a pre-trained RGB detector as its teacher is not really self-supervised." Where do you draw the line?

Solution sketches.

- *Q1.* Learning from all available data of the system, without any external input or labelling; slide 6 (repeated on slide 43). *Q2.* When the pairs are mined without manual effort — augmentations, super-resolution, synchronised sensor rigs. *Q3.* Predict each frame; fuse into a 3-D voxel map using the poses; render pseudo-labels from the map; fine-tune with a replay buffer.
- *Q4.* Each voxel collects votes from many views; a mistake made in a few views (a reflection) is out-voted. The map cannot fix a mistake made in *all* views. *Q5.* Random viewpoints cover the room unevenly: few, similar views per place make a weak vote, so map errors survive and are trained in, and the training mix is unbalanced. *Q6.* They switch the consistency terms off on pixels where the static-scene rule is false (moving objects, occlusions, pixels leaving the view), so the networks are not taught wrong geometry. *Q7.* With weight 1, familiar-looking unwalked ground (the leaves next to the path) is pushed to 0 at full strength: the robot learns to fear everything it has not literally stepped on. Without the term, the network never sees a negative example and may call obstacles such as tree trunks traversable.

- *Q8.* $P = (1.6, 0.32, 8)$, $RP + \boldsymbol{t} = (2.1, 0.32, 8)$, $\boldsymbol{p}_B = 500 \cdot (2.1/8,\ 0.32/8) = (131.25,\ 20)$: a shift of 31.25 px, half of before, as $\Delta x = ft/Z$ predicts — twice as far, half the shift. *Q9.* Interaction (grasp outcomes label the rock images, as in ReLMM) and geometry (stereo gives depth, and with a known camera baseline also absolute scale). Free labels: grasp success from the gripper, depth from stereo. *Q10.* Focal says *which boxes* are right; MTA says *where to attend* in intermediate layers, a denser signal that helps the student learn useful internal features. *Q11.* Cross-entropy $-\log 0.5 = 0.693$; focal $= 0.25 \cdot 0.25 \cdot 0.693 \approx 0.043$, i.e. 16 times smaller $(1/(\alpha(1-p_t)^2))$.
- *Q12.* $R(sP) + s\boldsymbol{t} = s(RP + \boldsymbol{t})$ and projection divides by depth, so every predicted pixel position is unchanged, hence all losses. Scale from: a stereo pair with known baseline, wheel odometry or an IMU, a known object size. *Q13.* Thompson sampling picks at random in proportion to how often each candidate wins under a randomly drawn member; the score picks by mean plus bonus. They agree when one grasp is best on both; they differ when candidates trade mean against disagreement (in Figure 14.58, Thompson would try B only when the member saying 0.9 is drawn). *Q14.* Rigid geometry is the wrong model for moving cars and crowds; DF-Net masks those pixels, but the masks are themselves computed from the (learned) predictions, so the defence is imperfect.
- *Q15.* Paired modalities: any two synchronised sensors. Physical laws: a moving camera (geometry) — plus an actuator for laws like "dropping is easy". Uncertainty: a single network ensemble or autoencoder on any sensor. Camera plus odometry: modalities yes, geometry yes (odometry even fixes scale), uncertainty yes; laws that need acting, no. *Q16.* Reasonable answers: the teacher was trained with labels once, on a different task, but the new task (audio) receives no labels, so by the slide-6 definition — no external input *while the system runs* — it counts; a stricter view counts every label ever used.

Concept Map — how this chapter connects

Backward links. Pseudo-labels and teacher–student distillation from *Chapter 12* reappear with a different teacher: a map (§14.3) or another sensor (§14.6). DINO features from *Chapter 11* are used frozen in WVN as a general description of terrain. Autoencoders from *Chapter 4* become WVN's familiarity test. Cross-entropy from *Chapter 2* is the grasp loss of ReLMM and the base of the focal loss; the KL divergence from Chapter 2 is the core of the MTA loss.

Forward links. Chapter 15 asks *why* SSL works; this chapter supplies deployed systems that improve with no labels at all, and their typical failure — shared, confident errors. Chapter 16 compares method families; Table 14.8 adds the system-level view. Chapter 13's world models are the close cousin: both close the loop between what a model learns and what an agent does.

Part III

Research Perspective

CHAPTER 15

Why Does Self-Supervised Learning Work?

When a measure becomes the theory, check the theory.

In this chapter

15.1 How to Read This Chapter: Four Levels of Evidence 709
15.2 The Central Dogma and When It Breaks . 710 • Formalizing the dogma • When the dogma breaks
15.3 Why Joint-Embedding Works ..... 715 • View choice as the whole game • Alignment and uniformity: the empirical picture • When it fails
15.4 Why Masked Reconstruction Works 719 • The information-bottleneck reading • What pixel regression forces the encoder to keep — and where that hurts
15.5 Why Self-Distillation Without Negatives Works — Sometimes 722 • The three levers • The competing explanations
15.6 A Collapse Taxonomy Across All Book Families 726 • Complete collapse and the partial kind you will actually meet
15.7 Scaling Beats Architecture at the Far End ..... 729 • The data-engine argument
15.8 Landscape of the Book's Methods 732
15.9 Summary ..... 733 • The chapter on one page

Chapter roadmap

Before you start. You need the methods themselves, not their theory: masked reconstruction (Chapter 5), contrastive learning and the InfoNCE loss (Chapter 9), the negative-free methods and what "collapse" means (Chapter 10), self-distillation with an EMA teacher (Chapter 11), and the two information quantities of Chapter 2 — entropy and mutual information. If any of those is hazy, read that chapter's summary first; this chapter argues *about* them and defines almost nothing new.

The idea in one sentence. Self-supervised learning works far better than anybody can currently explain, so the useful skill is not memorising one explanation but knowing, for each explanation, how much evidence stands behind it and what would prove it wrong.
What we cover.

- *An evidence scale* (Section 15.1). Four labels — observation, theorem, hypothesis, open question — with a marker for each. Every claim in the chapter carries one. This is the chapter's main tool, so it comes first.
- *The central dogma made precise* (Section 15.2). "Anything better than chance is a learning signal" becomes two conditions you can check, and every failure in the chapter is one of the two breaking.
- *Why joint-embedding works* (Section 15.3). The augmentations, not the loss, decide what the network is allowed to ignore. We draw that argument geometrically.
- *Why masked reconstruction works* (Section 15.4). The bottleneck story, and the one number it fails to predict: the linear-probe gap.
- *Why negative-free distillation works — sometimes* (Section 15.5). Four explanations that contradict each other, all still alive.
- *A collapse taxonomy* (Section 15.6) and *the scaling argument* (Section 15.7), where the question quietly changes from "which loss?" to "which corpus?"

Afterwards you will be able to (i) label any "why SSL works" claim you meet as observation, theorem, hypothesis or open question, and say why; (ii) state, for each of the five main explanations, the experiment that would falsify it; (iii) name the degenerate solution each family is secretly spending its recipe to avoid; and (iv) read a new SSL paper's discussion section and tell which sentences are load-bearing evidence and which are field folklore.

Every chapter of Part II ended with a "why does this actually work?" subsection, and each one gave a partial answer. This chapter collects those answers, confronts them with each other, and asks the harder question per family: not just *why* the method works, but *when it fails*, and *which of the explanations you have memorized deserve the trust you put in them*. The honest answer is uncomfortable. Self-supervised learning is an engineering triumph resting on a thin theoretical base: a handful of genuine theorems, a large body of empirical regularities, and a long tail of hypotheses that the field repeats with more confidence than the evidence supports.

We proceed family by family — the central dogma and its information-theoretic formalization, joint-embedding methods, masked reconstruction, self-distillation without negatives — separating at each step what is *theorem*, what is *empirical observation*, and what is *hypothesis*.

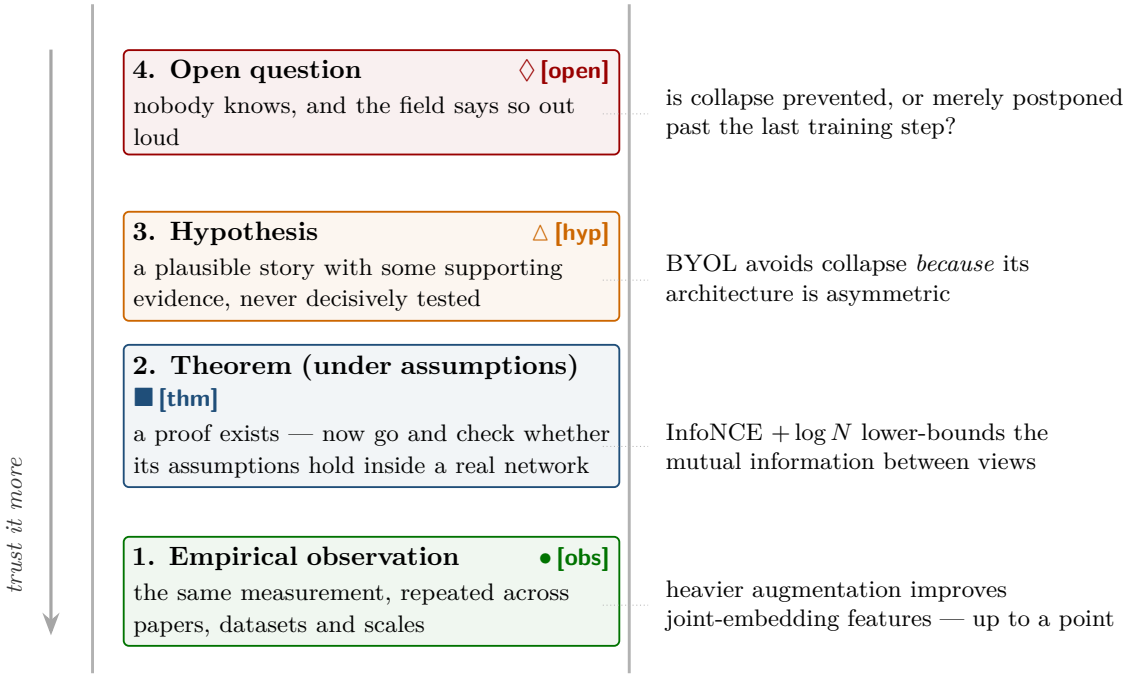


Figure 15.1. The evidence scale used throughout this chapter, with the marker that tags every claim from here on. Note the *direction*: empirical observation sits on the ground because it is the most trustworthy, not the least. Higher rungs are progressively less load-bearing. A theorem is only as strong as its assumptions, and a hypothesis is a research programme, not a fact.

How to read it: Read it bottom-up. Fix the four markers — • [obs] ■ [thm] △ [hyp] ◊ [open] — in your memory now; every later claim in the chapter wears one, and the colour alone tells you how hard to lean on it.

15.1 How to Read This Chapter: Four Levels of Evidence

Before any explanation, the tool for judging explanations. Every claim in this chapter carries one of four markers. The markers are small, they are coloured, and they never change meaning; once you have learned the four, you can skim the chapter and see its epistemic shape at a glance.

The ordering is deliberate and it is the opposite of the one you were taught. In theoretical computer science a theorem sits at the top. In empirical machine learning, a replicated measurement at the scale you care about beats a theorem whose assumptions fail inside a Vision Transformer. A theorem about two-layer linear networks is a beautiful object and tells you very little about a ViT-L trained for 800 epochs.

New words used from here on — the four markers

• [obs] **Empirical observation.** Somebody measured it, more than once, and it held. Example: MAE's linear probe trails DINO's at matched model size.

■ [thm] **Theorem under assumptions.** Somebody proved it, and the proof needs

assumptions — infinite negatives, a linear network, a known data distribution. The theorem is certain; its *applicability* is the thing to argue about.

△ [hyp] **Hypothesis.** A story that explains the observations and has survived some ablations, but has no proof and no decisive experiment. Most "why it works" paragraphs in SSL papers live here.

◇ [open] **Open question.** The field has looked and does not know. Naming these honestly is what separates a research chapter from a summary.

Common Confusion

"It's only a hypothesis" does not mean "it's probably false." The markers rank *how much the claim has been tested*, not how likely it is to be true. The asymmetry story for BYOL (Explanation 1 in Section 15.5.2) is a hypothesis and is also, in this author's judgement, most likely right. The marker tells you how much weight the claim can carry in an argument — whether you may build a design decision on it, or must run the experiment yourself.

Key takeaway — what the markers are for

Four markers, one per claim: ● [obs] measured repeatedly, ■ [thm] proved under assumptions, △ [hyp] plausible but untested, ◇ [open] unknown. When a paper's discussion section makes a claim you cannot mark, that is the claim to distrust. The rest of this chapter is an exercise in marking.

15.2 The Central Dogma and When It Breaks

Chapter 1 opened with the central dogma: *anything better than chance can be exploited as a learning signal.* The classical methods of Chapter 3 showed its shallow form — pairwise distances in pixel or feature space already support clustering and retrieval — and every method since has been a fancier way of cashing the same check. This section asks what that check is actually worth.

15.2.1 Formalizing the dogma

The dogma decomposes into two independent conditions, and it is useful to see them written down once, plainly:

Key Concept — Two conditions for self-supervision to work

Let X be raw data and T a pretext task defined on X alone (reconstruct a masked patch, predict an augmentation's parameter, agree with another view). A self-supervised method provides useful training signal if and only if both:

1. **The data has exploitable structure.** The images (or text, or sensor streams) occupy a set of much lower "effective dimension" than the raw space, so that T can be solved better than chance at all. Independent noise offers nothing: no pretext task on white noise has a solution better than the trivial one.
2. **Solving T requires downstream-useful features.** There must be an algorithm for T whose learned intermediate features are the features a downstream task (classification, segmentation, pose) also needs — *and* gradient-based training must find that algorithm rather than another one.

Condition 1 is usually satisfied for natural data and is rarely the binding constraint. Condition 2 is where every failure mode in this chapter lives: it fails silently, producing features that solve T beautifully and transfer nowhere.

In plain words

Two things have to be true. First, the data must *have* a pattern — otherwise there is nothing to find, and no clever loss invents one. Second, the shortest route to solving your made-up task must run *through* that pattern. If the network can pass the made-up task by some cheaper trick, it will, and you will have trained a trick detector. The second condition is the dangerous one, because the training loss looks identical either way.

Figure 15.2 draws the pair as two gates in series. Data enters on the left; useful features come out on the right only if both gates open. Every method in this book is an argument that its gates open.

Condition 2 has a clean information-theory reading ■ [thm] (an established framework rather than a single theorem). Let V_1, V_2 be two views (two augmentations, two modalities, or past and future pieces of a sequence), and Y whatever the downstream task cares about.

- **The claim** (contrastive predictive coding, van den Oord et al. 2018b): training an encoder to maximise a lower bound on $I(V_1; V_2)$ makes it capture *everything the views share*.
- **So**: if everything about Y is in what the views share — $I(V_1; Y) = I(X; Y)$, up to augmentation — then maximising view-to-view information also captures Y.

Every practical method's weak spot is that "if":

- views *share* things the task does not want (textures, backgrounds);
- views *do not share* things the augmentations destroyed (colour, fine texture, exact geometry) — which some tasks do want.

Figure 15.3 draws that argument, and — deliberately in the same frame — the objection to it. Read the left panel first; it is the story the field tells. Then read the right panel, which is why the story is contested.

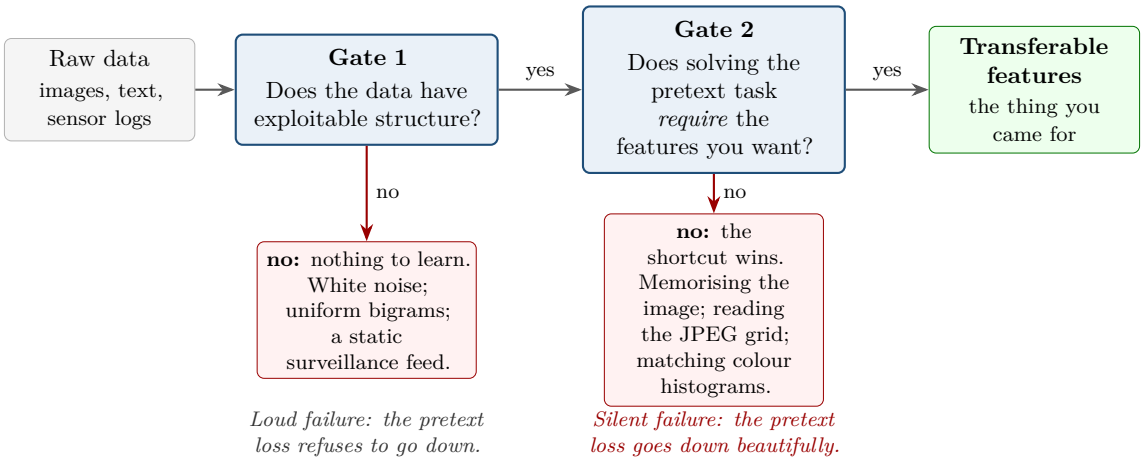


Figure 15.2. The central dogma as two gates in series. Gate 1 asks whether the data carries structure at all; gate 2 asks whether the pretext task's *easiest* solution happens to need that structure. Both must open. The asymmetry at the bottom is the point of the whole chapter: gate 1 fails loudly (the loss will not move) while gate 2 fails silently (the loss moves perfectly and the features are worthless).

How to read it: Follow the top row first, then look down. The red boxes under gate 2 — not gate 1 — are where real projects die.

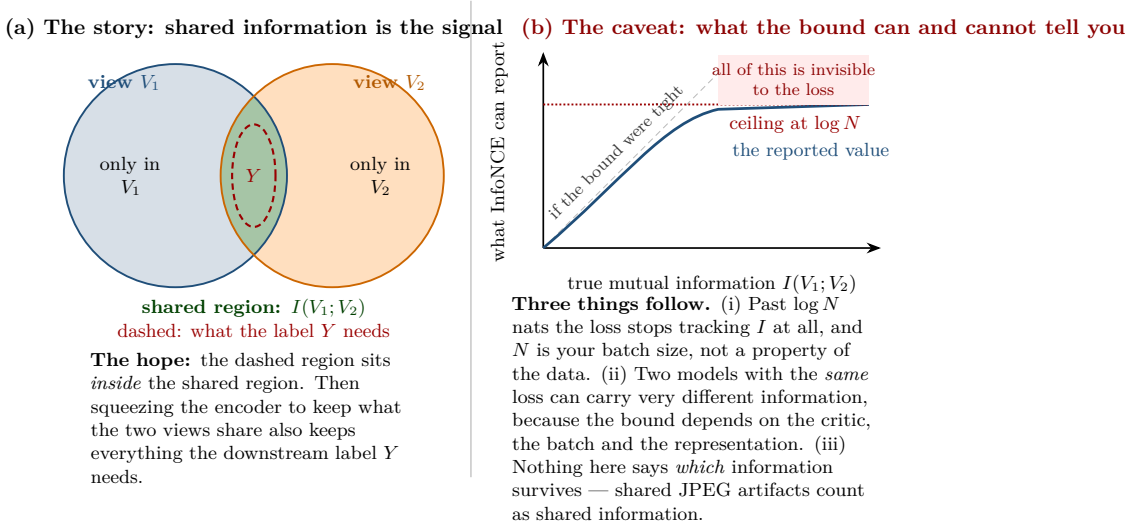


Figure 15.3. The information-theoretic explanation of joint embedding, and next to it the reason to distrust it. Panel (a): if everything the label needs lives in what the two views share, then maximising shared information is a sound proxy for the thing you want. Panel (b): the quantity a contrastive loss actually reports is a *lower bound* that saturates at $\log N$ for a batch of N (Poole et al., 2019), and a bound being pushed up is not the same as the underlying quantity going up.

How to read it: The dashed red region in (a) is the whole argument; the shaded red region in (b) is the whole objection. If you remember one thing: the saturation ceiling in (b) is set by your batch size.

The sharper version is the *InfoMin principle* (Tian et al., 2020): the best views *minimise* $I(V_1; V_2)$ while keeping the task-relevant information $I(V_1; Y)$ intact. More shared information is not always better. The sweet spot is where the views agree on *meaning* and disagree on everything else.

That explains two things:

- **Why stronger augmentation helps, then hurts** • [obs]. Augmentation strength is a dial on $I(V_1; V_2)$, and the best setting is somewhere in the middle.
- **The failure of Section 15.3.3.** If the augmentations never destroy some nuisance factor, the representation keeps it, and tasks sensitive to that factor stop tracking pretext performance.

The catch in "maximise mutual information". This is the part of the argument that is quoted least, and it matters most.

Mutual information between high-dimensional variables *cannot be measured accurately from a finite sample when it is large* — and large is exactly what training is trying to make it. Poole et al. (2019) showed that every practical estimate trades bias against noise, and the low-noise ones, InfoNCE included, can never report more than about $\log N$ for a batch of N. So "maximise InfoNCE" really means "push up a gauge that stops moving once the true value passes $\log N$". ■ [thm]

What does *not* follow — and this is the correction most readers need:

- a better InfoNCE score does *not* mean the representation holds more information;
- two models with the same InfoNCE score can hold very different amounts of *useful* information.

The score depends on the scoring network, the batch size and the representation itself, not only on what the representation knows. ■ [thm]

Common Confusion

"InfoNCE maximizes mutual information" vs. what is proven. What the theory gives is: (a) under specified assumptions, InfoNCE plus $\log N$ is a lower bound on MI (van den Oord et al., 2018b; Poole et al., 2019); (b) minimizing it therefore pushes the bound up, which pushes MI up only insofar as the bound is tight. Nothing in (a) or (b) constrains *which* information is preserved when optimizing it, nor says anything about representations trained this way being linearly separable or clustering nicely. Those two downstream properties — the ones the field actually cares about — have separate, weaker theoretical support and mostly rest on empirical observation.

Simplified intuition vs. more precise view

Simplified intuition. Two augmented copies of an image still show the same dog. A network that must tell matched copies apart from mismatched ones learns what makes a dog a dog, not which exact pixels were there.

More precise view. A network can equally solve the matching task by memorizing the two views' shared confound: the augmentation pipeline's resampling fingerprint, the compression artifacts, the color histogram of the same crop before augmentation. Whether the found solution is the semantic one depends on the augmentation distribution, the architecture, the batch size, and the optimizer state at the time the network commits to a strategy. The semantic-solution story is at best a hypothesis △ [hyp]about which minima gradient descent prefers, and partly an empirical observation • [obs]that with well-tuned augmentation, downstream accuracy is high. The theory that "MI optimization forces semantics" does not exist.

15.2.2 When the dogma breaks

With the two conditions named, the catalogue iteration becomes mechanical: every failure of self-supervision is one of the two conditions breaking.

No structure to exploit (condition 1 fails). Adversarially shuffled pixels destroy spatial locality, and a masked-reconstruction task on such data is solvable (the conditional distribution is still learnable) but teaches features that do not transfer to natural images • [obs]. Text token streams with artificially uniform bigram statistics break masked language modeling for the same reason. Low-entropy data — identical backgrounds in starfield images, static surveillance footage — produce trivial solutions to temporal pretext tasks.

Task solution decouples from downstream (condition 2 fails). The canonical example is color: Colorization (Zhang et al., 2016) and the split-brain variant (Zhang et al., 2017) train features heavily invested in chroma prediction, which is only loosely what classification needs. A subtler and more instructive example, from Chapter 4: a sufficiently deep autoencoder on ImageNet learns to *memorize* images into its code — reconstruction error drops to near zero while the learned representation is arbitrary — and all "why reconstruction teaches features" arguments applied to it are silently false • [obs]. Memorization is the identity-map solution of the pretext task, and it satisfies condition (1) richly while violating condition (2) completely.

The signal exists but the optimizer finds another basin. Self-distillation families (Section 15.5) illustrate the third failure: SimSiam (Chen and He, 2021) collapses outright when the stop-gradient is removed, despite the task (match the teacher's distribution) still having the same information structure. The problem is not information; it is that gradient

descent has a shorter path to a useless minimum than to a useful one. Whether you call this a break of the dogma or a break of the training procedure, the practical symptom is identical.

Failure Modes

The dogma fails silently. A pretext loss going down is necessary and nowhere near sufficient:

- The learned features may solve the task via memorization (Chapter 4) or via the augmentation pipeline's fingerprint (Section 15.3.3).
- Downstream accuracy can rise while individual downstream slices (rare classes, unusual poses) degrade, because the pretext task's information need not cover everything the evaluation averages over.
- A method can appear robust at one evaluation (linear probe) and be clearly broken at another (dense prediction), without anything about the training having changed — MAE's linear-probe gap (Section 15.4.2) is the standard cautionary example.

Key takeaway — the dogma, and what would refute it

The claim. Self-supervision works when the data has structure (gate 1) *and* the cheapest way to solve the pretext task runs through that structure (gate 2). ● **[obs]** for the pattern across methods; △ **[hyp]** for any particular claim that gate 2 is open in a particular pipeline.

What would have to be observed for this to be wrong. A pretext task trained on genuinely structureless data (independent noise, shuffled tokens with matched marginals) that nevertheless produced features transferring better than random initialisation — that would break gate 1. Or: a family of pretext tasks whose downstream transfer was *uncorrelated* with whether their solutions plausibly need semantic features, across many tasks and datasets — that would mean gate 2 is not the operative variable and something else is doing the work. Neither has been observed. The dogma is the safest claim in the chapter, which is also why it is the least informative: it tells you what to check, not what to build.

15.3 Why Joint-Embedding Works

Joint-embedding methods — SimCLR, MoCo, PIRL, and the negative-free members Barlow Twins and VICReg — make two augmented views of one image and train the embeddings to agree. Chapter 9 covered the mechanics and the MI story. This section collects what is actually known.

15.3.1 View choice as the whole game

InfoMin says that *choosing views is choosing the inductive bias.* In practice SimCLR's augmentation pipeline (Chen et al., 2020a) does more work than its loss: crop-resize combined with colour jitter is what forces meaning into the embedding • [obs].

Tian et al. (2020) showed this directly with "optimal views". When the information the views share is engineered to match the task-relevant information (for example, splitting an image into colour and luminance channels), the representation beats standard augmentation on the target task. Views chosen instead to *maximise* shared information make it worse • [obs]. This supports the broader idea △ [hyp] that the augmentation, not the loss, is the real prior.

The flip side is a recognized vulnerability: whatever the augmentation distribution does *not* touch, the encoder is free to use as a shortcut. Resizing and color jittering does not change JPEG grid artifacts much; the resulting features embed JPEG noise alongside semantics. Cropping does not remove the background from a foreground-obscuring image; the resulting features systematically overfit to background context, as the whole robustness literature around contrastive features documents • [obs].

Key Concept — Invariance as implicit prior

Stating "trained with augmentation family $\mathcal{T}$" is equivalent to stating an implicit prior: *the equivalence classes definable by* $\sim_{\mathcal{T}}$ *(*$x' \sim_{\mathcal{T}} x$ *iff* x' *is reachable from* x*) correspond to the task's irrelevant factors.* Choosing $\mathcal{T}$ is Bayesian model design. It is no weaker than hand-crafting features — just cheaper —, and it fails in the same way: skin-color-based medical images and watermark-correlated web images both carry signals $\mathcal{T}$ happens not to destroy.

15.3.1.1 The argument, drawn

Here is the geometric version of that claim, which most readers find easier to hold than the set-theoretic one. Two words first, because the argument needs them.

New words used from here on — two words for the picture

A **manifold** is a curved surface sitting inside a much bigger space. Think of the skin of a balloon: every point on it needs only two numbers (how far round, how far up) even though the balloon lives in three dimensions. Natural images are believed to sit on something like this — a thin, curved sheet inside the enormous space of all possible pixel arrays. Almost every point in that enormous space is static noise; the images live on the sheet.

A representation is **flat** along a direction when moving along that direction does not change the output. Walk one step that way and the encoder returns the same vector. "Invariant to rotation" means "flat along the rotation direction."

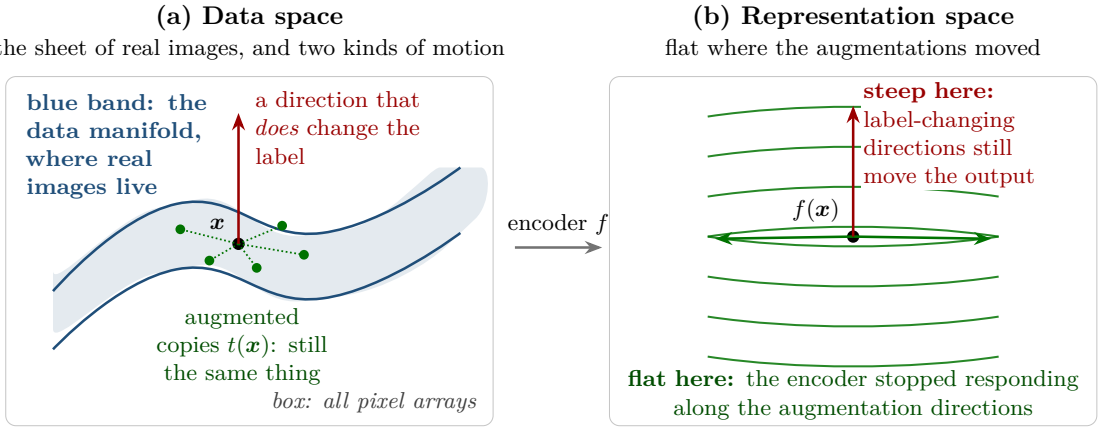


Figure 15.4. The augmentation-invariance argument in one picture. In data space (a), an augmentation moves a point along the manifold in directions that do not change what the image *is*; some other directions do change it. The joint-embedding loss asks the representation (b) to be flat — unchanging — along exactly the first set of directions, while remaining free to vary along the rest. (The curves in (b) are level sets of f: the closer together they are, the faster the output changes.) That is the entire mechanism, and it is also the entire risk: the encoder is flattened along whatever your augmentation list happens to move, whether or not those directions are the nuisance ones for your task.

How to read it: Compare the two green arrows in (b) with the green dotted spokes in (a). They are the same directions. Then ask the uncomfortable question: what if one of those spokes was carrying information your downstream task needed?

Figure 15.4 makes three things concrete that the prose above states abstractly. First, the augmentation family is a *choice of directions*, not a regulariser. Second, "invariance" is a local flatness property of the encoder, so it can be measured: perturb an input along an augmentation direction and see how far the embedding moves. Third — and this is the failure mode of Section 15.3.3 — directions your augmentations never move are directions the encoder was never asked about. It may keep them, it may discard them, and nothing in the loss decides which • **[obs]**.

15.3.2 Alignment and uniformity: the empirical picture

Chapter 9 named the two measurable quantities that explain why InfoNCE works in practice:

- **alignment** — positive pairs land close together;
- **uniformity** — the embeddings spread out over the whole unit sphere.

This split is an empirical observation • **[obs]** with a solid justification in the limit: with infinitely many negatives drawn uniformly, InfoNCE splits exactly into an alignment term and a uniformity term, and training satisfies both. At real batch sizes the split is only approximate, but it is still a useful diagnostic:

- good alignment, collapsed uniformity ⇒ collapsed, for all practical purposes;

- good uniformity, bad alignment ⇒ just a random projection.

Common Confusion

Uniformity is a diagnostic, not the mechanism. The uniformity of a trained embedding distribution is easy to measure (log of average pairwise Gaussian kernel, for one standard choice), and it correlates with downstream accuracy across methods • [obs]. The causal arrow runs from "negatives prevent collapse" to "the sphere fills up," not the other way: you can fill the sphere with garbage features by maximizing uniformity alone on random weights.

15.3.3 When it fails

The failure pattern of joint-embedding is distinctive and, in hindsight, obvious: the method is only as good as the equivalence relation the augmentation defines.

- **Shortcut to augmentation details.** An encoder that detects which crop of an image it sees — e.g. via interpolation statistics at the boundary, or via low-level color distribution matching — solves the matching task without learning category information. The SimCLR paper itself ablated this: color distribution alone carries much of the signal, which is exactly why color jitter is necessary • [obs]. Every augmentation added because the previous one left a shortcut is a footnote in the history of this failure mode.
- **Semantic false negatives.** Two different images of the same class repel each other. At large batch sizes this is a real loss of signal, not a schematic one; the clustering-aware methods (DeepCluster, Chapter 3; SwAV, Chapter 10) are precisely attempts to stop penalizing semantic neighbors. • [obs] — measured, though its quantitative effect on modern models is smaller than the early papers suggested.
- **Invariance when you want equivariance.** If the downstream task needs to know the rotation of an object or the exact color of a region, a representation trained to discard exactly that is structurally a bad starting point • [obs]; augmentation-matched fine-tuning softens but does not remove the conflict (the invariance-vs-covariance discussion of Chapter 8).

Key takeaway — joint embedding, and what would refute it

The claim. Joint-embedding methods work because the augmentation family $\mathcal{T}$ declares which directions of variation are nuisance, and the loss flattens the encoder along exactly those (Figure 15.4). The loss function itself is secondary; alignment and uniformity are diagnostics of the resulting geometry, not its cause. • [obs] for the dependence on $\mathcal{T}$; △ [hyp] for "$\mathcal{T}$ is the *dominant* variable."

What would have to be observed for this to be wrong. Three clean refutations, in order of how badly each would hurt:

- A joint-embedding pipeline reaching competitive transfer with *no* augmentation at all — two identical views — without smuggling the variation in somewhere else (different crops of a video, different sensors). Augmentation would then not be the prior.
- Downstream performance that is *insensitive* to swapping $\mathcal{T}$ for a family with clearly different invariances, on a task whose labels depend on one of the swapped factors. Concretely: a rotation-invariant model matching a rotation-sensitive one on an orientation-labelled benchmark.
- A measurement showing that encoders trained with the same $\mathcal{T}$ but different losses end up with *different* flat directions. That would mean the loss, not the augmentation, chooses the invariances.

The first two have been looked for and not found. The third is measurable today with a few hundred GPU-hours and, to this book's knowledge, has not been done systematically — which is why the word "dominant" above wears a △ **[hyp]** and not a ● **[obs]**.

15.4 Why Masked Reconstruction Works

Masked image modeling — MAE (He et al., 2022) as canonical — inverts the contrastive bet: instead of telling the encoder what *not* to encode, it forces the encoder to retain enough detail to fill in erased pixels. Chapter 5 covered the architecture and training. Here: why it works, in the bottleneck language of Chapter 4, and where the pixel-regression choice bites.

15.4.1 The information-bottleneck reading

Why predicting pixels teaches meaning. Write M for the mask, x_M for the visible patches, and train to make $\mathbb{E}\|\hat{x}(x_M) - x\|^2$ small. Perfect reconstruction is impossible — the exact texture of hidden fur cannot be known from the visible patches. So the encoder is pushed in two directions at once (△ **[hyp]** for the claim that it is exactly these two; ● **[obs]** for each one on its own):

- **Compression.** The few visible patches must carry enough information to rebuild everything else.
- **Meaning.** The best guess for a hidden patch depends on *what the object is*: an erased ear is filled differently for a dog than for a cat. Predicting pixels well therefore requires knowing the category and the shape — not just local colour.

This is an information-bottleneck geometry in the Tishby sense (Tishby et al., 2000): the mask family defines what is irrelevant (the high frequencies that cannot be predicted contextually anyway), and the bottleneck forces the encoder to keep precisely what was compressed out but recoverable from visible context. 75% masking is strong compression — strong enough that memorization strategies fail, unlike the text case (BERT, at 15%, must

solve an easier task per token because text carries more information per token; Chapter 5 covers the asymmetry). ● [obs]

Simplified intuition vs. more precise view

Simplified intuition. To fill in the erased pixels you must know what was there in essence — a category, a layout, a coarse texture —, so the encoder learns semantics by construction.

More precise view. Optimal pixel regression learns whatever the *visible-to-hidden conditional distribution* rewards: object class and layout yes, but also texture statistics, per-dataset palette, and per-scene lighting. What it systematically under-weights is the category-decision *boundary*: pixel MSE does not care much whether a boundary token is classified one way or the other when the resulting pixels are similar. This — not the bottleneck's compression — is the cleanest explanation of the linear-probe gap below △ [hyp] — strongly suggested by the mask-ratio and fine-tuning ablations, but not shown.

15.4.2 What pixel regression forces the encoder to keep — and where that hurts

Pixel regression keeps low-level detail alongside semantics. Against contrastive training the consequence is a recognizably different feature geometry ● [obs]. The practical symptom appears in the linear probe. A frozen linear probe on MAE features trails contrastive features of the same scale by a large margin, yet full fine-tuning from the same MAE checkpoint matches or exceeds contrastive fine-tuning (He et al., 2022). Interpretations:

- **Features live in the wrong geometry for linearity.** Reconstruction-shaped features concentrate semantic information in directions that are linearly accessible only after the backbone re-shapes them during fine-tuning; the projection head of contrastive training does that re-shaping as part of the pretext objective. △ [hyp]
- **Too much irrelevant information retained.** Pixel MSE keeps texture and palette detail that a linear classifier cannot suppress, so the probe's effective dimensionality is polluted. △ [hyp]
- **Both together.** When the target is a *learned, self-distilled* token distribution rather than raw pixels (iBOT), the linear-probe gap largely closes (Zhou et al., 2022). ● [obs] (A fixed dVAE codebook is not enough: BEiT's linear probe is *weaker* than MAE's (Bao et al., 2022).) Since the target network is nonlinear, this is evidence for explanation one; but it is also evidence for explanation two, because a learned codebook has fewer nuisance dimensions than RGB space.

Figure 15.5 draws the first two interpretations side by side, because they make the same prediction about the linear probe and different predictions about everything else — which is exactly what makes them separable by experiment.

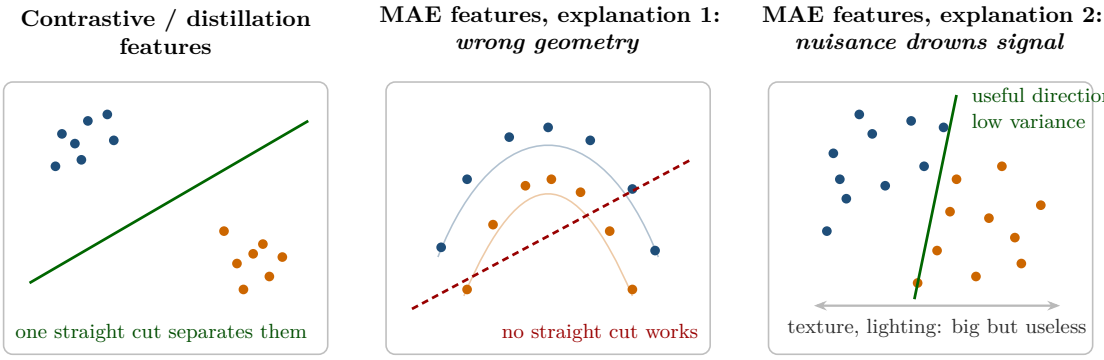


Figure 15.5. Two live explanations of MAE's linear-probe gap, drawn so that their *different* predictions are visible. Both reproduce the observed fact (a weak linear probe). They diverge on kNN: explanation 1 says the information is present but curved, so kNN should still be strong; explanation 2 says the information is swamped, so kNN should be weak too. The experiment that separates them is cheap and is spelled out in the takeaway below.

How to read it: Cover the captions and just look at the dots. Panel 2 has the classes fully separated — along a curve. Panel 3 has them overlapping in every direction but one faint one. Those are genuinely different worlds.

Operationally the lesson has been absorbed into combination methods (DINOv2's stack includes an iBOT-style masked-prediction term (Zhou et al., 2022; Oquab et al., 2024)): masked modeling gives strong fine-tuning initializations and good dense-prediction features; contrastive or distillation objectives give strong frozen and kNN behavior. The modern recipe hedges because no single theory says it should not have to.

Key takeaway — masked reconstruction, and what would refute it

The claim. Masking is an information bottleneck whose compression target is set by the mask ratio, and pixel regression makes the encoder keep low-level detail alongside semantics — which is why fine-tuning is strong and the frozen linear probe is not. ● **[obs]** for the gap itself; △ **[hyp]** for every proposed cause of it.

What would have to be observed for this to be wrong.

- *Against the bottleneck story:* a masking ratio low enough that local interpolation solves the task — say 15% on images — nevertheless yielding MAE-quality transfer. The bottleneck account says that must fail, and it is a one-run experiment.
- *Against explanation 1 (wrong geometry):* frozen MAE features whose kNN accuracy and non-linear-probe accuracy are *as weak as* the linear probe. If the information

is not recoverable by any cheap readout, it is not merely curved — it is absent, and explanation 2 wins.

- *Against explanation 2 (nuisance pollution):* swapping the pixel target for a learned-token target (iBOT-style) closing the linear-probe gap *without* changing what a dense-prediction head can read. Observed partially (Zhou et al., 2022) — though iBOT also adds an image-level distillation loss, so the test is not clean — which is why both explanations are still standing.

Notice that all three tests are affordable. The reason they have not settled the matter is that the community reports linear-probe numbers and rarely reports the matched kNN and depth-of-probe sweep on the same checkpoint.

15.5 Why Self-Distillation Without Negatives Works — Sometimes

BYOL (Grill et al., 2020), SimSiam (Chen and He, 2021), DINO (Caron et al., 2021), and DINOv2 (Oquab et al., 2024) train an online network to match a target network that is *itself derived from the online one* — via stop-gradient, an exponential moving average (EMA), an extra predictor head, or centering of the teacher's output distribution — without any explicit repelling force. This was operationally shocking in 2020–2021: the earlier chapters' collapse analyses say the trivial fixed point (predict the same constant, whatever the input) is right there, and these methods empirically avoid it. The honest summary today is that we have several *incompatible* explanations, none of which is unique or complete. Each one holds some levers and fails to hold others.

15.5.1 The three levers

Key Concept — The stop-gradient / EMA / predictor triangle

The three mechanisms that appear, in various combinations, in every working negative-free method:

- **Stop-gradient** $\mathrm{sg}(\cdot)$: the teacher branch is blocked from receiving gradients, so the loss cannot be reduced by moving the teacher toward the student.
- **EMA teacher**: teacher weights $\boldsymbol{\theta}_t \leftarrow m\,\boldsymbol{\theta}_t + (1-m)\,\boldsymbol{\theta}_s$ trail the student, making the teacher a slowly-changed ensemble of past students.
- **Predictor**: an extra small head applied to the student output only, breaking the symmetry between the two branches.

DINO adds a fourth lever, *centering* of the teacher's logits (running mean subtraction), which has a provable anti-collapse effect at the level of the softmax geometry: one dimension of the teacher's output cannot dominate.

Figure 15.6 shows where each lever sits in the two-branch picture and which method pulls which. Read the table underneath it as the ablation summary it is: every "−" in the grid is a published experiment where the method collapsed or degraded.

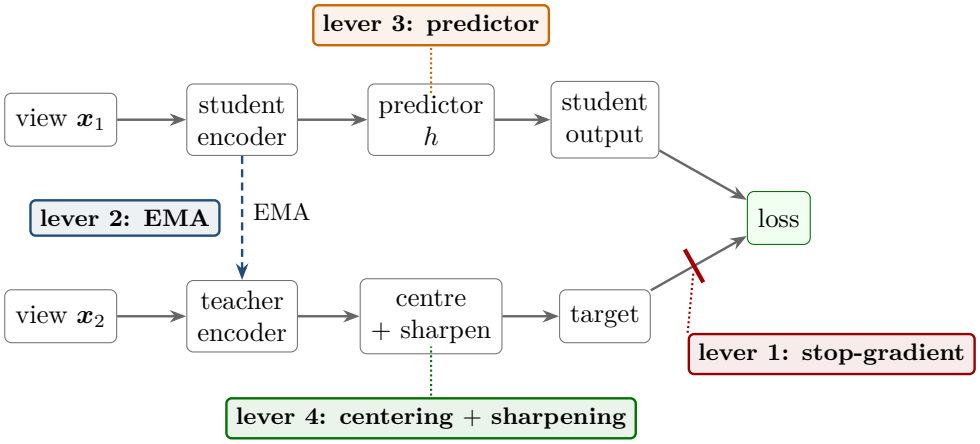


Figure 15.6. The four levers of negative-free self-distillation, and where each one sits. Nothing in this picture pushes any two samples apart — there is no repulsion term anywhere — and yet the constant solution is avoided. Which of the four levers does the work, and whether any is strictly necessary, is the subject of the four competing explanations below.

How to read it: Trace the two paths from the two views to the loss, then note the red bar: gradients never travel back along the lower path. That single missing arrow is the most load-bearing absence in modern SSL.

Table 15.1. Which method pulls which lever. "−" marks a lever the method deliberately omits; the right-hand column records what the published ablation found when a lever that *is* present is removed.

Method	stop-grad	EMA	predictor	centering	Reported effect of removal
SimSiam (Chen and He, 2021)	yes	–	yes	–	Remove either present lever: immediate collapse, kNN at chance • [obs]
BYOL (Grill et al., 2020)	yes	yes	yes	–	Remove the predictor: collapse. Remove the EMA (teacher = student): collapse too, in BYOL's own ablation • [obs]
DINO (Caron et al., 2021)	yes	yes	–	yes	Remove centering: one output dimension dominates. Remove sharpening: outputs go uniform • [obs]

15.5.2 The competing explanations

Explanation 1: asymmetry as an implicit constraint. Removing the predictor from BYOL or the stop-gradient from SimSiam induces immediate collapse (Grill et al., 2020; Chen

and He, 2021) — ● **[obs]**, and the cleanest experiment in the whole area. The story: with symmetric branches and gradients flowing, the trivial constant solution is the globally easy minimum; each asymmetry (stop-grad, predictor, momentum teacher) tilts the optimization landscape away from it, either by removing the gradient path into the constant, or by making the student chase a moving target that its own present commit cannot satisfy without keeping information about the input. △ **[hyp]** — the concise correct statement is that no theorem says the landscape must be so tilted; the evidence is ablation after ablation consistent with it.

Key takeaway — explanation 1, and what would refute it

Falsifier. A negative-free method with *fully symmetric* branches — no stop-gradient, no predictor, no EMA, no centering — that trains to competitive accuracy without collapsing. One such result would show the asymmetry is sufficient but not necessary, and the explanation would shrink from "this is why it works" to "this is one way to make it work." A weaker but still damaging observation: a method where removing an asymmetry hurts accuracy *without* the embedding variance falling, which would mean the lever was buying something other than collapse avoidance.

Explanation 2: implicit contrast through normalization or batch statistics. In some configurations BatchNorm layers (present in BYOL's encoder) let the representations of other samples in the batch influence the loss through the normalization statistics, smuggling in a weak repulsion. △ **[hyp]** — suggestive ablations exist, but BYOL survives with group normalization and weight standardization (no batch statistics at all) in follow-up work, so this cannot be the whole story. DINO's centering plays a similar role explicitly. SimSiam deliberately removes batch normalization from the prediction head, weakening this channel, and still works — ● **[obs]**.

Key takeaway — explanation 2, and what would refute it

Falsifier. Largely already observed, which is why this explanation has the weakest standing of the four. The clean version: train BYOL with every batch-coupling path removed — LayerNorm everywhere, no BatchNorm, batch size one with gradient accumulation so no sample can see another — and show it still avoids collapse. Each step in that direction has been taken and the method has survived; a complete version would close the question. Conversely, the explanation would be *supported* by showing that embedding variance tracks batch size in exactly the way an implicit repulsion term predicts.

Explanation 3: the EMA as a temporal ensemble. A teacher that is a trailing average of students is approximately an exponentially-weighted ensemble of the student's history. Predicting the consensus of your recent selves is a self-consistent update rule that provably (at least in linearized models) has the constant solution among its fixed points but empirically falls away from it early in training, when the student's features are still changing direction fast. △ **[hyp]** — linear-model analyses are published; the non-linear case is ◊ **[open]**. The DINOv2/v3

training recipe's use of KoLeo regularization (Sablayrolles et al., 2019) — pushing batch features away from each other's nearest neighbors — is plausibly partly belt-and-braces for this channel △ [hyp].

Key takeaway — explanation 3, and what would refute it

Falsifier. A method with *no* temporal averaging at all — SimSiam is nearly this already, since its target is the current network's own output — that matches EMA-based methods on the metrics the EMA is supposed to protect (embedding rank, teacher entropy, kNN stability). That SimSiam works at all is the strongest existing evidence against making the EMA necessary. What would *support* the explanation: showing that the collapse-onset time scales the way the EMA coefficient m predicts, as m is swept across two orders of magnitude at fixed everything else.

Explanation 4: it just has not collapsed yet. The most subversive story, not disproven: negative-free methods are in a slow collapse that good recipes make slower than training △ [hyp]. Under this view the stop-grad/EMA/predictor triangle delays rather than prevents collapse, which would explain the observed sensitivities (to warmup of the EMA coefficient, to centering updates, to learning-rate schedules) and the documented training-window behavior of DINO-style models. No published experiment cleanly distinguishes "delayed" from "prevented" at the training horizons people use.

Key takeaway — explanation 4, and what would refute it

Falsifier. Run one negative-free method for ten to a hundred times the usual number of steps, at fixed learning rate and fixed EMA coefficient, and monitor the eigenvalue spectrum of the representation covariance (Figure 15.7). If the spectrum is stationary — no slow drift of small eigenvalues toward zero — the "slow collapse" story is dead and the levers genuinely prevent rather than postpone. If the small eigenvalues creep down monotonically, it is alive and everybody's recipe is a stopwatch.

This is the most useful falsification test in the chapter because it is purely a matter of compute and patience: no new method, no new theory, no new dataset. That nobody has published it is a statement about incentives, not about difficulty. ♢ [open]

Research Frontier

Open: a collapse-proof characterization. What is missing is a theorem of the form: with such-and-such architecture, loss, and optimizer, all attractors except collapse are ruled out, or conversely collapse has zero basin of attraction. Existing theory covers linear networks (special cases), one-layer cases, and the dynamics around already-good initializations. Whether the triangle levers work by *basin geometry* (no path to collapse from typical initialization) or by *speed* (the optimizer never reaches the collapse basin in available steps) is genuinely unresolved. A resolved theory would tell us which levers

are necessary and which are redundant — a question of direct economic interest, because each lever costs hyperparameters.

15.6 A Collapse Taxonomy Across All Book Families

"Collapse" has been used loosely so far; it deserves a taxonomy, because different families fail *into different kinds of triviality*.

15.6.1 Complete collapse and the partial kind you will actually meet

The collapse everyone pictures is *complete*: the encoder outputs the same vector for every input, kNN accuracy drops to chance, and the loss flatlines within a few hundred steps. It is dramatic, obvious, and rarely what damages a real run. The version that damages real runs is **dimensional collapse**: the embeddings stay different from each other, the loss keeps improving, and yet they all lie inside a thin slab of the space. Most of the representation's dimensions carry nothing.

New words used from here on — reading a covariance spectrum

> Take the embeddings of a large batch, subtract the mean, and compute the covariance matrix. Its **eigenvalues** measure how much the cloud of embeddings spreads along each of its own natural axes, sorted from widest to narrowest. A big eigenvalue means "embeddings differ a lot along this direction"; an eigenvalue near zero means "all embeddings have the same value along this direction — the dimension is dead." Plotting them on a log scale, largest first, is the single cheapest health check for any joint-embedding run.

Dimensional collapse matters here for a specific reason: it is the measurement that distinguishes Explanation 4 of Section 15.5.2 — "it just has not collapsed yet" — from its rivals. A method that genuinely prevents collapse has a stationary spectrum; a method that merely postpones it has a spectrum whose tail creeps downward across training. Both look identical on a loss curve • **[obs]**. This is also why VICReg's variance term and DINOv2's KoLeo regulariser exist: they are, directly, penalties on the tail of this spectrum going flat (Bardes et al., 2022; Sablayrolles et al., 2019).

Two patterns cut across the table. First, every family spends a conspicuous fraction of its recipe preventing a specifically identifiable degenerate solution, and the "why it works" stories are mostly stories about those preventions being sufficient in practice, not in theory. Second, the blocking mechanism is always *adding a constraint the trivial solution cannot satisfy*: balance constraints, negatives, bottlenecks, stopped gradients, high masking ratios. There is no family in which gradient descent on the unconstrained objective is known to avoid collapse on principle.

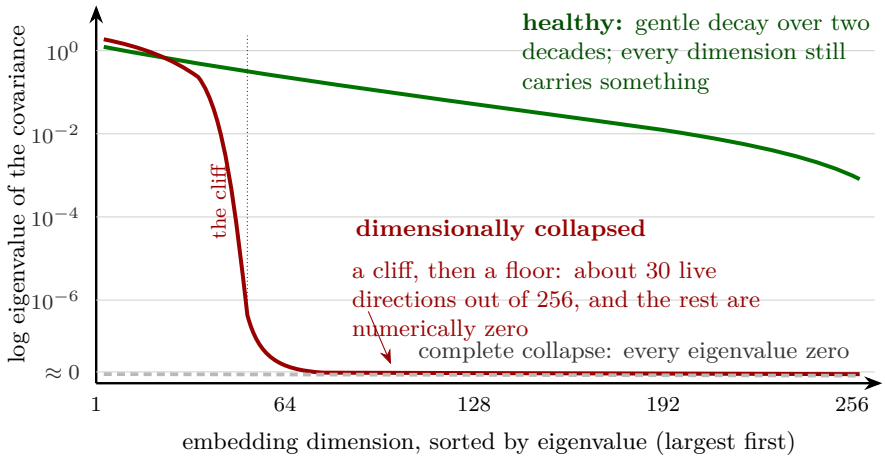


Figure 15.7. The health check. Each curve is the eigenvalue spectrum of the covariance of a batch of embeddings, sorted largest-first on a logarithmic axis. A healthy representation decays gently: information is spread across the dimensions. A dimensionally collapsed one falls off a cliff and then runs along the floor — the embeddings differ from each other, so the loss and the training curves look fine, but they differ only inside a thin subspace. Complete collapse (dashed) is the degenerate case where even that subspace vanishes. Curves are schematic; the *shape* is what is reproducible across papers • **[obs]**.

How to read it: Look for the cliff, not the height. A spectrum that drops several decades within the first tenth of the dimensions is the warning sign, and it appears long before any downstream metric moves.

Key takeaway — the collapse taxonomy, and what would refute it

The claim. Every family in this book spends a visible part of its recipe blocking one specific degenerate solution, and every blocking mechanism works the same way: it adds a constraint the degenerate solution cannot satisfy. • **[obs]** across all nine rows of Table 15.2.

What would have to be observed for this to be wrong. A self-supervised objective, trained by plain gradient descent with no architectural asymmetry, no negatives, no balance constraint, no bottleneck and no masking, that reliably avoids its own trivial solution. Not once, on one dataset — reliably, across seeds and scales. Such a method would show that the constraint framing is a historical accident of how the field searched, rather than a property of the problem. Nothing close exists today, and the reason is worth stating plainly: we have no theory that predicts which basin gradient descent will enter, so the field has resorted to removing the bad basin by construction. ◊ **[open]**

Historical Note

The first clear articulation that "rich input structure alone generates learning signal" is older than deep learning: clustering as compression, and minimum-description-length principles in the 1970s–80s, all assume that a model finding structure in unlabeled data is doing something epistemically respectable. Becker and Hinton's Imax (Becker and Hinton, 1992) phrased the modern form in 1992 — two halves of the visual field predict each other

Table 15.2. Collapse modes by self-supervised family. "Signal destruction" means the model recovers solution information from a channel other than the input's semantics.

Family	Trivial solution	Mechanism blocking it	What breaks when the mechanism fails	Confidence of diagnosis
Clustering-based (DeepCluster, SwAV)	All samples assigned one cluster	Balanced assignment constraints (Sinkhorn); periodic re-init	Degenerate pseudo-labels; feature norms drift	Empirical • [obs]
Autoencoders (Ch. 4)	Identity / memorization	Bottleneck size; denoising; masking	Perfect reconstruction, random features	Empirical • [obs]
VAE	Posterior collapse: $q(\boldsymbol{z} \mid \boldsymbol{x}) \approx p(\boldsymbol{z})$	KL weighting, decoder capacity limits	Latent unused; no representation learned	Empirical, mechanism understood • [obs]
Masked image modeling (MAE)	Predict dataset mean	Mask ratio $\gg$ effective redundancy	Blurry mean-image decoder; encoder ignores semantics	Empirical • [obs]
Joint-embedding w/ negatives (SimCLR, MoCo)	Constant embedding	Negative pairs repel	Rank collapse of embedding matrix	• [obs] + linear-case ■ [thm]
Negative-free (SimSiam, BYOL, DINO)	Constant student output	Stop-grad, EMA, predictor, centering	Instant loss collapse; kNN at chance	Ablations clear • [obs]; mechanism △ [hyp]
Masked language modeling (BERT)	Predict unigram distribution	Bidirectional context; masking fraction	Degenerate token statistics; no syntax	Established empirically • [obs]
Self-training (Ch. 12)	Teacher confirms own errors	Confidence thresholding; augmentation asymmetry; student capacity $\geq$ teacher	Confirmation bias runaway	Empirical • [obs]
Cross-modal distillation (Ch. 14)	Student ignores own modality	Losses on paired-time samples; focal reweighting	Student reduces to teacher bias	Empirical • [obs]

— and Bromley et al.'s Siamese network (Bromley et al., 1993) supplied the architecture. What changed in 2018–2020 was not the principle but the demonstration, at scale, that the signal is strong enough to compete with ImageNet labels (CPC (van den Oord et al., 2018b), MoCo (He et al., 2020), SimCLR (Chen et al., 2020a)). The field's theory is still catching up with that demonstration.

15.7 Scaling Beats Architecture at the Far End

DINOv2 and DINOv3 (Oquab et al., 2024; Siméoni et al., 2025) encode the single most important empirical lesson of SSL so far, and it is worth stating bluntly: past a certain capability level, the dominant variable is no longer the architecture, the loss, or the theory; it is the *data engine*.

Paper Analysis: DINOv2: learning robust visual features without supervision (Oquab et al., 2024)

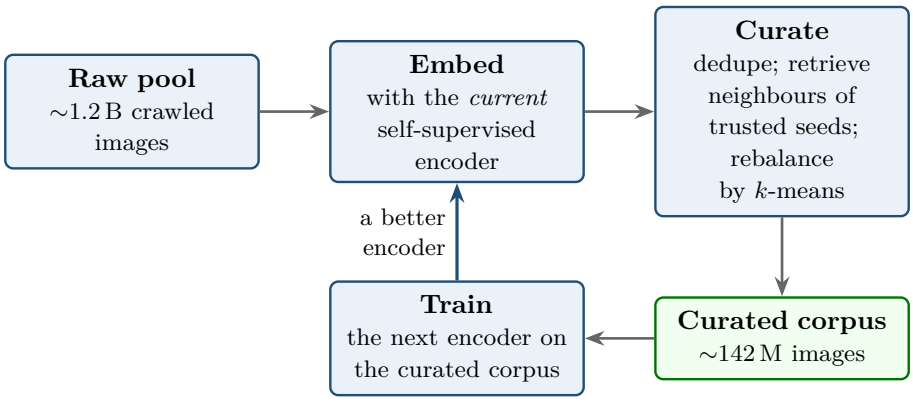
Problem. Prior SSL models matched ImageNet classification but degraded sharply on dense and out-of-distribution tasks; curated ImageNet was not a good image-trainer for a general-purpose encoder. **Method.** The DINO/iBOT recipe, but trained on a purpose-built corpus: approximately 142 million images, filtered from an initial pool of about 1.2 billion by deduplication and *retrieval-based curation* — for each image in trusted curated sets (including small labeled sets), retrieve its nearest neighbors in an existing embedding space and keep them. **Results.** ViT-L and ViT-g models reached ImageNet kNN accuracy of about 83% and linear-probe accuracy of about 86%, and matched or outperformed weakly-supervised baselines such as CLIP-style pipelines on a wide battery of tasks including depth estimation, retrieval, and segmentation. • **[obs]** (numbers as reported). **What to remember.** The architecture and objective are nearly unchanged from DINO; the delta is the data pipeline. Data curation is expressible as SSL machinery itself: it uses a self-supervised embedding to select the data for a better self-supervised embedding.

Paper Analysis: DINOv3 (Siméoni et al., 2025)

Problem. Scale DINOv2's recipe to web-scale and to larger models. **Method.** Approximately 1,689 million images (about 17 TB), assembled by extending the DINOv2 pipeline: hierarchical k-means balancing of the curated pool, scaled retrieval curation, and training up to a roughly 7-billion- parameter ViT. Adds a post-training high-resolution adaptation step and a regularizer (Gram anchoring) targeted at keeping dense features clean at long training horizons. **Results.** State-of-the-art across a frozen-features evaluation suite — global tasks, dense tasks including segmentation and 3D correspondence, and video — against both self-supervised and weakly-supervised baselines. • **[obs]** (as reported). **What to remember.** The marginal returns per logarithm of additional *model* size were realized only because data quality was actively maintained during scaling; scaling alone on the previous data distribution had visibly flattened.

15.7.1 The data-engine argument

Chapter 11 described the pipeline mechanics. The research point belongs here: the pipeline itself argues that *the differences between SSL objectives are, at the far end, second-order*



The open question *(Section 15.7): the encoder chooses the data that trains the next encoder. Does the loop converge to broad coverage, or does it amplify the first encoder's blind spots? Nobody has characterised its fixed points.* ◊ ***[open]***

Figure 15.8. The data engine as a self-referential loop. The step that makes DINOv2 and DINOv3 work is not in the loss: it is the arrow that runs backwards, where a self-supervised embedding is used to *select* the data for a better self-supervised embedding (Oquab et al., 2024; Siméoni et al., 2025; Vo et al., 2024). Numbers are those reported for the DINOv2 pipeline.

How to read it: Follow the arrows once round, then notice that the loop has no external input after the first turn. That is both why it scales and why its failure mode is hard to see from inside.

compared to the differences between data distributions. All of the method-level knobs this chapter has worried about — view choice, projection heads, centering, predictor asymmetry — are choices made inside a fixed data distribution. As that distribution is actively shaped toward semantic coverage, the requirements on those knobs relax; conversely no knob setting discovered on ImageNet transfers confidently to distribution-shifted corpora. The theoretical shift this implies, relative to chapters 4–11, is significant: "Does the loss avoid collapse?" becomes a minor question when the dominant empirical variable is "Is the corpus balanced against overrepresented near-duplicates and semantically thin regions?" (Vo et al., 2024).

This is also where the chapter's honesty rule bites hardest. The claim "scaling beats architecture" is an • **[obs]**at present scaling frontiers; there is no theorem saying it must remain true, and there are specific counter-hypotheses (e.g. dense tasks with geometric structure may care about architecture more at all scales). The safest reading is *conditional*: *at present accessible scales, for the tasks the field measures, data quality and distribution are the binding constraint.*

Key takeaway — scaling, and what would refute it

The claim. At present scales and on the tasks the field measures, the data distribution is the binding constraint, not the objective or the architecture. • [obs] — and conditional: it is a statement about a particular scaling frontier, not a law.

What would have to be observed for this to be wrong. A task family where two models trained on *identical* curated corpora, at matched parameter count and compute, separate by a wide margin purely on architecture or objective — and separate *more* as scale increases rather than less. Dense geometric tasks on shifted domains (medical, satellite, underwater) are the natural place to look, because web-curated corpora contribute almost no coverage there and architectural inductive biases express what no feasible corpus can. If the ranking of published models reverses on such a slice with the data engine held fixed, the claim is falsified on that slice — and, since the claim was only ever conditional, that is a refinement rather than a refutation. What would refute it outright is the reverse pattern at the frontier: curation ceasing to buy anything while objective changes keep paying.

Research Frontier

Open theoretical questions raised by this chapter.

1. **Necessary conditions on views.** InfoMin gives a sufficient design principle for view choice but not a verifiable condition on a *given* augmentation pipeline. Can we compute, or bound from data, whether a pipeline destroys task-relevant information before spending the training run? (Related to identifiability results for nonlinear ICA viewed through augmentations; current versions require assumptions that real crop-jitter families do not satisfy.)
2. **Collapse basin geometry.** For the negative-free triangle (Section 15.5.1), is collapse an attractor that training never reaches (speed story), or a saddle that training escapes by construction (geometry story)? These have radically different implications for recipe transfer.
3. **Why masking ratio is so flat.** MAE tolerates masking ratios from about 60% to about 90% with modest variation. No information- theoretic account predicts a plateau that wide, nor explains why text (BERT, 15%) tolerates so much less without appeal to "information density per token," which has resisted formalization.
4. **A theory of the data engine.** Retrieval-based curation bootstraps a representation to select data for a better representation. What are the fixed points and failure modes of this loop — does curation amplify the embedding's blind spots at scale?
5. **Linear-probe vs. fine-tune divergence.** A representation theory that predicts *which* pretraining objective yields linearly separable features, as opposed to merely fine-tunable features, would settle several of the disagreements tabulated above at once.

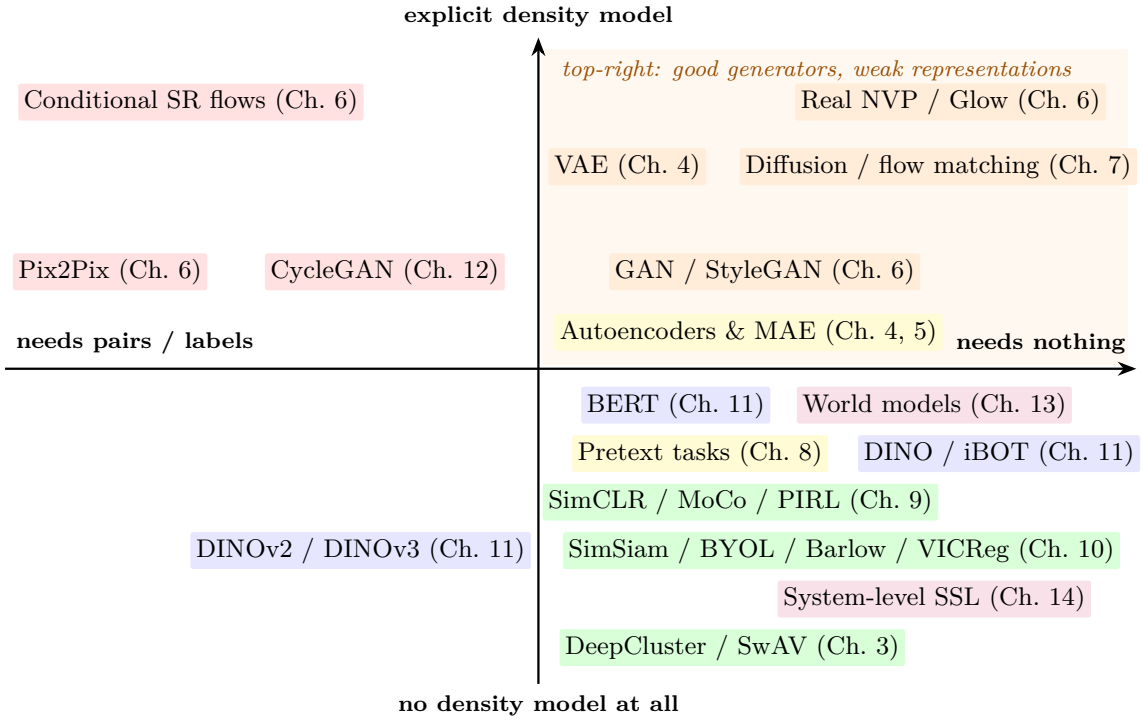


Figure 15.9. Landscape of the methods in this book on the two axes this chapter argues are load-bearing. Horizontal: whether the method needs label-like supervision or curation *at all*, from paired-data methods (left) to pure unlabeled-stream methods (right). Vertical: how explicitly the method models the data density, from explicit likelihood models (top) to methods with no density model at all (bottom). Positions are schematic. The point is the shaded top-right quadrant: the methods there (flows, VAEs, diffusion, GANs) are strong generators, but no *representation* success sits there, because turning an explicit density model without label-like structure into good features remains unsolved (Section 15.7).

15.8 Landscape of the Book's Methods

Figure 15.9 places every method family of this book on the two axes the earlier sections argued matter most.

- **Horizontal: how much label-like structure does it need?** Not in the strict semi-supervised sense, but practically: "would it still work on anonymous, uncurated pixel streams with no trusted seed set?"
  - Pix2Pix needs paired data.
  - Curated pipelines (DINOv2/v3) need a small trusted seed set to retrieve against.
  - Contrastive methods work on uncurated, unlabelled streams.
- **Vertical: how explicit is its density model?** From explicit likelihoods (flows), through implicit density models (diffusion, GANs, VAE), to none at all (joint-embedding and self-distillation, which never model $p(x)$).

The top-right corner holds no representation success, and that is no accident. It is where methods with an explicit density model *and* no need for labels or pairs sit — flows, autoregressive pixel models, and (with a bound or an implicit density) VAEs, diffusion and GANs. Those are exactly the families this book showed stalling as *representation* learners, even while they thrived as *generators*. Chapters 6 and 7 explain why: maximising likelihood spends capacity on invisible detail. This chapter adds the other half: every representation success to the right of the diagram came from *dropping* the density model in favour of pretext tasks, contrasts, or masked/autoencoding bottlenecks.

Key takeaway — what the top-right corner means

Read Figure 15.9 as a record of what the field tried and where it stopped. The top-right corner — an explicit density model that needs no pairing, no curation and no labels — holds good generators but no representation success, not because nobody wanted one but because likelihood objectives spend their capacity on detail that no downstream task uses. Every representation success on the right of the diagram came from giving up the density-model framing.

What would refute the reading. A likelihood-trained model whose frozen features match DINOv2-class features on kNN and dense transfer at matched compute. Diffusion features (DIFT, Chapter 7) are the closest existing challenge and they are a byproduct of a generative model, not a competitive frozen encoder — which is exactly the distinction the top-right corner records. △ [hyp]

15.9 Summary

Prompt: per family, why does it work, and when does it fail. The compressed answer, all confidence levels shown:

- **Central dogma.** Works when X has learnable structure and the pretext task's solution requires downstream-useful features; fails by memorization, shortcuts, and optimizer basin selection. MI bounds give the vocabulary (van den Oord et al., 2018b; Poole et al., 2019); augmentation choice is the operative prior (Tian et al., 2020). ● [obs], on a ■ [thm] framework
- **Joint-embedding.** Works because alignment+uniformity is a well-behaved geometry at scale, and because $\mathcal{T}$ defines the nuisance factors. Fails along the untouchable factors of $\mathcal{T}$: shortcuts and missing covariance. ● [obs]
- **Masked reconstruction.** Works as an information bottleneck whose compression target is the masking pattern. Fails vs contrastive on frozen linear evaluation because pixel regression does not prioritize category boundaries. ● [obs] for the gap, △ [hyp] for the explanation

- **Self-distillation.** The stop-grad / EMA / predictor triangle blocks collapse by mechanisms whose theoretical status is unsettled; four competing explanations coexist, none proven. ● [obs] for the ablations, △ [hyp] for the mechanisms, ◊ [open] for the synthesis
- **At scale.** Data engines dominate architecture and objective (Oquab et al., 2024; Siméoni et al., 2025); "why SSL works" is silently changing referent from training dynamics to corpus design. ● [obs]

15.9.1 The chapter on one page

Table 15.3 is the chapter compressed into the two columns that matter for revision: how much evidence each explanation carries, and the single observation that would sink it. If you remember one artefact from this chapter, remember the right-hand column — an explanation with no entry there is not an explanation, it is a slogan.

Key takeaway — the chapter in three sentences

Self-supervised learning is an engineering success with a thin and contested explanation, and pretending otherwise is the main way students get caught out in research discussions. The four markers — ● [obs] ■ [thm] △ [hyp] ◊ [open] — exist so that you can hold a strong opinion about a method's behaviour while holding a weak one about its cause. When someone tells you why SSL works, the useful reply is not "I disagree" but "what would you expect to see if that were false?"; if they cannot answer, Table 15.3 has a row that can.

Check Your Understanding

(a) [**Level 1**] State the two conditions of the central dogma in your own words, and identify which condition the autoencoder memorization failure violates.

(b) [**Level 1**] Which of these is theorem-grade: "InfoNCE maximizes mutual information," or "InfoNCE $+ \log N$ lower-bounds mutual information under stated assumptions"? Why does the difference matter for interpreting training curves?

(c) [**Level 2**] The InfoMin principle recommends minimizing $I(V_1; V_2)$ subject to preserved task information. Give one augmentation change that moves a SimCLR pipeline in the InfoMin direction, and one that moves away from it.

(d) [**Level 2**] Explain the alignment/uniformity split, and give a training configuration that would produce good alignment with bad uniformity and one producing the reverse.

(e) [**Level 3**] A colleague reports that their contrastive model solves matching "perfectly" (near-zero train loss) but transfers poorly to segmentation. Using the shortcut-to-augmentation framing, propose two hypotheses and one diagnostic each.

(f) [**Level 3**] Reproduce the information-bottleneck reading of MAE: identify the "compression" term and the "relevance" term. What, in this reading, is predicted to

Table 15.3. The chapter's explanations, their evidence level, and the observation that would falsify each. Every row's falsifier is an experiment somebody could run; none requires new theory.

Explanation	Level	What would have to be observed for it to be wrong
Central dogma: structure + task alignment (§15.2)	● [obs]	Transferable features learned from genuinely structureless data; or transfer that is uncorrelated with whether the pretext solution needs semantics
InfoNCE bounds mutual information (§15.2.1)	■ [thm]	Nothing — the theorem is true. What is falsifiable is the *use* of it: show transfer tracking estimated MI once the batch-size ceiling $\log N$ is crossed
Augmentation is the real prior (§15.3.1)	● [obs]+ △ [hyp]	Competitive transfer with no augmentation; or insensitivity to swapping the augmentation family on a task whose labels depend on the swapped factor
Alignment/uniformity explains contrastive success (§15.3.2)	● [obs]	A model with good alignment and good uniformity that transfers badly, or the converse — the diagnostic would stop being diagnostic
Masking is an information bottleneck (§15.4.1)	△ [hyp]	MAE-quality transfer at a mask ratio low enough for local interpolation to solve the task
MAE's probe gap is wrong geometry (§15.4.2)	△ [hyp]	Frozen MAE features whose kNN and deep-probe accuracy are as weak as the linear probe
Asymmetry blocks collapse (§15.5.2)	△ [hyp]	A fully symmetric negative-free method that trains to competitive accuracy
BatchNorm smuggles in repulsion (§15.5.2)	△ [hyp]	Largely already refuted: BYOL survives without BatchNorm (GroupNorm + weight standardization); SimSiam drops BatchNorm from the predictor and still works
The EMA is a stabilising ensemble (§15.5.2)	△ [hyp]	SimSiam, which has no EMA, matching EMA methods on rank, teacher entropy and stability
Collapse is delayed, not prevented (§15.5.2)	◊ [open]	A stationary covariance spectrum over 10×–100× the usual training length (Figure 15.7)
Data quality beats architecture (§15.7)	● [obs]	A task family where architecture separates models trained on identical corpora, and separates *more* with scale

change if you replace the pixel-MSE decoder loss with a dVAE-token classification loss (BEiT)?

(g) **[Level 3]** List the three levers of the negative-free triangle and give, for each, an ablation result from SimSiam, BYOL, or DINO that shows what happens when it is removed.

(h) **[Level 4]** Critique "MAE's linear-probe gap proves masked modeling doesn't learn semantics." What experiment, using only published methods from this book, would sharpen the claim, and what would each outcome imply?

(i) **[Level 4]** The collapse taxonomy contains two rows whose blocking mechanism is, at root, the same constraint. Identify them and argue whether they should be merged.

(j) **[Level 5]** The data-engine argument says scaling data quality beats architecture. Construct a concrete falsification test: a distribution of tasks on which the claim would visibly reverse within published model sizes. Defend your choice.

Solution sketches.

(a) Structure in data *and* task-solution requiring downstream features. Memorization satisfies the first and violates the second — reconstruction is solved perfectly with arbitrary features.

(b) Only the second is theorem-grade (Poole et al., 2019). The first makes training curves into MI estimates, which they provably are not beyond $\log N$ bits; relying on it misreads saturation as "maximum information reached."

(c) Toward InfoMin: stronger color jitter plus crop diversity (destroys nuisance confounds without touching category structure). Away: adding a modality-invariant augmentation that erases the discriminative factor (e.g. extreme blur that makes small-object classes indistinguishable).

(d) Good alignment / bad uniformity: train with positives only (no negatives) and no collapse protection — quickly converges to a constant. Good uniformity / bad alignment: initialize randomly and maximize uniformity directly — sphere fills with noise.

(e) Hypotheses: (i) boundary-interpolation statistics fingerprint the crop; (ii) color histogram survives jitter and indexes identity. Diagnostics: (i) evaluate with crop-only augmentations and no jitter — if matching accuracy stays high, structure leaks; (ii) match images across global-color-degraded copies — a drop indicates the histogram channel was load-bearing.

(f) Compression: masking rate (forces visible-token bottleneck); relevance: pixel MSE rewards whatever predicts pixels. With dVAE targets, the relevance term shifts from pixel-to-token statistics; predicted change per the chapter's hypothesis: narrower linear-probe gap, because token boundary errors cost direct classification loss. What was observed: BEiT's fixed dVAE tokens do *not* narrow it (its linear probe is weaker than MAE's, Bao et al., 2022), while iBOT's self-distilled tokens do (Zhou et al., 2022) — a discrete target alone is not enough.

(g) Stop-grad: removing it collapses SimSiam instantly (Chen and He, 2021). Predictor: removing it collapses BYOL in the original ablations (Grill et al., 2020). EMA: in DINO's ablation, using the student itself as the teacher fails completely (Caron et al., 2021); without centering or sharpening one output dimension dominates.

(h) The gap alone proves only that features are not *linearly* accessible, nothing about semantic content. Sharpening experiment: kNN classification, probe depth sweep, and dense-task transfer on the same frozen MAE vs. SimCLR backbones (all published). If kNN and dense transfer are both strong when linear probe is weak, the "wrong-geometry" explanation is supported; if all three degrade together, the "no semantics" reading gains support.

(i) Negative-free self-distillation (stop-grad/EMA triangle) and autoencoder-bottleneck rows both block the constant shortcut by forcing the output channel through an information-limited pathway. Arguable either way: merging is defensible if you classify by *blocked trivial solution*, splitting is defensible if you classify by *mechanism family* (capacity constraint vs. gradient-path constraint).

(j) A distribution of dense geometric tasks (correspondence, depth, pose) on distribution-shift domains (medical, satellite, underwater) where (i) web-curated corpora contribute nearly no coverage, and (ii) architectural inductive biases (equivariance, multi-scale) express what no corpus at feasible scale does. If architecture choice reverses the ranking of published models there while data-engine size is held fixed, "scaling beats architecture" is falsified on that slice. The defense rests on the conditional reading of the claim: it has only ever been an empirical claim at given scale and task mix, not a theorem.

Concept Map — how this chapter connects

Backward dependencies.

- *Ch. 1 (central dogma)*: formalized in Section 15.2.1; "anything better than chance" becomes a two-condition criterion.
- *Ch. 2 (mutual information, KL)*: the estimation toolkit behind Section 15.2.1; (Poole et al., 2019) was previewed there.
- *Ch. 3 (DeepCluster) and Ch. 10 (SwAV)*: clustering-row of the collapse taxonomy; balanced assignment as the anti-collapse constraint.
- *Ch. 4 (VAE, bottlenecks)*: posterior collapse; information-bottleneck language used for MAE; (Tishby et al., 2000).
- *Ch. 5 (MAE)*: Section 15.4 re-reads it as an IB; the linear-probe gap originates there (He et al., 2022).
- *Ch. 6 (flows, GANs)*, *Ch. 7 (diffusion)*: top half of the landscape figure; why explicit density modeling stalled as a representation engine.
- *Ch. 8 (augmentations, invariance vs covariance)*: the equivalence-relation view of $\mathcal{T}$ in Section 15.3.1.

- *Ch. 9 (SimCLR, MoCo, InfoNCE)*: MI-bound caveats, InfoMin, alignment/uniformity (Chen et al., 2020a; He et al., 2020; van den Oord et al., 2018b; Tian et al., 2020).
- *Ch. 10 (SimSiam, BYOL, Barlow, VICReg)*: competing collapse explanations (Chen and He, 2021; Grill et al., 2020; Zbontar et al., 2021; Bardes et al., 2022).
- *Ch. 11 (DINO family, iBOT, KoLeo, data engines)*: EMA teacher, centering, KoLeo, and the DINOv2/v3 scaling argument (Caron et al., 2021; Zhou et al., 2022; Sablayrolles et al., 2019; Oquab et al., 2024; Siméoni et al., 2025; Vo et al., 2024).
- *Ch. 12 (self-training, Noisy Student)*: confirmation-bias row of the taxonomy (Lee, 2013; Xie et al., 2020).
- *Ch. 13 (world models, DINO-WM)*: landscape placement and the frozen-feature planning evidence (Zhou et al., 2024).
- *Ch. 14 (system-level SSL)*: cross-modal distillation's specific failure mode; the landscape's system quadrant (Frey et al., 2022; Rivera Valverde et al., 2021).

Forward. Chapter 16 turns the diagnoses here into an operational decision procedure: which family fits a given data regime, which evaluation artifacts to distrust, and a researcher's checklist that operationalizes the confidence ladder.

CHAPTER 16

Comparing the Families: Reconstruct, Contrast, or Distill?

Everything should be made as simple as possible, but not simpler.

attributed to *Albert Einstein*

In this chapter

16.1 Reading the comparison tables 740
16.2 Master comparison tables ..... 742 • What the tables show, compressed
16.3 The confusion clearinghouse ..... 749 • Smaller confusions worth one line each
16.4 Decision guidance: which family first ..... 753 • Eight heuristics
16.5 Evaluation cross-reference 757
16.6 Summary 758

Chapter roadmap

Before you start. This chapter assumes you have met the methods; it does not re-teach any of them. If a name is unfamiliar, the family tree in Figure 16.1 will tell you which chapter to go back to.

What this chapter is for. Chapters 2–14 introduced methods one family at a time, in lecture order. That order is good for learning and bad for choosing. This chapter re-sorts everything onto shared axes so you can answer a practical question: *given my data and my goal, which family should I reach for, and what does that choice cost me?*

Read it in this order.

- *The family tree* (Figure 16.1) — every method in the book on one page, with arrows showing what descended from what. Start here; it is the map the rest of the chapter annotates.
- *The comparison tables* — the same methods scored on axes that actually differ between them.

- *The confusion clearinghouse* — the mistakes that recur across the whole field, collected in one place.
- *Decision guidance* (Figure 16.2) — a flowchart, plus eight heuristics.

Afterwards you will be able to place any new paper you read into the tree within a minute, and explain to someone else what it changed relative to its nearest neighbour.

Key Concept — The three questions every family answers differently

Strip away the architectures and every method in this book is a choice about what to do with deliberately damaged data. *Reconstruction* says: put the damage back, and whatever you needed to know in order to do that is the representation. *Pretext prediction* says: name the damage, and whatever you needed to know in order to recognise it is the representation. *Joint embedding* says: ignore the damage — produce the same answer for both damaged copies — and whatever survives that demand is the representation. The families differ not in power but in *what they force the network to keep*. Reconstruction keeps spatial detail. Joint embedding keeps whatever is invariant to the augmentations you chose. Pretext keeps whatever tracks the transformation you applied. Every empirical difference in the tables below traces back to that one sentence.

Chapters 2–14 presented methods one family at a time, in lecture order. That order is good for learning and bad for choosing. Someone with unlabelled data and a goal does not ask "what did Lecture 07 say?". They ask: *should I reconstruct, contrast, distil, cluster or generate — and what does each choice cost me?*

This chapter answers that in four ways:

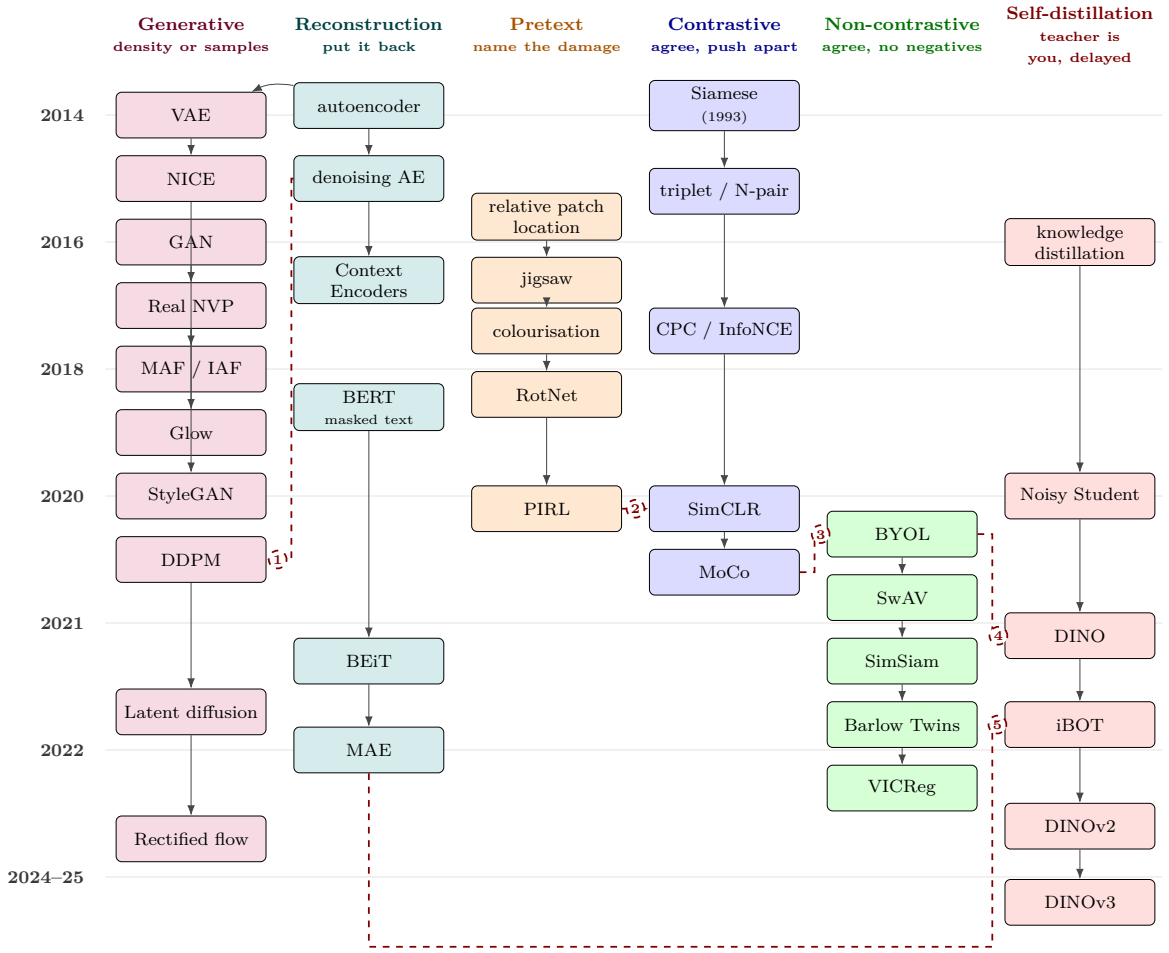
- comparison tables that put every method on the same axes (Section 16.2);
- the confusions that recur across the whole field, collected in one place (Section 16.3);
- decision guidance (Section 16.4);
- which evaluation to trust for which claim (Section 16.5).

It introduces no new method. The deeper question — *why* the families differ as they do — was the subject of Chapter 15.

16.1 Reading the comparison tables

Tables 16.1, 16.2, and 16.3 list every method this book covers, grouped by family, with one row per method and these shared columns:

- **Chapter.** Where in this book the method is treated in full.
- **Signal source.** What in the data produces the supervisory target — the single most informative axis of comparison.



Solid arrows are descent inside one family. **Dashed red arrows** are ideas crossing *between* families — and those crossings are where most of the progress in this book actually happened:

(1) denoising stops being one step and becomes a *schedule* of steps ⇒ diffusion. **(2)** stop *naming* the transformation and start *ignoring* it ⇒ contrastive learning. **(3)** drop the negatives and prevent collapse by asymmetry instead. **(4)** the slow EMA copy is promoted from a trick to a *teacher*. **(5)** add masked-patch prediction on top of a joint-embedding loss.

Figure 16.1. Every method in this book, arranged by family (columns) and time (rows). Solid arrows are descent inside one family; dashed red arrows are ideas jumping between families. The modern state of the art — DINOv2, latent diffusion — sits at the bottom of the page precisely because it inherits along several of these arrows at once.

How to read it: Pick a method you know and trace its arrows upwards. Then look at the red arrows: those are the moments when somebody borrowed from a different column, and they are worth more attention than the black ones.

- **Pairs?** Does training construct explicit pairs (or tuples) of related samples — two augmented views, an observed/corrupted pair, an image/caption pair? "–" means the objective is defined per single sample (or on the dataset as a whole for clustering).
- **Negatives?** Does the loss explicitly push *apart* samples it believes are different? This is the sharpest dividing line inside the joint-embedding family (Chapters 9–11).
- **Labels?** "No" means no human-provided category labels for the main objective. "Pseudo" means targets are generated by the model itself or by an automatic process (pseudo-labels, teacher outputs, hand transforms). Entries marked *(supervised)* require human labels — they appear because the course treats them as deliberate boundary cases (Pix2Pix, Diffusion Policy).
- **Strength / Key failure.** One honest line each. The failure lines are the more valuable column: they tell you what to suspect first when the method underperforms.

Two warnings before reading any such table. First, the rows are not commensurable: the entries span two decades, different backbones, and different data scales, so no column implies a performance ranking. Second, the columns suppress what makes each method distinctive; they are an index into the chapters, not a replacement for them.

16.2 Master comparison tables

Table 16.1. Classical methods and clustering hybrids (Chapter 2 foundations; Chapter 3). k-means and kNN themselves make no SSL claim but are reused by nearly every other family as pseudo-label machinery (DeepCluster, SwAV, DINOv3 data curation) and as evaluation readouts.

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
k-means (1957/67/82) (Steinhaus, 1956; MacQueen, 1967; Lloyd, 1982)	Ch. 2–3	Co-occurrence of nearby points in feature space	–	–	No	Zero assumptions, anchors the whole pseudo-label loop	Spherical clusters; seed sensitivity (mitigated by k-means++ (Arthur and Vassilvitskii, 2007)); K must be chosen
kNN (context)	Ch. 2	Neighborhood agreement	–	–	No	Non-parametric, zero training; default frozen-feature readout	Only as good as the metric and space it runs in; $O(N)$ query cost (mitigated by KD-trees (Bentley, 1975), LSH (Indyk and Motwani, 1998))

continues on next page

(continued)

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
DBSCAN / OPTICS (Ester et al., 1996; Ankerst et al., 1999)	Ch. 2	Local density	–	–	No	Finds non-spherical clusters, marks noise, no fixed K	Fails when cluster densities differ; ε tuning is fragile in high dimensions
DeepCluster (Caron et al., 2018)	Ch. 3	Cluster assignments used as pseudo-labels, iterated	–	–	Pseudo	First end-to-end proof that "better than chance" bootstraps	Can collapse to few/trivial clusters; restarts forget; quality bounded by the initial features
STEGO (Hamilton et al., 2022)	Ch. 3	Correspondence of pretrained features (segment-then-distill)	Yes	Yes	No	Sharp unsupervised segmentation by distilling frozen correspondences into a compact probe	Inherits the frozen features' granularity; CRF-like smoothing can oversmooth small classes

Table 16.2. Generative methods (Chapters 4–7; autoregressive interlude in Chapter 7). The shared bet: a model that can reproduce or denoise data must have understood its structure. The disagreement is over what the bet costs.

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
KDE (baseline)	Ch. 6	Data density itself	–	–	No	Explicit density, zero learning; the honest "how hard is this really" baseline	Curse of dimensionality: useless beyond tens of dimensions
Autoencoder (AE)	Ch. 4	Observed $\boldsymbol{x}$ as its own target ($\boldsymbol{x} \to \boldsymbol{z} \to \hat{\boldsymbol{x}}$)	Obs./ recon	–	No	Dead simple; bottleneck forces compression	Memorization: identity learned through enough capacity; latent space unstructured
VAE (Kingma and Welling, 2014; Rezende et al., 2014)	Ch. 4	Reconstruction plus a prior-matching KL term	Obs./ recon	–	No	Principled density lower bound; structured, samplable latent space	KL collapses on powerful decoders; reconstructions blurry

continues on next page

(continued)

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
MAE (He et al., 2022)	Ch. 5	Masked patches as targets (75% mask)	Obs./ recon	–	No	Trains at ViT scale; asymmetric decoder is cheap; excellent finetuning features	Weak linear probes; pixel targets waste capacity on imperceptible detail
PixelRNN (van den Oord et al., 2016)	Ch. 7	Chain rule: next pixel from previous ones	–	–	No	Exact likelihood; strong density estimates	Sequential generation; models texture as happily as content; enormous image cost
Real NVP (Dinh et al., 2017)	Ch. 6	Exact density via invertible change of variables	–	–	No	Exact, tractable likelihood and exact inversion	Architecture constrained to triangular Jacobians; features not the goal
GLOW (Kingma and Dhariwal, 2018)	Ch. 6	Same, plus invertible 1×1 convolutions	–	–	No	High-quality samples with an interpretable latent factorization	Likelihood track record on complex data is poor; memory-hungry at scale
GAN (Goodfellow et al., 2014; Mirza and Osindero, 2014)	Ch. 6	Discriminator-vs-generator game on real vs. fake	–	Implicit	No	Sharpest samples of its era; no likelihood needed	Training instability; mode collapse; no density, no encoder
Pix2Pix (cGAN) (Isola et al., 2017)	Ch. 6	*Paired* image-to-image correspondence	Yes	Implicit	*Supervised pairs*	Excellent paired translation (maps, edges to photos)	Requires aligned pairs: strictly not SSL at all — the course's boundary case
StyleGAN (Karras et al., 2019)	Ch. 6	GAN signal routed through a style/AdaIN pathway	–	Implicit	No	Disentangled, controllable latent directions	Inherits GAN instability; disentanglement is qualitative and human-judged
DAE (Vincent et al., 2008, 2010)	Ch. 7	Corrupted → clean pairs (explicit noise)	Yes (corr./clean)	–	No	First working recipe for learning by undoing corruption	Fixed, hand-chosen noise process; shallow benefit relative to later methods

continues on next page

(continued)

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
DDPM (Ho et al., 2020)	Ch. 7	Noise level t: predict the noise in $\boldsymbol{x}_t$	Obs./noisy	–	No	Stable training; state-of-the-art sample quality	Hundreds of sequential function evaluations to sample
Flow matching / rectified flow (Lipman et al., 2023; Liu et al., 2023)	Ch. 7	Velocity field $v = \boldsymbol{z} - \boldsymbol{x}$ along an interpolation path	Obs./ interpol.	–	No	Nearly straight sampling paths; training is a plain regression	The learned path is straight only approximately; per-sample v is stochastic in data
LDM / Stable Diffusion (Rombach et al., 2022)	Ch. 7	Diffusion composed in a VAE/GAN latent space	Obs./noisy	–	No	Resolution decoupled from diffusion compute; conditioning pipeline mature	Two-stage composition inherits errors of the autoencoder
DIFT (Tang et al., 2023)	Ch. 7	Free byproduct: intermediate diffusion features	–	–	No	Emergent semantic correspondence at zero training cost	Best layer/timestep per task must be found empirically; not a training objective
Diffusion Policy (Chi et al., 2023)	Ch. 7	Action denoising conditioned on observations	Yes	–	*Supervised demos*	Multimodal action distributions with stable regression-free training	Behavior cloning underneath: fidelity to demonstrations, no exploration

Table 16.3. Joint-embedding and self-training methods (Chapters 9–13). "Teacher" rows distill from a model output rather than data structure; "Pairs?" reads "Yes" whenever a second, related view/sample is constructed, including by a momentum encoder.

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
PIRL (Misra and van der Maaten, 2020)	Ch. 9	Instance identity, predicted invariant to jigsaw	Yes	Yes	No	Decoupled the representation from transformation equivariance	Instance discrimination conflates scene with class

continues on next page

(continued)

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
SimCLR (Chen et al., 2020a)	Ch. 9	Two augmented views of one image	Yes	Yes	No	Simple recipe; strong features with a clear augmentation story	Needs very large batches; heavy dependence on the augmentation policy; false negatives
MoCo v1/v2/v3 (He et al., 2020; Chen et al., 2020c, 2021)	Ch. 9	Views matched against a momentum-encoded queue	Yes	Yes	No	Batch-size independence via the queue; EMA teacher stabilizes keys	Queue+EMA machinery; stale keys if the queue drifts
BYOL (Grill et al., 2020)	Ch. 10	Predict one view's representation from the other	Yes	No	No	First competitive negative-free method; strong probes	Collapse prevention is architectural magic (EMA, predictor); theory contested
SimSiam (Chen and He, 2021)	Ch. 10	Same, with stop-gradient and no EMA	Yes	No	No	Minimalist proof that stop-grad suffices	Collapses without stop-grad or predictor; no stable-theory guarantee
Barlow Twins (Zbontar et al., 2021)	Ch. 10	Cross-correlation of two views driven to identity	Yes	No	No	Elegant redundancy reduction; no asymmetry needed	Large embedding dimension needed; brittle to batch statistics
VICReg (Bardes et al., 2022)	Ch. 10	Variance-invariance-covariance regularization	Yes	No	No	Explicit, decomposed collapse prevention	Three hyperparameters balance three forces
SwAV (Caron et al., 2020)	Ch. 10 (context)	Cluster assignments swapped between views	Yes	No (cluster)	Pseudo	Online clustering matched contrastive accuracy	Prototype management; needs careful multi-crop
iBOT (Zhou et al., 2022)	Ch. 11	Masked token prediction under self-distillation	Yes	No	No	Unifies MAE-style masking with DINO-style distillation; dense features	Complex composite objective
BERT (Devlin et al., 2019)	Ch. 11	Masked token identity (15% masking)	No	No	No	Defined the masked-modeling paradigm for language; bidirectional context	Pretrain/finetune mismatch; masking ratio geared to text, not images

continues on next page

(continued)

Method	Ch.	Signal source	Pairs?	Neg.?	Labels?	Strength	Key failure
DINO (Caron et al., 2021)	Ch. 11	Self-distillation: student matches EMA teacher across crops	Yes	No	No	Strong kNN features; emergent attention segmentation	Centering and sharpening must be balanced; fragile without both
DINOv2 (Oquab et al., 2024)	Ch. 11	DINO + iBOT + KoLeo on curated 142M images	Yes	No	No	Off-the-shelf universal visual features that match weak supervision	Data engine, not algorithm, does most of the work; expensive to replicate
DINOv3 (Siméoni et al., 2025; Vo et al., 2024)	Ch. 11	Same objective, scaled through dedupe/retrieve curation to 1,689M images	Yes	No	No	Scaling laws of SSL realized; state of the art across vision tasks	Curation pipeline is the method; inaccessible without infrastructure
DUNE (Sariyildiz et al., 2025)	Ch. 12	Feature match to three frozen heterogeneous teachers	Yes	No	No (teachers)	Compiles many teachers into one universal encoder	Ceiling set by the union of teachers
AM-RADIO (Ranzinger et al., 2024)	Ch. 12 (context)	Multi-teacher distillation with aggressive orchestration	Yes	No	No (teachers)	Efficient fused foundation encoder at deployment scale	Orchestration complexity; inherits teacher biases
Noisy Student (Xie et al., 2020)	Ch. 12	Pseudo-labels from a labeled-data teacher on unlabeled data	Yes	No	Pseudo	Student surpasses the teacher; simple loop	Semi-supervised, not pure SSL; needs labeled seed data
Lee-style pseudo-labels (Lee, 2013)	Ch. 12	The network's own most-confident predictions	–	No	Pseudo	Nearly free add-on to any classifier; recycles unlabeled data at zero cost	Confirmation bias: confident errors reinforce themselves
DINO-WM (Zhou et al., 2024)	Ch. 13	Action-conditioned prediction in frozen DINOv2 space	Yes	No	No	Zero-shot planning by imagining in a pretrained feature space	World model quality bounded by both the frozen features and the learned dynamics

Key Concept — Signal source

The **signal source** of a self-supervised method is whatever in the data produces the supervisory target: held-out input statistics (masked patches, next tokens, denoising levels, pixel densities — Chapters 4–7), hand-built invariances that define equivalence (pretext transforms — Chapter 8), correspondence between two views of one sample (Chapters 9–11), a model's own smoothed outputs (teacher distributions, pseudo-labels — Chapters 11–12), or the geometry of the dataset as a whole (clusters, graph neighborhoods — Chapters 2–3). It is the most predictive single column in Tables 16.1–16.3: methods sharing a signal source share their strengths and their characteristic failure modes, even when their objectives look nothing alike. When two methods disagree on everything else, it is usually the signal source that decided the comparison.

Key Concept — Collapse-avoidance mechanism

Any objective that asks two pathways (branches, networks, or views) to agree has a trivial optimum — *collapse*, the constant output that Chapter 10 defines — so every such method in this book is, implicitly, one **collapse-avoidance mechanism** plus machinery around it. Four families of mechanisms cover the tables: explicit repulsion against negatives (SimCLR, MoCo), architectural asymmetry (SimSiam's predictor plus stop-gradient), temporal asymmetry (the EMA teacher of BYOL and DINO, with centering and sharpening), and statistic-level anti-collapse terms (Barlow Twins' cross-correlation target, VICReg's variance-covariance terms). The mechanism — not the architecture or the loss shape — is the property that predicts how a method fails when it is weakened, which is why the failure-lines column of the joint-embedding table reads as a list of mechanisms rather than of models.

16.2.1 What the tables show, compressed

Three patterns survive all the detail.

1. Where the signal comes from predicts what the method is good at. Methods that learn from *matching things up* — two views, a teacher's output, a clean/noisy pair — give the best representations. Methods that learn to *model the whole distribution* — flows, GANs, PixelRNN — give the best samples. Almost nothing is best at both.

2. Negatives were a means, not an end. Every strong method without negatives puts something else in their place that does the same job: stop-gradient, an EMA teacher, centring, a variance penalty. Chapter 15 asks whether these replacements are truly equivalent.

3. At scale, the bottleneck moved from the loss to the data. Most of the gain from DINO to DINOv2 to DINOv3 came from better data curation, not a better objective (Siméoni et al., 2025; Vo et al., 2024; Oquab et al., 2024). That is why Chapter 14's systems view — build a pipeline that produces its own training signal — is arguably where the whole book was heading.

Failure Modes

The same word, different failure. Comparing failure modes across the families:

- **Masked reconstruction (MAE, Chapter 5):** the objective is satisfied by low-level texture statistics, so the representation can lag the reconstruction by a wide margin — pixel fidelity is easy to achieve without learning anything transferable.
- **Contrastive with negatives (SimCLR, MoCo, Chapter 9):** class collision — in-batch or queued neighbors are false negatives pushed apart — and representation drift if the EMA coefficient $\boldsymbol{m}$ of the key encoder strays from 1.
- **Non-contrastive joint embedding (SimSiam, BYOL, Chapter 10):** full collapse to constant output the moment the asymmetry weakens — predictor removed, stop-gradient dropped, or EMA teacher unstuck from its schedule.
- **Self-distillation (DINO, Chapter 11):** collapse is split in two — one dimension dominates the output (centering is the guard) or all dimensions go uniform (sharpening is the guard) — and either wins when its guard is mis-set.
- **Clustering (DeepCluster, Chapter 3; SwAV, Chapter 10):** degenerate partitions, in which one cluster absorbs most of the dataset; success depends on balanced assignment *and* on the encoder improving between re-clustering rounds.
- **Generative density models (KDE, flows, GANs, Chapter 6):** mode collapse (GANs) or smoothing away the modes (KDE) — the family's failures live in the samples, not in the features.

Read the key-failure column of Tables 16.1–16.3 through this list: the failures are family properties, not per-model accidents, so the first thing to suspect when a run underperforms is the family's characteristic failure, not the model's.

Key takeaway — how to read any comparison table in this book

Numbers are comparable only *within* an evaluation protocol and *within* a domain. A linear-probe number and a fine-tuning number are measuring different things; an ImageNet number and a medical-imaging number are measuring different worlds. Before comparing two rows, check that the protocol column and the data column match. Most published disagreements in this field dissolve once you do.

16.3 The confusion clearinghouse

Examinations, paper reviews, and lab discussions recycle a small set of conceptual errors about this field. This section corrects the eight that matter most, each with a pointer to the chapter where the full argument lives.

Common Confusion

SSL vs. unsupervised learning vs. semi-supervised learning. These are points on a supervision axis, not synonyms. *Unsupervised learning* is the umbrella: any learning from unlabeled data, historically dominated by density estimation (KDE, Chapter 6) and clustering (Chapter 2). *SSL* is the modern subprogram: construct an explicit input-target task from the input's own structure and train a deep network on it (Chapters 3–14), usually to obtain transferable features rather than to solve the constructed task. *Semi-supervised learning* assumes a small labeled set *plus* unlabeled data and optimizes the labeled task throughout (Chapter 12). Noisy Student (Xie et al., 2020) is *semi-supervised*: its teacher is trained on labels, and its loop is only well-defined with them. DINOv2 pretraining followed by a labeled linear probe is SSL used *for* a downstream task; the probe stage alone would be what the literature calls having a small labeled budget — informally "semi-supervised evaluation" but not the semi-supervised *training* algorithms of Chapter 12. Quick rule: if any label touched any loss, it is not SSL; if SSL features are only probed with labels, the labels touched evaluation, not training.

Common Confusion

Generative vs. discriminative objectives. These name two different things the objective can ask for, and both can be self-supervised. A *generative objective* models the data distribution itself — an explicit density (KDE, flows, PixelRNN), an implicit sampler (GANs), or a denoising procedure (DDPM, flow matching); the target has the dimensionality of $\boldsymbol{x}$ (Chapters 4–7). A *discriminative objective* models a decision — which of two views matches this one (SimCLR), which cluster this belongs to (DeepCluster, SwAV), which rotation was applied (RotNet) — and the target is low-dimensional (Chapters 3, 8–9). Masked reconstruction (MAE, BERT) sits between: generative machinery applied to part of the input. The confusion arises because VAEs use a discriminative encoder and discriminative methods generate pseudo-labels. The criterion is always what the *loss compares*, not which architecture computes it. The empirical consequence, from the tables: generative objectives buy sample quality and density access; discriminative objectives buy feature quality; masked modeling buys scaling behavior. Nobody yet buys all three at once.

Common Confusion

Contrastive vs. non-contrastive vs. self-distillation. All three are joint-embedding methods: align two views of one sample. *Contrastive* methods add an explicit repulsion term against other samples (negatives): SimCLR against the batch, MoCo against the queue (Chapter 9). *Non-contrastive* joint-embedding methods drop negatives and prevent collapse by construction — predictor plus stop-gradient (SimSiam, BYOL) or batchwise regularizers (Barlow Twins, VICReg) (Chapter 10). *Self-distillation* is a special alignment structure: a student trained to match a *teacher* that is the student's own exponential

moving average, with asymmetries (centering, sharpening, crops) between the two pathways — DINO (Chapter 11). So BYOL is simultaneously non-contrastive *and* self-distilling (EMA teacher), while SimSiam is non-contrastive but not distillation (no EMA). The axes are: negatives? and momentum-teacher? — giving the 2×2 that organizes Chapters 9–11. Saying "contrastive" when you mean "joint-embedding" is the spoken form of this error and it hides a real design choice.

Common Confusion

Invariance vs. equivariance. A representation f is *invariant* to a transformation T if $f(T(\boldsymbol{x})) = f(\boldsymbol{x})$: the output does not change. It is *equivariant* if $f(T(\boldsymbol{x})) = T'(f(\boldsymbol{x}))$ for a predictable companion transformation T': the output changes in step. Joint-embedding objectives (Chapters 9–11) explicitly *train invariance to the augmentation family* $\mathcal{T}$ — augmented views are pulled together. RotNet (Chapter 8) does the opposite: it requires the feature to *encode* the transformation, because the rotation must be predicted from it. Both are SSL with the same augmentations; the objective, not the augmentation, decides which property is learned (see the next box). Whether to want invariance or covariance is a *downstream* decision: invariance to pose helps ImageNet classification and can actively hurt pose-aware reading of medical scans. The safe statement is historically tight: SSL features inherit the invariances of their training augmentations, no more and no less — which is why cropping policy (multi-crop in SwAV/DINO) matters as much as the loss.

Common Confusion

Augmentation as augmentation vs. augmentation as supervision. In supervised learning, augmentation enlarges an effective dataset: the labeler said "cat", augmentation says "still a cat" — augmentation is pure regularization. In SSL, augmentation *is* the supervisory signal: for SimCLR, the label of a pair is "same image, different augmentation" (Chapter 9); for MAE, the mask defines what is missing (Chapter 5); for RotNet, the rotation *is* the label (Chapter 8). The confusion matters because conclusions do not transfer between the two roles. Choosing stronger supervised augmentation is usually safe; choosing different SSL augmentations changes *what is learned*, sometimes catastrophically — grayscale jitter in SimCLR removes color as a discriminative shortcut (Chen et al., 2020a), while the same jitter in MAE merely regularizes, since color must be reconstructed. When someone reports that an augmentation "improved SSL," always ask: improved which role — the definition of identity (contrastive), the difficulty of reconstruction (generative), or the decision task (pretext)?

Common Confusion

Pretext loss vs. downstream loss. The *pretext* task is the synthetic objective trained on unlabeled data (predict rotation, reconstruct pixels, align views); the *downstream* task is what you actually care about (classify, detect, plan (Zhou et al., 2024)). Good

numbers on the pretext task imply almost nothing about downstream performance. A rotation predictor can reach high accuracy using color histograms of a corner patch; a contrastive model can lower InfoNCE by encoding augmentation artifacts. This is why the book uses two separate evaluations at different stages (Chapters 1, 8): pretext accuracy as a *sanity signal during training* (is anything being learned at all?) and only linear-probe/kNN/finetune numbers as the actual claim. The failure of the field's first decade was often the conflation — reporting the pretext number — and its discipline since has been to standardize the downstream protocol (Oquab et al., 2024; Siméoni et al., 2025).

Common Confusion

Clustering-foundations (Chapter 2) vs. clustering-this-book (Chapter 3). Lecture 02 deliberately teaches clustering twice at different levels, and the exam has reportedly caught students on it. Chapter 2 presents k-means (Lloyd, 1982), DBSCAN (Ester et al., 1996), and neighbors as *algorithms on a fixed space*: given vectors, partition them. Chapter 3 presents *deep clustering*: the vectors are themselves learned, and the clustering feeds back as pseudo-labels (DeepCluster (Caron et al., 2018)) or is distilled into a segmentation head (STEGO (Hamilton et al., 2022)). The distinction is the loop. Asking "is DeepCluster better than k-means?" is a category error: k-means runs *inside* DeepCluster once per pseudo-label round over the current features, which are the thing being improved. The right comparison questions are about the loop: does the re-clustering converge; can the pseudo-labels freeze a bad partition; how does the initial feature space bias the fixed point (Chapter 3, Failure Modes)?

Common Confusion

Is a world model self-supervised? This is Lecture 08's debate (Chapter 13), and the honest answer is: the title is defensible either way, but the interesting versions are SSL by construction. A *world model* predicts future observations (or rewards) from current ones, optionally conditioned on actions. The case *for* SSL: the learning signal comes entirely from the stream of experience — next frames, next states — with no external annotation, exactly as in Chapters 5, 9, 11. DINO-WM (Zhou et al., 2024) fits this cleanly: prediction lives in frozen DINOv2 space, and only dynamics are learned, from unlabeled interaction. The case *against*: classical model-based RL learns a model *in service of reward*, just as all supervised learning serves its label — "world model" is a system role, not a training paradigm, and Diffusion Policy (Chi et al., 2023) shows the same architecture running purely supervised. Useful rule: ask where the targets come from. If from the environment's unlabeled stream — plausibly SSL; if from demonstrations or rewards — not. The debate is only worth having about representation quality, not definitions.

16.3.1 Smaller confusions worth one line each

Five shorter ones come up again and again in paper reviews and exams:

- **"DINO's EMA teacher is temporal ensembling, so DINO is semi-supervised."** False: DINO never sees a label (Chapter 11; the semi-supervised boundary is in Chapter 12).
- **"VAEs and contrastive learning both maximise mutual information."** The VAE's ELBO bounds a *likelihood*, not the information $I(X;Z)$ in a representation. And the informal "MI view" of InfoNCE is a bound with problems of its own (Chapter 15; Poole et al. (2019)).
- **"MAE is just one-step diffusion."** Both remove noise, but that shared shape does not make them one family: the noise, the time scales and the uses all differ (Chapters 5, 7).
- **"The reparameterisation trick removes the randomness from the VAE."** No: $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ is exactly as random, with exactly the same distribution. The randomness is only *moved* into an input $\boldsymbol{\epsilon}$ so backprop can reach $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. The alternative, REINFORCE, is also unbiased — it is just far noisier (Section 4.6.2).
- **"k-means needs labels to evaluate."** No: Chapter 2's internal/external evaluation split is a different thing from the SSL probing protocol of Chapter 1.

16.4 Decision guidance: which family first

Failure Modes

The four ways practitioners pick the wrong family.

(1) Demanding invariance to the thing you are trying to measure. The most expensive mistake in the book, and the easiest to make, because the default recipe is a joint-embedding method with the standard augmentation set. If your task is counting cells, estimating pose, reading a dial or judging orientation, then rotation and cropping are not nuisances — they *are* the label. An invariance loss will systematically destroy the signal, and your loss curve will look perfectly healthy while it does so.

(2) Judging a frozen-feature method by a fine-tuning number, or vice versa. MAE looks mediocre under a linear probe and excellent after fine-tuning; DINOv2 looks superb frozen and gains comparatively little from fine-tuning. Comparing one method's probe number against another's fine-tuning number is not a close call — it is meaningless.

(3) Assuming the leaderboard transfers to your domain. Every headline number in this book was measured on web photographs. Medical scans, satellite imagery, industrial inspection and microscopy have different statistics, different invariances and different notions of "same object". Method rankings are not stable across that gap.

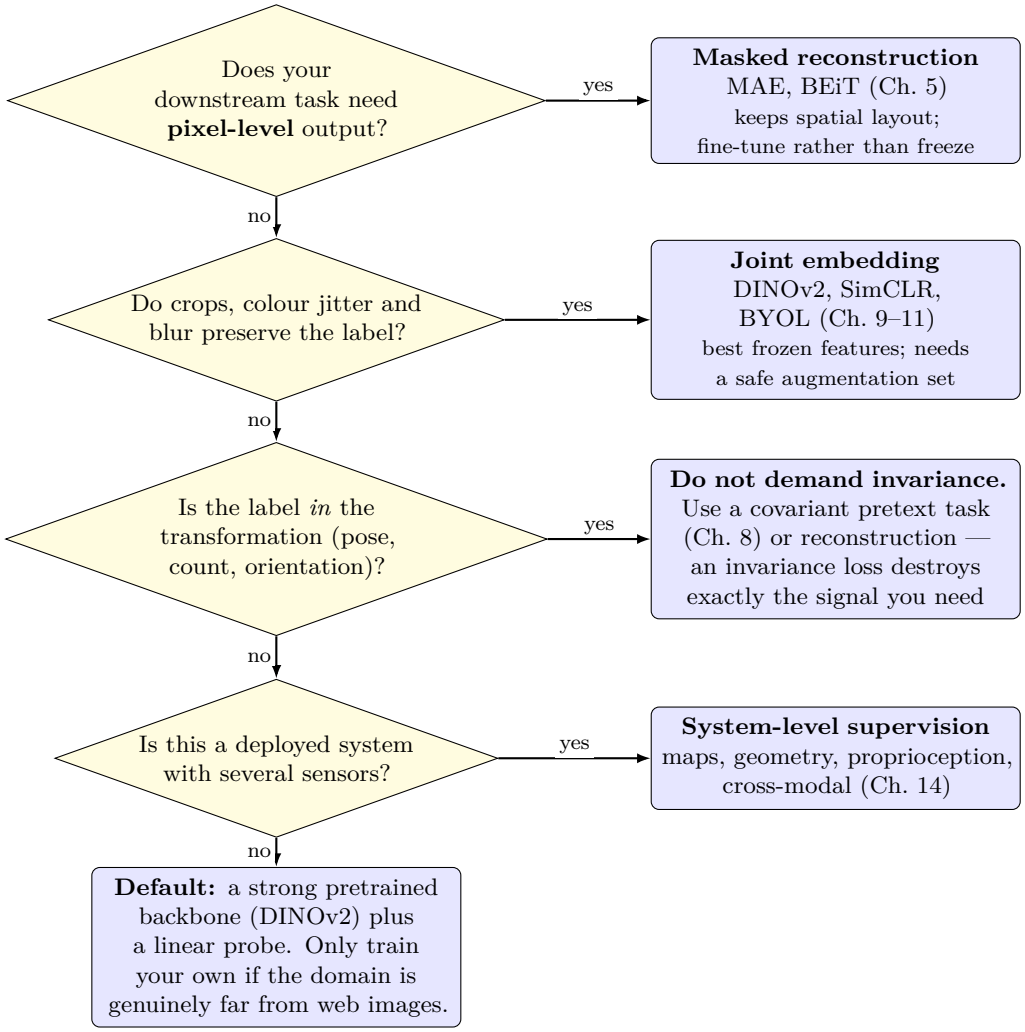


Figure 16.2. Choosing a family, as a sequence of questions about your problem rather than about the methods. The questions are ordered so that the ones that rule out the most options come first.

How to read it: Start at the top and answer honestly about your *data. The second question is the one people get wrong most often.*

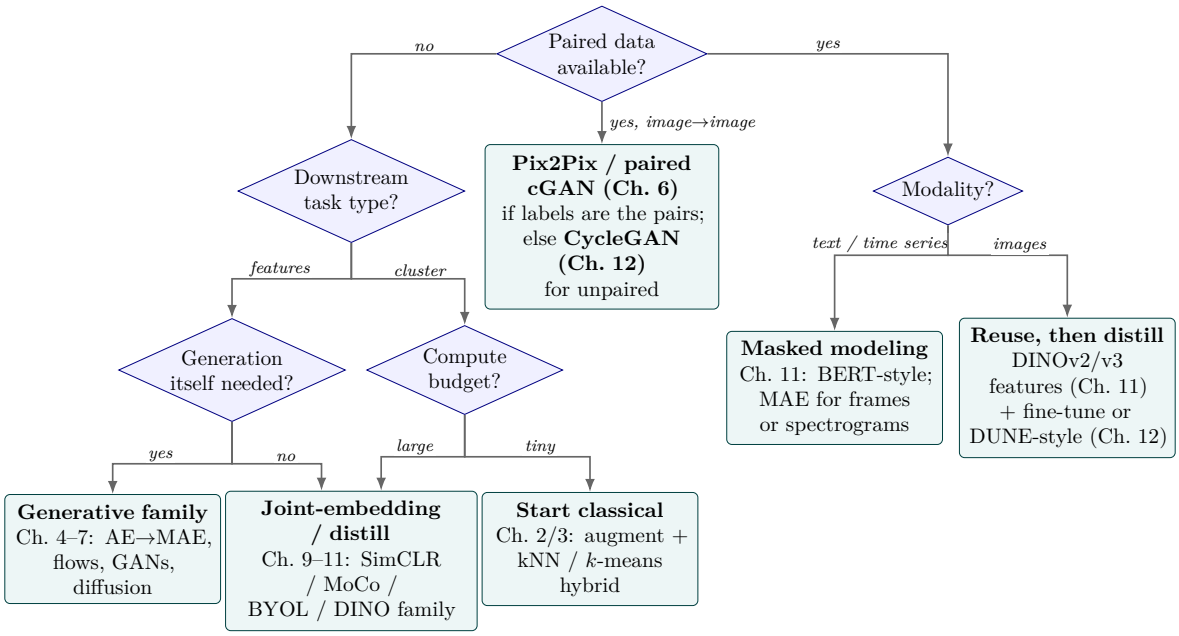


Figure 16.3. Which family should I try first? The flowchart decides a *starting* family; heuristics (H1)–(H8) below refine the choice, and Tables 16.1–16.3 pick the method inside the family. "Paired data" means aligned input–target or input–input pairs (pix2pix-style); demonstration pairs for Diffusion Policy belong to Chapter 7/12, not this chart.

(4) Training your own backbone when you did not need to. For most problems, a strong public backbone plus a linear probe beats anything you can pretrain on your own data budget. Pretraining from scratch is justified when the domain is genuinely far from the public pretraining distribution — and that is a claim worth testing before it is a claim worth acting on.

The table columns answer "what does it do"; this section answers "what should I do." Figure 16.3 reduces the choice to four questions — modality, paired data, compute, downstream type — and the heuristics below give the fuller, less tidy version. Use both: the flowchart picks a starting chapter, the heuristics refine, and the tables then let you comparison-shop inside the family.

16.4.1 Eight heuristics

1. **Buy before you build.** If pretrained DINOv2/DINOv3 or MAE features exist for your modality (Chapter 11), run the Chapter-1 frozen readouts (kNN, linear probe) on them *first*. Hours, not weeks. If they already clear your bar, SSL training was never the bottleneck — and you will know your floor before spending compute.
2. **Limited labels change the answer.** With a small labeled set plus unlabeled data, the semi-supervised tools of Chapter 12 — pseudo-labels (Lee, 2013), Noisy Student

(Xie et al., 2020) — are usually the cheapest real gain, before any SSL pretraining. Only when unlabeled data dwarfs the labeled set by orders of magnitude does pure-SSL pretraining dominate.

3. **If you must generate, the choice is likelihood vs. sharpness vs. speed.** Flows (Chapter 6) give exact density; autoregressive models (Chapter 7) give the best per-dimension density at sequential cost; diffusion and flow matching (Chapter 7) give sharp samples from an iterative procedure; GANs (Chapter 6) give speed and sharpness with no density and no encoder. Picking within the generative family is a constraint-satisfaction problem, not an accuracy league table.
4. **Budget decides architecture inside a family, not the family.** A single GPU is enough for a small SimCLR (with reduced batch, Chapter 9) or MoCo (queue substitutes for batch), and for MAE with its asymmetric decoder (Chapter 5). DINOv3-scale training is not an algorithm purchase, it is a data-engineering purchase (Chapter 11); do not pay for the objective and then cut the curation.
5. **Choose your invariant to match your task, not your religion.** Class-level recognition favors heavy augmentation invariance (Chapter 9–11); correspondence, segmentation, and 3D structure favor equivariance-preserving or correspondence-distilling signals — STEGO (Chapter 3), DIFT (Chapter 7), dense iBOT features (Chapter 11). Decide from the downstream task column in Table 16.4, not from which paper is newest.
6. **Text and time series default to masking.** The masked-token objective (BERT, Chapter 11; MAE, Chapter 5) is the robust default wherever a discrete or patchable input admits corruption; it scales and transfers, and its failures are documented rather than mysterious. Deviate only for reasons you can name.
7. **Debug the loop, not the number.** Iterative methods — DeepCluster (Chapter 3), self-training (Chapter 12) — have moving targets, so a single accuracy curve tells you nothing. Diagnose with the tools of Chapter 2/3: pseudo-label purity, cluster sizes, neighbor agreement; and with the collapse monitors of Chapters 10–11 (embedding variance, teacher entropy). Fix the signal before re-running.
8. **If your system produces its own data, think system-level.** Robots, simulators, and deployed services emit paired modalities, physical-law constraints, and uncertainty — the three axes of Chapter 14 — and the course's closing case studies show that the marginal bit of signal there beats re-tuning a standard objective. Before adding GPUs, audit what your system throws away (Frey et al., 2022; Zou et al., 2018; Frey et al., 2023; Rivera Valverde et al., 2021; Sun et al., 2022).

Key takeaway — the decision, compressed

Dense output needed? Reconstruction. Safe augmentations available? Joint embedding. Label lives in the transformation? Neither — go covariant. Deployed system with several sensors? System-level supervision. And in most real cases: take a strong public backbone and put a linear probe on it, and only pretrain your own when you can show the domain is genuinely far from what that backbone saw.

Table 16.4. Family → default frozen-feature readout(s) → why, following Chapter 1's evaluation taxonomy. "Linear probe" means a linear classifier trained on *frozen* features; "finetune" is listed only where it is the conventional primary readout.

Family	Default readout(s)	Why these
Classical / clustering (Ch. 2–3)	Cluster metrics vs. labels (NMI, ARI); neighbor agreement	The output *is* a partition, so evaluate the partition; label agreement is used because unsupervised quality has no ground truth
Masked reconstruction (Ch. 5, Ch. 11 BERT)	Finetune; (linear probe as diagnostic)	Pixel targets under-use global semantics (Ch. 5); finetuning releases what probing structurally cannot read
Contrastive joint-embedding (Ch. 9)	Linear probe; kNN	Objective trains linear separability of instances, which transfers to classes; kNN sanity-checks the feature geometry without any training
Non-contrastive / self-distillation (Ch. 10–11)	kNN; linear probe; dense-task transfer	These methods' selling point is off-the-shelf feature structure — emergent neighborhoods and attention maps (Caron et al., 2021; Oquab et al., 2024) — so the frozen readouts are the headline
Self-training / distillation (Ch. 12)	Downstream accuracy after finetune; student-vs-teacher gap	The claim is about a deployed student, not about a frozen space
Density estimators: KDE, flows, autoregressive (Ch. 6–7)	Held-out likelihood / bits-per-dimension; sample FID-style scores	The objective is the density, so evaluate the density; features were never the promise
Implicit generative: GANs (Ch. 6)	Sample quality (FID-type), precision/recall of modes; downstream only as a bonus	No encoder, no likelihood: the artifact is the sampler, and mode coverage is the failure to check (Ch. 6)
Denoising families: DAE, diffusion, flow matching (Ch. 7)	Sample quality + FID; derivative readouts for free features (DIFT correspondence, Diffusion-Policy success rate)	The training signal is the denoising step; evaluation splits between the generative artifact and the emergent byproducts
World models (Ch. 13)	Zero-shot planning performance in representation space (Zhou et al., 2024)	A world model's value is decision quality, so evaluate decisions the frozen features make possible, never the prediction loss itself

16.5 Evaluation cross-reference

Chapter 1 (1 Section 1.7) defines the readouts; here is which combination each family has converged on, and why. One caveat up front: evaluation is part of the method's identity. Comparing MAE to DINOv2 only on linear probes is not neutral — it tests the one metric MAE's designers did not optimize for (Chapter 5) and DINOv2's designers did (Chapter 11).

The pattern worth remembering: *evaluate the thing the family claims to produce, and one frozen-feature readout besides.* Mismatched readouts – testing a GAN on linear probes, testing MAE on kNN alone, testing DINO-WM on reconstruction error — have produced most of the spurious "method A beats method B" headlines in this literature.

16.6 Summary

The three tables of Section 16.2 compress the book onto one page each; the confusion boxes compress its review meetings onto one section. The decision guidance is deliberately specific but deliberately modest: it chooses a starting family, not an endpoint. What it cannot choose is the empirical question Chapter 15 leaves open — *why* the families differ as they do — because the honest current answer is: for reasons that are partly theory, partly engineering, and partly history. That is not a flaw of this chapter. It is the subject of research, and Part III's point is that you now have the vocabulary to participate in it.

Check Your Understanding

Q1. (Level 1) Without looking: name the *signal source* and the *negatives?* column value for SimCLR, BYOL, MAE, and DDPM. Why are these two columns enough to place all four into the right table?

Q2. (Level 1) A colleague says: "We are doing semi-supervised learning: we have 2000 labeled images and 2 million unlabeled ones, and we will pretrain with DINO then finetune." Using the first clearinghouse box, correct the terminology precisely.

Q3. (Level 2) A review claims VICReg is "contrastive without pairs." Use Tables 16.3 and the third clearinghouse box to write the reviewer's correction in two sentences.

Q4. (Level 2) Explain why "invariance" is a property of a representation relative to a transformation set, not a property of a method — and give one method (RotNet) whose SSL objective requires the opposite.

Q5. (Level 2) Using Figure 16.3: a hospital has 500k unlabeled chest X-rays, 3k labeled ones, one A100, and wants pneumonia classification. Trace the flowchart, then check against heuristics H1–H2 and Table 16.4: name the primary readout you would report.

Q6. (Level 3) The tables place MAE *and* DIFT in the generative chapter-neighborhood, yet one trains representation quality and one exploits a frozen generative model. Articulate the "is generation the goal or the instrument" axis and place MAE, DDPM, DIFT, and KDE on it.

Q7. (Level 3) An ablation removes SimCLR's color jitter and linear-probe accuracy rises on a benchmark whose classes differ mainly in colour. Using the augmentation-as-supervision box, explain why this does *not* contradict Chapter 9's claim that jitter helps.

Q8. (Level 3) DeepCluster and k-means are compared in a student report, which concludes "k-means is faster, so it wins." Write the two-sentence rebuttal the seventh box licenses.

Q9. (Level 4) Section 16.5 warns that "evaluate the thing the family claims to produce" still leaves one systematic bias across families. Identify it using the timeline implicit in the tables — what do newer families evaluate that older ones could not?

Q10. (Level 4–5) Take the flowchart seriously as a falsifiable artifact: give one real scenario (data, budget, task) where following it leads to the empirically *wrong* first choice, and propose the missing question node. Defend your node with a reference to a chapter's evidence.

Q11. (Level 5) Chapters 15 and 16 disagree in emphasis: one asks *why* families work, the other *when to pick* them. Show, with one worked example (e.g., SimSiam vs. SimCLR on small data), why answering the "when" question well currently *requires* progress on the "why" question, not just more benchmarks.

Solution sketches.

Q1. SimCLR: two augmented views, negatives yes; BYOL: view prediction, negatives no; MAE: masked patches (an observed/reconstructed pair), negatives no; DDPM: denoising at level t (a clean/noisy pair), negatives no. Strength/failure column differences follow from signal source; only the joint-embedding family asks the negatives question.

Q2. This is SSL pretraining *plus* standard supervised finetuning. Semi-supervised (Chapter 12) would mean a training loop where labels and unlabeled data interact — pseudo-labels, teacher–student. The DINO stage is SSL; the finetune stage is supervised; nothing is semi-supervised.

Q3. VICReg *has* pairs (two views) but no negatives; "contrastive" requires explicit repulsion. The right label is joint-embedding with batchwise regularizers (Chapter 10); the 2×2 from the box fixes it.

Q4. Invariance is $f(T(\boldsymbol{x})) = f(\boldsymbol{x})$ for T in a chosen $\mathcal{T}$ — you cannot ask "is SimCLR invariant" without saying to what. RotNet (Chapter 8) demands the rotation be *recoverable* from features: covariance, not invariance, with the same augmentation family.

Q5. Flowchart: no pairs → features → generation not the goal → joint-embedding. Realistically: heuristic H1 (try pretrained DINOv2/v3 or a medical MAE first), then H2 (use the 3k labels via pseudo-labeling, Chapter 12). Readout per Table 16.4: frozen linear probe + kNN for any SSL candidate, then supervised finetune for the deployed number.

Q6. KDE/DDPM: generation *is* the deliverable. MAE: generation is the instrument; the deliverable is features (Chapter 5). DIFT: generation happened elsewhere; the deliverable is correspondence inside a frozen generative model (Chapter 7) — delinked from generation cost.

Q7. Jitter defines what counts as "same image"; removing it makes color a discriminative shortcut, lifting probe accuracy on any benchmark where color is informative (Chapter 9). The augmentation changed the learned semantics; accuracy moved because the task's invariances moved.

Q8. DeepCluster contains k-means as its pseudo-label subroutine run on *learned* features; comparing runtimes compares an inner step with the outer loop (Chapter 3). The meaningful comparisons are feature quality and fixed-point behavior, not speed.

Q9. The bias is historical accumulation: newer families (diffusion, world models, systems) evaluate decisions, correspondence, deployment — artifacts older families could not produce — so cross-family tables implicitly grade older methods on axes chosen by their successors. Mitigation: always pair the family-native readout with one frozen probe.

Q10. Example: abundant cheap labels in deployment (a factory with a QA line emitting pass/fail on every unit) — the chart routes to a self-supervised start when a supervised or semi-supervised baseline would win (Chapter 12, heuristic H2). Missing node: "does the running system produce labels or paired modalities as a byproduct?" — Chapter 14's first axis.

Q11. SimSiam's stop-gradient trick works where the collapse mode is avoidable by asymmetry; whether it beats SimCLR on small data depends on batch-statistics theory (Chapters 9–10), which is exactly the kind of claim Chapter 15 labels hypothesis or theorem case-by-case. The "when" table inherits the "why" uncertainty; a decision rule is only as stable as the explanation beneath it.

Concept Map — how this chapter connects

Backward. This chapter is a rearrangement of material owned elsewhere: Chapter 1 (1) supplies the evaluation taxonomy used in Section 16.5; Chapter 2 (2) the k-means/kNN/DBSCAN rows of Table 16.1; Chapter 3 (3) DeepCluster and STEGO; Chapters 4 (4), 5 (5), 6 (6), and 7 (7) every generative row — AE/VAE, MAE, KDE, flows, GAN and its variants, DAE, DDPM, flow matching, latent diffusion, DIFT, Diffusion Policy; Chapter 8 (8) RotNet, augmentation-as-supervision, and the pretext/downstream split; Chapter 9 (9) PIRL, SimCLR, MoCo; Chapter 10 (10) BYOL, SimSiam, Barlow Twins, VICReg, SwAV; Chapter 11 (11) DINO, DINOv2/v3, iBOT, BERT as cross-modality masked modeling; Chapter 12 (12) pseudo-labels, Noisy Student, DUNE, AM-RADIO, and the semi-supervised boundary; Chapter 13 (13) the world-model debate and DINO-WM; Chapter 14 (14) the system-level signal audit underlying heuristic H8; Chapter 15 the "why" that this chapter's "when" rests on. **Sideways.** The clearinghouse boxes are deliberately redundant with the per-chapter *Common Confusion* boxes: a confusion caught twice, at first exposure and at synthesis, stays caught. **Forward.** Chapter 29's exercises recycle this chapter's tables and flowchart as raw material (match-method-to-scenario, classify-a-new-paper), and the appendices' cheat sheets are this same content rearranged for lookup rather than argument. If you remember one artifact from this chapter, keep Table 16.4: readouts, not logos, are what distinguish the families.

Part IV

Tutorials, Quizzes, Code Labs and the Exam

CHAPTER 17

Tutorial Quiz: Autoencoders, VAEs and Masked Autoencoders

In this chapter

17.1 The six ideas the quiz tests 763
17.2 Questions 1–9: knowledge 765
17.3 Questions 10–20: case studies 769
17.4 The quiz on one page 776

Chapter roadmap — Part IV: the tutorials, quizzes and code labs

What Part IV is. The course runs a tutorial alongside the lectures. Each tutorial came with a quiz (with an official answer sheet) and some with a coding notebook or case study. Part IV goes through all of them in the order of the course, then solves the first-cycle exam and builds a question bank in its pattern, and explains every answer from the ground up, so that you understand *why* the right option is right, not just which letter it is.

Chapter	Tutorial material	Lecture chapters
17	Week 3 quiz: autoencoders, VAEs, MAE (20 questions)	4, 5
18	Coding tutorials 1 and 2: a VAE and a tiny MAE on CIFAR-10	4, 5
19	Generative models quiz: KDE, flows, GANs (10 questions)	6
20	Tutorial 5 quiz: denoising and diffusion (16 questions)	7
21	Tutorium 6 quiz: pretext tasks and contrastive learning (18 questions)	8, 9
22	Week 8 quiz and lab: representation learning, SimSiam, DINO (15 questions)	10, 11
23	Week 9 quiz and case study: self-training, cycle consistency, world models (18 questions)	12, 13
24	System-level self-supervision quiz (10 questions and a bonus)	14
25	The first-cycle exam, reconstructed and solved	all
26	Question bank I: 60 multiple-choice questions, theory and calculations	all
27	Question bank II: 22 coding-on-paper templates	all
28	Question bank III: 18 case studies and a 120-point mock exam	all

How every question is explained. Each question is printed in a blue box with its options, followed by the official answer. Then: *The idea* explains the answer in plain words, often with a small example; *Why not the others* says in one line what is wrong with each distractor; and *In the book* points to the section that teaches it. Where the official answer sheet is imprecise or has a typo, a short note says so.

How to use it. Cover the answer, pick an option, and write one sentence of reasoning *before* reading on. Quizzes like these are designed so that two options sound right; the one sentence is what separates them.

Chapter roadmap — this chapter

The quiz. Twenty single-choice questions from the week 3 tutorial. Questions 1–9 test knowledge (what is the loss, what does the KL term do, why the reparameterisation trick). Questions 10–20 are short *case studies*: a symptom is described, and you must name the cause or the fix.

What you need. Chapter 4 (autoencoders, the information bottleneck, VAEs and the ELBO) and Chapter 5 (masked autoencoders). The six ideas on the next page are enough to answer all twenty.

17.1 The six ideas the quiz tests

Almost every question in this quiz is one of six ideas in disguise. Learn these, and the quiz becomes a recognition exercise.

1. **An autoencoder is trained only to copy.** Encoder f_θ squeezes x into a code $z = f_\theta(x)$; decoder g_ϕ rebuilds $\hat{x} = g_\phi(z)$; the loss is the reconstruction error $\|x - \hat{x}\|^2$. Nothing else is in the loss.
2. **The bottleneck decides what is learned.** Squeezing through a small code forces the network to keep what is *common* across the dataset (shapes, objects) and drop what is not (sensor noise). Three dials control the squeeze: latent size, network capacity, and regularisation.
3. **A small code is not a guarantee.** A big enough network can give each training image its own "index number" in even a one-dimensional code and decode it like a lookup table. That is memorisation, not understanding.
4. **A VAE adds a map of the code space.** It makes the encoder output a *distribution* $q_\phi(z \mid x)$ and pulls it towards a fixed prior $p(z) = \mathcal{N}(0, I)$ with a KL term. Then every point you draw from $\mathcal{N}(0, I)$ decodes to something sensible: the model can generate.
5. **The ELBO is a lower bound because the true likelihood needs an impossible integral.** We cannot compute $\log p_\theta(x) = \log \int p_\theta(x \mid z)\, p(z)\, dz$, so we maximise a computable quantity below it.
6. **MAE hides most of the image and asks for it back.** The big encoder sees only the visible patches; a small decoder fills in the rest. Hiding a lot (75%) makes the task hard enough to need real understanding.

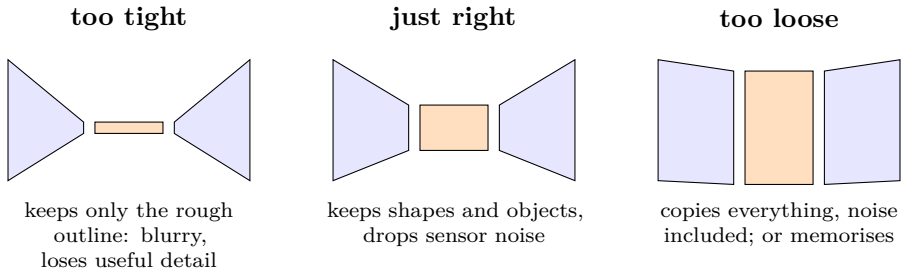


Figure 17.1. The bottleneck dial behind Questions 2, 3, 6, 10, 11 and 13. Blue: encoder and decoder; orange: the code. The same reconstruction loss gives very different representations depending on how much can pass through the middle. A large network can still cheat a narrow code by memorising (Question 6), so "capacity" belongs on the dial too.

How to read it: Read left to right as "more and more passes through". The quiz's good answers are always the middle panel.

17.2 Questions 1–9: knowledge

Q1 Standard autoencoder objective

An undercomplete autoencoder has encoder f_θ, decoder g_ϕ, input x and code $z = f_\theta(x)$. Which objective best describes its basic training goal?

A. Maximise mutual information between x and every hidden activation.
B. Minimise reconstruction error between x and $g_\phi(f_\theta(x))$.
C. Minimise KL divergence between $q_\phi(z \mid x)$ and a uniform class distribution.
D. Maximise the entropy of the latent representation.

Answer: B — minimise the reconstruction error.

The idea. Follow the data: x goes in, z is the squeezed code, $\hat{x} = g_\phi(f_\theta(x))$ comes out. The only thing the training can compare is $\hat{x}$ with x, so the loss is their difference, usually $\|x - \hat{x}\|^2$. "Undercomplete" just means the code is smaller than the input, which is what makes copying non-trivial.

Why not the others. A: mutual information is a way to *analyse* what layers keep, not the loss that is trained. C: a KL term belongs to a VAE, and it pulls towards a Gaussian prior, not a class distribution (there are no classes). D: random codes have maximal entropy and reconstruct nothing.

In the book. §4.1.

Q2 The hyperparameters that matter

Which hyperparameters most directly control whether an autoencoder compresses meaningfully or simply memorises the dataset?

A. Latent dimensionality, encoder/decoder capacity, and regularisation.
B. Batch size only.
C. The number of output classes and label smoothing.
D. The test-set labels and classifier head dimension.

Answer: A — latent size, capacity, regularisation.

The idea. These are exactly the three knobs of the bottleneck dial (Figure 17.1): how wide the code is, how clever the networks on either side are, and what extra pressure (noise, sparsity, a KL term) pushes against simply copying.

Why not the others. B: batch size changes how smoothly training goes, not how much can pass through the code. C and D: classes, labels and classifier heads do not exist during autoencoder training; they are evaluation-time things.

In the book. §4.2, §4.3.

Q3 What the bottleneck should do

Which statement best captures the desired role of the latent bottleneck?

A. Preserve every detail of the input, including random noise.

B. Remove all information except the class label.

C. Retain information useful for explaining the dataset while suppressing redundancy and noise.

D. Always be one-dimensional to guarantee abstraction.

Answer: C — keep what explains the dataset, drop redundancy and noise.

The idea. Picture a thousand photos of the same cup on a table. What they share (the cup's shape, the table) is worth storing once in the code; the grain of the camera sensor differs in every photo and cannot be predicted from the others, so a good code drops it. That is the *information bottleneck* view.

Why not the others. A describes copying, not abstraction. B needs a label the autoencoder never sees. D: a one-dimensional code does not guarantee abstraction — see Question 6.

In the book. §4.2.

Q4 The KL term in the VAE

What is the main purpose of the KL divergence term in the ELBO?

A. It forces the decoder to ignore z.

B. It regularises the approximate posterior towards the prior distribution.

C. It replaces the reconstruction term.

D. It makes the model deterministic.

Answer: B — it pulls $q_\phi(z \mid x)$ towards the prior $p(z)$.

The idea. For each image the encoder says "your code is roughly here, give or take this much": a small Gaussian cloud $q_\phi(z \mid x)$. The KL term measures how different each cloud is from the standard cloud $\mathcal{N}(0, I)$ and charges for it. Result: all the images' clouds sit together inside the standard cloud, with no empty gaps between them — so a random draw from $\mathcal{N}(0, I)$ lands somewhere the decoder knows.

Why not the others. A describes a *failure* (posterior collapse, Question 14), not the purpose. C: the ELBO keeps both terms. D: a VAE is probabilistic by design.

In the book. §4.5, §4.6.3.

Q5 Where the mask tokens go in MAE

In a masked autoencoder, why are mask tokens introduced before the decoder rather than before the encoder?

A. Because the encoder should spend computation only on visible image evidence.
B. Because mask tokens are ground-truth labels.
C. Because the decoder cannot process positional embeddings.
D. Because the encoder must always be convolutional and cannot process tokens.

Answer: A — the encoder should only work on what is visible.

The idea. With 75% masking only 49 of 196 patches are visible. A transformer's cost grows with the square of the number of tokens, so the encoder on 49 tokens is roughly 16× cheaper per layer than on 196. The mask tokens ("something is missing here") are only needed when the image is rebuilt, so they are added in the small decoder.

Why not the others. B: mask tokens are learned placeholders, not labels. C: the decoder does use positions — it must know *where* each missing patch goes. D: MAE's encoder is a transformer.

In the book. §5.3.

Q6 A tiny code can still memorise

Why can an autoencoder with a very small latent dimension still memorise a finite dataset?

A. Because the decoder never uses the latent code.
B. Because a high-capacity encoder–decoder can implement a codebook-like mapping.
C. Because low-dimensional latents always increase stochasticity.
D. Because reconstruction loss explicitly penalises generalisation.

Answer: B — a big network can turn even one number into a lookup table.

The idea. Take 1000 training images and a code with a single number. A powerful encoder can send image 1 to $z = 0.001$, image 2 to $z = 0.002$, and so on; a powerful decoder learns "0.001 means this exact picture". The code is tiny, the reconstruction is perfect, and nothing general was learned — a new image has no sensible code.

Why not the others. A: the decoder relies on z entirely. C: a plain autoencoder is deterministic; small does not mean random. D: the loss neither rewards nor penalises generalisation; it simply does not ask for it.

In the book. §4.2 ("why it is not a guarantee").

Q7 Why a lower bound (ELBO)?

Why does the VAE maximise a lower bound instead of $\log p_\theta(x)$ directly?

A. Because the marginal likelihood requires integrating over latent variables, which is generally intractable.
B. Because $\log p_\theta(x)$ is always negative.
C. Because neural networks cannot optimise log functions.
D. Because VAEs do not define a likelihood.

Answer: A — the integral over all codes cannot be computed.

The idea. The decoder tells us $p_\theta(x \mid z)$ for one code z. The probability of the image itself is the average over *every possible* code: $p_\theta(x) = \int p_\theta(x \mid z)\, p(z)\, dz$. In 128 dimensions that integral has no formula, and random sampling almost never hits the few codes that explain x. The ELBO replaces it with something computable that is always $\leq \log p_\theta(x)$.

Why not the others. B: negative numbers optimise fine. C: log-losses are everywhere. D: the decoder *is* a likelihood model.

In the book. §4.5, §4.5.1.

Q8 The reparameterisation trick

What is the main purpose of $z = \mu + \sigma \odot \epsilon$ with $\epsilon \sim \mathcal{N}(0, I)$?

A. It allows gradients to flow through the stochastic sampling operation.
B. It removes the decoder from the model.
C. It forces all reconstructions to be identical.
D. It changes the prior from Gaussian to categorical.

Answer: A — it lets gradients pass through the random draw.

The idea. "Draw z from $\mathcal{N}(\mu, \sigma^2)$" is a black box to backpropagation: how should z change if μ changes? Rewrite it as "draw ϵ from a fixed $\mathcal{N}(0, 1)$, then compute $z = \mu + \sigma\epsilon$". Now the randomness sits in ϵ, which needs no gradient, and z is an ordinary formula in μ and σ: $\partial z/\partial \mu = 1$, $\partial z/\partial \sigma = \epsilon$. *Numbers:* $\mu = 2$, $\sigma = 0.5$, $\epsilon = -1.2$ gives $z = 1.4$; nudging μ to 2.1 moves z to exactly 1.5.

Why not the others. B, C, D describe nothing the trick does.

In the book. §4.6.2.

Q9 Plain autoencoder versus VAE as a generator

What is the key conceptual difference between a deterministic autoencoder and a VAE as a generative model?

A. A deterministic AE always has better samples than a VAE.

B. A VAE imposes a structured latent distribution so new samples can be generated by sampling $z \sim p(z)$.

C. A deterministic AE cannot be trained with gradient descent.

D. A VAE removes the reconstruction objective.

Answer: B — the VAE shapes the code space so that random codes are valid.

The idea. A plain autoencoder's codes can sit anywhere, in odd clumps with big empty gaps; decode a point from a gap and you get noise (Question 16). The VAE's KL term packs the codes into a known shape, $\mathcal{N}(0, I)$, so you know *where* to draw from.

Why not the others. A: plain autoencoders usually generate worse, because they have no map of their code space. C: both train by gradient descent. D: the VAE keeps reconstruction.

In the book. §4.4, §4.6.5.

17.3 Questions 10–20: case studies

The case studies describe what someone *observed*. The trick is to translate the symptom back to one of the six ideas.

Q10 Two models, same loss, different behaviour

Two autoencoders reach the same average reconstruction loss on noisy images. Model A has a very large code and reproduces the random noise accurately. Model B has a constrained code and reconstructs stable object structure while smoothing the noise. Which matches the desired bottleneck behaviour?

A. Model A, because preserving noise proves the latent is complete.

B. Model B, because the bottleneck should keep dataset-explainable structure and suppress noise.

C. Neither, because autoencoders must preserve all pixels exactly.

D. Model A, because higher capacity always improves representation learning.

Answer: B — Model B.

The idea. The same number can hide different errors: A spends its error budget on blurring the *objects* slightly while copying the noise; B spends it on the noise while getting the objects right. Only B's code contains what another image of the same scene would share. This is idea 2 (Figure 17.1, middle).

Why not the others. A, D: copying noise is the "too loose" panel. C: pixel-exact copying is not the goal of representation learning.

Q11 Near-zero training loss, poor validation

You train an autoencoder on a small robotics image set. Training reconstruction loss is near zero, but validation reconstruction and the downstream policy are poor. What is the most principled intervention?

A. Increase encoder and decoder capacity further.
B. Add stronger bottleneck pressure, regularisation, corruption, or data augmentation.
C. Remove the bottleneck entirely.
D. Evaluate only on training reconstruction loss.

Answer: B — squeeze harder: regularise, corrupt, augment.

The idea. "Perfect on training, bad on new data" is the signature of memorisation (idea 3). The fix is to make memorising harder: a smaller code, a penalty, noise on the input (a denoising autoencoder), or more varied data through augmentation — each one makes the lookup-table strategy fail.

Why not the others. A and C make memorising *easier*. D hides the problem instead of fixing it.

In the book. §4.3.

Q12 Great reconstruction, poor linear probe

An autoencoder reconstructs excellently, but its frozen encoder does poorly under a linear probe. What is the best interpretation?

A. Reconstruction quality and representation usefulness are not equivalent.
B. The reconstruction loss must have been implemented incorrectly.
C. Linear probing is invalid for self-supervised models.
D. The decoder should be used as the representation instead of the encoder.

Answer: A — reconstructing well $\neq$ representing well.

The idea. A linear probe asks: "can a single straight cut in code space separate cats from trucks?". Reconstruction asks: "can the decoder redraw every pixel?". A code can be excellent at the second (it stores colours, textures, exact positions) and poor at the first ("cat-ness" is spread out in a tangled way). The two questions reward different information. The VAE notebook in the next chapter lets you measure exactly this.

Why not the others. B is possible in any project but not the best explanation of a well-known effect. C: linear probing is a standard test. D: the encoder is the part that is kept.

In the book. §5.6; linear probing in §8.8.

Q13 A weak bottleneck

A high-capacity autoencoder has a very large code and almost no regularisation. It reconstructs training images extremely accurately, including sensor noise. Which failure mode is most likely?

A. The model cannot reconstruct even simple patterns.
B. The latent representation may preserve nuisance variables and noise.
C. The latent representation must become disentangled.
D. The KL term becomes infinite.

Answer: B — the code keeps nuisance and noise.

The idea. Nothing forces the code to choose, so it keeps everything, including things you do not care about ("nuisance variables": lighting, exact camera position, sensor noise). The right panel of Figure 17.1.

Why not the others. A is the opposite problem (too tight). C: nothing in reconstruction training guarantees disentanglement. D: a plain autoencoder has no KL term.

Q14 KL near zero, decoder still fine

You train a VAE. The KL term collapses close to zero, while the decoder still reconstructs reasonably well. What is the most likely interpretation?

A. The decoder may be ignoring the latent variable, causing posterior collapse.
B. The model has learned a perfectly disentangled latent representation.
C. The prior has disappeared from the objective.
D. The model has become a masked autoencoder.

Answer: A — posterior collapse.

The idea. KL ≈ 0 means $q_\phi(z \mid x) \approx \mathcal{N}(0, I)$ *for every image*: the code no longer depends on the input, so it carries no information about it. If reconstructions still look acceptable, the decoder must be producing them without z — for example a powerful decoder that just outputs a typical, average-looking image. The KL term was "paid off" by giving up the code.

Why not the others. B: collapse makes the code useless, not disentangled. C: the prior is still in the KL term — it is just being matched exactly. D is unrelated.

In the book. §4.7; you can provoke it in the notebook with a large β (§18.2).

Q15 Blurry VAE reconstructions

A VAE with a Gaussian decoder likelihood gives smooth reconstructions that lack sharp detail. Which explanation is most consistent?

A. The Gaussian reconstruction objective encourages averaging when multiple outputs are plausible.

B. The KL term directly sharpens images.

C. The model must be overfitting class labels.

D. The reparameterisation trick prevents the decoder from learning.

Answer: A — squared error rewards the average of the plausible answers.

The idea. A Gaussian decoder means the loss is squared error. Suppose the code does not pin down exactly where a hair strand is: it is equally likely at pixel 10 or pixel 12. Drawing it sharply at pixel 10 costs a lot whenever the truth is 12, and vice versa. Drawing a faint smear over both places costs less on average — so the decoder learns to smear (Figure 17.2). Blur is the optimal answer to an uncertain question under squared error.

Why not the others. B: KL acts on the code, not on pixels. C: no labels are used. D: the trick is what makes training possible.

In the book. §4.6.

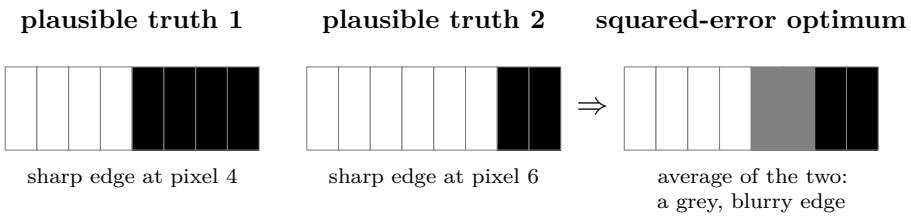


Figure 17.2. Why squared error blurs (Question 15). If two sharp images are equally plausible, the prediction with the lowest expected squared error is their pixel-wise average: grey where they disagree.

How to read it: Look at pixels 4 and 5: black in one truth, white in the other, so the best single guess is 50% grey.

Q16 Random codes into a plain autoencoder

After training a deterministic autoencoder, you decode random Gaussian vectors. The outputs look nonsensical. What is the best explanation?

A. The AE was not trained to make arbitrary latent samples follow a known prior.

B. The decoder can only reconstruct if labels are provided.

C. Reconstruction loss prevents the decoder from using z.

D. The encoder should have been removed during training.

Answer: A — nobody told the codes to fill $\mathcal{N}(0, I)$.

The idea. The decoder only ever saw codes that the encoder produced. Those might all lie, say, in a thin curved sheet far from the origin. A random $\mathcal{N}(0, I)$ vector lands off that sheet, in territory the decoder never trained on. This is exactly what the VAE's KL term fixes (Question 9), and what happens in the notebook when you set $\beta = 0$.

Why not the others. B, C, D are false statements about autoencoders.

Q17 MAE on a compute budget

You design an MAE for high-resolution images with limited compute. Which design best preserves the main MAE advantage?

A. A large encoder on only visible patches and a lightweight decoder.
B. Feed all patches and mask tokens into the encoder.
C. A large decoder, and discard the encoder after pretraining.
D. No positional information, because masking already gives location cues.

Answer: A — big encoder on visible patches, small decoder.

The idea. Put the compute where it is kept: after pretraining the decoder is thrown away and the encoder is used for every downstream task. Running the big encoder on 25% of the tokens is also what makes MAE about three times faster to train than running it on all of them.

Why not the others. B loses the speed-up (Question 5). C invests in the part that is thrown away. D: without positions the decoder cannot know where a missing patch belongs.

In the book. §5.3.

Q18 75% masking beats 15%

One MAE run masks 15% of patches, the other 75%. The 75% run learns better downstream representations. Why is this plausible?

A. Low masking may allow local interpolation, while high masking forces broader contextual inference.
B. High masking makes the decoder irrelevant.
C. High masking gives the encoder more visible patches.
D. Low masking prevents the model from using positional information.

Answer: A — with little hidden, neighbours give the answer away.

The idea. Images are very redundant. If one patch in seven is missing, its neighbours on all sides are visible, and "copy and blend the neighbours" fills it well — no understanding needed. With three patches in four missing, whole parts of the object are gone; to fill them the model must know what the object is ("this is a dog, so there is a leg here"). Figure 17.3. Text is different: every word carries a lot of meaning, which is why BERT masks only 15%.

Why not the others. B: the decoder still does the rebuilding. C is backwards. D: positions are used at every ratio.

In the book. §5.1, §5.5.

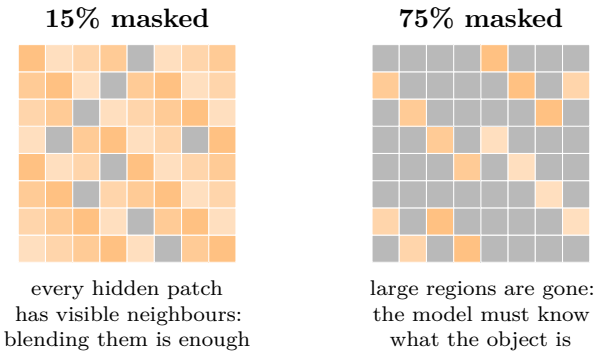


Figure 17.3. Low versus high masking ratio (Question 18). Grey: hidden patches. With 15% hidden, local blending solves the task; with 75% hidden, it does not.

How to read it: Pick any grey square on the left and count its visible neighbours; then try the same on the right.

Q19 MAE versus full pixel reconstruction

You compare reconstructing all pixels through a low-dimensional bottleneck with reconstructing heavily masked patches with a large vision encoder. Why might the second give stronger representations for recognition?

A. It creates a difficult prediction task while keeping the encoder focused on visible semantic evidence.

B. It avoids learning spatial structure.

C. It uses class labels more efficiently.

D. It forces the model to memorise each image exactly.

Answer: A — a hard task, with the encoder looking only at real evidence.

The idea. The two methods make the task hard in different ways. The bottleneck autoencoder sees everything and must *compress*; it tends to keep low-level detail that helps redraw pixels (Question 12). MAE's encoder has plenty of room (a full-size token per visible patch) but must *predict* what it cannot see; that pushes it towards what things are and how parts relate. Masking replaces the bottleneck as the thing that makes the task non-trivial.

Why not the others. B: spatial structure is essential. C: no labels are used. D is the opposite of the goal.

In the book. §4.8, §5.5.

Q20 Why stacked autoencoders were trained layer by layer

An older method trains one shallow autoencoder at a time and then stacks them. What problem was this designed to address?

A. Deep autoencoders were hard to train end-to-end because of vanishing gradients.
B. Shallow layers cannot use nonlinearities.
C. Layer-wise training removes the need for any decoder.
D. Layer-wise training makes the latent variables random like a VAE.

Answer: A — vanishing gradients in deep networks.

The idea. Before ReLU, batch normalisation and good initialisation (before about 2010), gradients shrank at every layer on the way back, so the first layers of a deep network barely learned. The workaround: train a one-layer autoencoder, freeze it, train another one on its codes, and so on; then stack them and fine-tune. Today deep autoencoders train end to end, which is why the method is historical.

Why not the others. B: shallow networks use nonlinearities too. C: every layer still has its own decoder while it is trained. D: stacked autoencoders are deterministic.

In the book. §4.3.

17.4 The quiz on one page

Q	Ans.	The one-line lesson
1	B	An autoencoder's loss is reconstruction error, nothing else.
2	A	Latent size, capacity and regularisation set the bottleneck.
3	C	Keep what explains the dataset; drop noise and redundancy.
4	B	KL pulls each image's code cloud towards $\mathcal{N}(0, I)$.
5	A	Mask tokens join in the decoder so the encoder sees only real patches.
6	B	A big network turns even a one-number code into a lookup table.
7	A	$\log p_\theta(x)$ needs an intractable integral, hence the ELBO.
8	A	$z = \mu + \sigma\epsilon$ moves the randomness out of the gradient's way.
9	B	A VAE gives the code space a known shape, so random codes decode.
10	B	Suppressing noise is the goal, not a failure.
11	B	Memorisation $\Rightarrow$ make copying harder.
12	A	Good reconstruction does not imply linearly useful features.
13	B	A loose bottleneck keeps nuisance and noise.
14	A	KL ≈ 0 with fine reconstructions = decoder ignores z.
15	A	Squared error averages the plausible answers into blur.
16	A	A plain AE never shaped its code space, so random codes fail.
17	A	Spend compute on the encoder you keep.
18	A	Hide enough that blending neighbours is not enough.
19	A	Masking makes the task hard without starving the encoder.
20	A	Layer-wise training was a workaround for vanishing gradients.

Key takeaway — the pattern behind the case studies

Every case study had the shape "the loss looks fine, but ...". Low loss with noise copied (Q10, Q13), low training loss with poor validation (Q11), good reconstruction with a poor probe (Q12), small KL with fine reconstructions (Q14), fine reconstructions with bad samples (Q16). The lesson of the week: *the training loss measures the pretext task, not the thing you want.* Always check the thing you want separately.

CHAPTER 18

Coding Tutorials: A VAE and a Tiny MAE on CIFAR-10

In this chapter

18.1 Coding Tutorial 1: a VAE, cell by cell 777 • Setup, imports and configuration • Data • The model • The loss • Training, curves, pictures and the probe • What the run shows
18.2 Coding Tutorial 1: the play-around questions, solved 789
18.3 Coding Tutorial 2: a tiny MAE, cell by cell 790 • Setup, configuration and data • Patchify and unpatchify • Random masking and the model • Loss, visualisation and training • What the run shows
18.4 Coding Tutorial 2: the play-around questions, solved 803

Chapter roadmap

The material. Two Colab notebooks that accompany the week-3 quiz (Chapter 17). Coding Tutorial 1 trains a small convolutional VAE on CIFAR-10; Coding Tutorial 2 trains a tiny masked autoencoder. Each ends with "play-around" questions that are the quiz's case studies turned into experiments.

How this chapter is laid out. Every code cell of both notebooks is printed in full, in order, exactly as in the `.ipynb` files. After each cell comes what it does, line by line where it matters, and how it connects to the maths of Chapters 4 and 5. Then come the notebook's real outputs, and finally worked solutions to every play-around question.

Where the outputs come from. The VAE figures are the outputs saved in the notebook (50 epochs on an RTX 4090, about 30 seconds). The MAE notebook was shipped without outputs, so it was run for this book with its default settings on a laptop CPU (8 epochs, 2.3 minutes of training). Exact numbers will differ slightly on your machine; the trends will not.

18.1 Coding Tutorial 1: a VAE, cell by cell

What the notebook teaches. (1) How a VAE encodes an image into a mean μ and a log-variance $\log \sigma^2$; (2) why the loss has a reconstruction term and a KL term; (3) how

the reparameterisation trick makes the random draw trainable; (4) how latent size, β (KL pressure) and data-set size change reconstructions and samples. Its case-study map links the experiments to week-3 Questions 11, 12, 14, 15 and 16.

18.1.1 Setup, imports and configuration

```
# Colab setup + GPU check
# In Colab: Runtime -> Change runtime type -> GPU, then run this cell.
# This installs from requirements.txt if it is available; otherwise it installs the small package list directly.
# This notebook uses simple print logging instead of progress-bar renderers.
!if [ -f ../requirements.txt ]; then pip install -q -r ../requirements.txt; elif [ -f requirements.txt ]; then pip install -q -r requirements.txt; else pip install -q torch torchvision numpy matplotlib; fi

import sys, subprocess
try:
    import torch
    print("Python:", sys.version.split()[0])
    print("PyTorch:", torch.__version__)
    print("CUDA available:", torch.cuda.is_available())
    if torch.cuda.is_available():
        print("GPU:", torch.cuda.get_device_name(0))
        try:
            print(subprocess.check_output(["nvidia-smi", "--query-gpu=name,memory.total,memory.free", "--format=csv,noheader"], text=True).strip())
        except Exception:
            pass
    else:
        print("No GPU detected. In Colab, enable GPU via Runtime -> Change runtime type -> GPU.")
except ImportError:
    print("PyTorch is not installed yet. In Colab, uncomment and run the %pip install line above, then restart runtime.")
```

Listing 18.1. VAE cell 1: Colab setup and GPU check.

The `!` line is a shell command (it only runs inside Jupyter or Colab): install the packages. The rest prints the Python and PyTorch versions and whether a GPU is visible. Nothing here affects the model.

```
import os, math, random, time
from pathlib import Path
from dataclasses import dataclass

import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
```

```python
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
from torchvision.utils import make_grid

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", DEVICE)
if DEVICE.type == "cuda":
    print("GPU:", torch.cuda.get_device_name(0))
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True

def default_data_root():
    from pathlib import Path
    # If run from project root, use ./data symlink. If run from notebooks/, use ../data.
    for candidate in [Path("./data"), Path("../data")]:
        if candidate.exists() or candidate.is_symlink():
            return str(candidate)
    # Colab fallback: download CIFAR10 into the runtime workspace.
    return "./data"
```

Listing 18.2. VAE cell 2: imports, seeds and device.

Fixing the three random seeds (Python, NumPy, PyTorch) makes the run repeatable. `DEVICE` is the GPU if there is one. The TF32 switches let recent NVIDIA GPUs multiply faster with slightly lower precision. `default_data_root` looks for an existing `data` folder so CIFAR-10 is not downloaded twice.

```python
# Self-contained config cell: safe to run even if previous cells were skipped or rerun out of order.
from dataclasses import dataclass

def default_data_root():
    from pathlib import Path
    for candidate in [Path("./data"), Path("../data")]:
        if candidate.exists() or candidate.is_symlink():
            return str(candidate)
    return "./data"

@dataclass
class CFG:
    DATA_ROOT: str = default_data_root()
    TRAIN_SUBSET: int = 20_000      # use 50_000 for full CIFAR10 train split
    VAL_SUBSET: int = 5_000
    BATCH_SIZE: int = 256
    EPOCHS: int = 50                 # 5-10 is good for tutorial time
    LR: float = 2e-3
    LATENT_DIM: int = 128
    BETA: float = 0.10              # KL weight; try 0.0, 0.01, 1.0, 5.0
    RECON_LOSS: str = "mse"         # "mse" or "bce"
```

```
    NUM_WORKERS: int = 2

cfg = CFG()
print(cfg)
```

Listing 18.3. VAE cell 3: the configuration — the knobs to play with.

All experiment settings in one place:

- `TRAIN_SUBSET = 20_000`: train on 20,000 of the 50,000 CIFAR-10 training images (faster); `VAL_SUBSET = 5_000` test images for validation.
- `LATENT_DIM = 128`: the size of z, i.e. the width of the bottleneck.
- `BETA = 0.10`: the weight of the KL term. $\beta = 1$ is the true ELBO; smaller β favours sharp reconstructions, larger β a well-shaped code space.
- `RECON_LOSS`: `"mse"` (Gaussian decoder) or `"bce"` (Bernoulli decoder, treats each pixel value as a probability).
- `EPOCHS = 50`, `LR = 2e-3`, batch 256.

18.1.2 Data

```
transform = transforms.ToTensor()  # PIL image -> tensor in [0, 1]

# Download/load CIFAR10. Labels are kept only for the representation probe.
train_full = datasets.CIFAR10(root=cfg.DATA_ROOT, train=True, download=True,
    ↪ transform=transform)
test_full = datasets.CIFAR10(root=cfg.DATA_ROOT, train=False, download=True,
    ↪ transform=transform)

# Fixed random subset keeps the tutorial fast and reproducible.
rng = torch.Generator().manual_seed(SEED)
train_indices = torch.randperm(len(train_full),
    ↪ generator=rng)[:cfg.TRAIN_SUBSET].tolist()
val_indices = torch.randperm(len(test_full), generator=rng)[:cfg.VAL_SUBSET].tolist()

train_ds = Subset(train_full, train_indices)
val_ds = Subset(test_full, val_indices)

pin = DEVICE.type == "cuda"  # faster CPU->GPU transfer when CUDA is used
train_loader = DataLoader(train_ds, batch_size=cfg.BATCH_SIZE, shuffle=True,
                          num_workers=cfg.NUM_WORKERS, pin_memory=pin)
val_loader = DataLoader(val_ds, batch_size=cfg.BATCH_SIZE, shuffle=False,
                        num_workers=cfg.NUM_WORKERS, pin_memory=pin)

classes = train_full.classes
print(f"train examples: {len(train_ds)} | val examples: {len(val_ds)} | classes:
    ↪ {classes}")
```

Listing 18.4. VAE cell 4: load CIFAR-10 and make fixed random subsets.

`ToTensor` turns each 32×32 colour image into a tensor of shape $3 \times 32 \times 32$ with values in $[0, 1]$ — matching the decoder's final sigmoid. A seeded `randperm` picks the same subsets every run. The labels are loaded but, as the comment says, used only by the linear probe at the end: VAE training never sees them.

```
def show_images(x, title=None, nrow=8):
    # Detach from graph, move to CPU, and keep valid image range.
    x = x.detach().cpu().clamp(0, 1)
    grid = make_grid(x, nrow=nrow, padding=2)
    plt.figure(figsize=(nrow, nrow))
    plt.imshow(grid.permute(1, 2, 0))
    plt.axis("off")
    if title:
        plt.title(title)
    plt.show()

x0, y0 = next(iter(train_loader))
show_images(x0[:32], "CIFAR10 examples")
```

Listing 18.5. VAE cell 5: a helper to show a grid of images.

`make_grid` tiles a batch into one picture; `permute(1,2,0)` reorders channels-first (PyTorch) to channels-last (matplotlib).

18.1.3 The model

```
class ConvVAE(nn.Module):
    def __init__(self, latent_dim=128):
        super().__init__()
        self.latent_dim = latent_dim
        # Encoder downsamples 32x32 -> 16x16 -> 8x8 -> 4x4.
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=4, stride=2, padding=1),   # 16x16
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1),  # 8x8
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1), # 4x4
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Flatten(),
        )
        # Encoder outputs parameters of q(z|x), not a single deterministic code.
        self.fc_mu = nn.Linear(128 * 4 * 4, latent_dim)
        self.fc_logvar = nn.Linear(128 * 4 * 4, latent_dim)

        # Decoder maps latent vector back to an image.
        self.fc_dec = nn.Linear(latent_dim, 128 * 4 * 4)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
```

```
27              nn.ReLU(inplace=True),
28              nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding=1),
29              nn.BatchNorm2d(32),
30              nn.ReLU(inplace=True),
31              nn.ConvTranspose2d(32, 3, kernel_size=4, stride=2, padding=1),
32              nn.Sigmoid(),
33          )
34
35      def encode(self, x):
36          h = self.encoder(x)
37          return self.fc_mu(h), self.fc_logvar(h)
38
39      def reparameterize(self, mu, logvar):
40          # z = mu + sigma * epsilon keeps sampling differentiable.
41          std = torch.exp(0.5 * logvar)
42          eps = torch.randn_like(std)
43          return mu + eps * std
44
45      def decode(self, z):
46          h = self.fc_dec(z).view(-1, 128, 4, 4)
47          return self.decoder(h)
48
49      def forward(self, x):
50          # Encode image -> sample latent -> decode reconstruction.
51          mu, logvar = self.encode(x)
52          z = self.reparameterize(mu, logvar)
53          recon = self.decode(z)
54          return recon, mu, logvar
55
56  model = ConvVAE(cfg.LATENT_DIM).to(DEVICE)
57  print(model)
58  print("parameters:", sum(p.numel() for p in model.parameters()) / 1e6, "M")
```

Listing 18.6. VAE cell 6: the convolutional VAE.

Step by step — an image's trip through `ConvVAE`

Step 1. **Encoder** (lines 5–17). Three convolutions with kernel 4, stride 2 halve the size each time: $32 \to 16 \to 8 \to 4$, while the channels grow $3 \to 32 \to 64 \to 128$. Flatten: $128 \times 4 \times 4 = 2048$ numbers.

Step 2. **Two heads** (lines 19–20). Two linear layers map those 2048 numbers to μ (128 numbers) and $\log \sigma^2$ (128 numbers). Together they describe $q_\phi(z \mid x) = \mathcal{N}(\mu, \sigma^2)$: not one code but a small cloud of codes. The network predicts the *log* of the variance so the output can be any real number while $\sigma^2 = e^{\log \sigma^2}$ stays positive.

Step 3. **Reparameterise** (`reparameterize`). $\sigma = \exp(0.5 \log \sigma^2)$, draw $\epsilon \sim \mathcal{N}(0, I)$, return $z = \mu + \epsilon\sigma$ — week-3 Question 8 in three lines.

Step 4. **Decoder** (lines 22–33 and `decode`). A linear layer back to 2048 numbers, reshaped to $128 \times 4 \times 4$, then three transposed convolutions double the size

$4 \to 8 \to 16 \to 32$ and end in 3 channels with a sigmoid, so every output pixel is in $[0, 1]$.

The printed model has 1.12 million parameters — tiny by modern standards, which is why it trains in seconds on a GPU.

18.1.4 The loss

```python
def vae_loss(recon, x, mu, logvar, beta=0.1, recon_loss="mse"):
    batch = x.size(0)
    # Reconstruction term: how well x_hat matches x.
    if recon_loss == "mse":
        recon_term = F.mse_loss(recon, x, reduction="sum") / batch
    elif recon_loss == "bce":
        recon_term = F.binary_cross_entropy(recon, x, reduction="sum") / batch
    else:
        raise ValueError("recon_loss must be 'mse' or 'bce'")

    # KL term: keeps q(z|x) close to N(0, I).
    kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) / batch
    total = recon_term + beta * kl
    return total, recon_term, kl

@torch.no_grad()
def evaluate(model, loader):
    # Validation: same loss, but no gradient updates.
    model.eval()
    totals = {"loss": 0.0, "recon": 0.0, "kl": 0.0, "n": 0}
    for x, _ in loader:
        x = x.to(DEVICE, non_blocking=True)
        recon, mu, logvar = model(x)
        loss, rec, kl = vae_loss(recon, x, mu, logvar, cfg.BETA, cfg.RECON_LOSS)
        bs = x.size(0)
        totals["loss"] += loss.item() * bs
        totals["recon"] += rec.item() * bs
        totals["kl"] += kl.item() * bs
        totals["n"] += bs
    return {k: totals[k] / totals["n"] for k in ["loss", "recon", "kl"]}
```

Listing 18.7. VAE cell 7: the negative ELBO, and a validation loop.

This is the ELBO of §4.5, turned into code:

$$\text{loss} = \underbrace{\textstyle\sum_{\text{pixels}} (x - \hat{x})^2}_{\texttt{recon_term}} + \beta \cdot \underbrace{\left(-\tfrac{1}{2} \textstyle\sum_{j=1}^{128} \left(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2\right)\right)}_{\texttt{kl}},$$

each averaged over the batch (`/ batch`). Three details:

- `reduction="sum"` adds the squared error over all $3 \times 32 \times 32 = 3072$ values of an image. Summing (not averaging) keeps the reconstruction and KL terms on comparable scales, as in the ELBO. The $\frac{1}{2}$ of the Gaussian log-likelihood is dropped; it only rescales.
- The KL line is the closed form of $\mathrm{KL}(\mathcal{N}(\mu, \sigma^2) \,\|\, \mathcal{N}(0, 1))$ per dimension (§4.6.3). Check one dimension: $\mu = 0$, $\sigma = 1$ gives $-\frac{1}{2}(1 + 0 - 0 - 1) = 0$ — no cost for matching the prior. $\mu = 2$, $\sigma = 1$ gives $-\frac{1}{2}(1 + 0 - 4 - 1) = 2$.
- `evaluate` runs the same loss under `torch.no_grad()` on the validation images and averages the three numbers.

18.1.5 Training, curves, pictures and the probe

```
optimizer = torch.optim.AdamW(model.parameters(), lr=cfg.LR, weight_decay=1e-4)
history = []
start = time.time()

for epoch in range(1, cfg.EPOCHS + 1):
    model.train()
    running = {"loss": 0.0, "recon": 0.0, "kl": 0.0, "n": 0}
    print(f"Starting epoch {epoch}/{cfg.EPOCHS}...")
    for step, (x, _) in enumerate(train_loader, start=1):
        x = x.to(DEVICE, non_blocking=True)
        optimizer.zero_grad(set_to_none=True)  # clear old gradients
        recon, mu, logvar = model(x)           # forward pass
        loss, rec, kl = vae_loss(recon, x, mu, logvar, cfg.BETA, cfg.RECON_LOSS)
        loss.backward()                        # compute gradients
        optimizer.step()                       # update parameters

        bs = x.size(0)
        running["loss"] += loss.item() * bs
        running["recon"] += rec.item() * bs
        running["kl"] += kl.item() * bs
        running["n"] += bs
        if step == 1 or step == len(train_loader) or step % 25 == 0:
            print(f"  step {step:03d}/{len(train_loader)} | loss {loss.item():.2f} |
    ↪ recon {rec.item():.2f} | KL {kl.item():.2f}")

    train_metrics = {k: running[k] / running["n"] for k in ["loss", "recon", "kl"]}
    val_metrics = evaluate(model, val_loader)
    history.append({"epoch": epoch, **{f"train_{k}": v for k, v in
    ↪ train_metrics.items()},
                    **{f"val_{k}": v for k, v in val_metrics.items()}})
    print(f"Epoch {epoch:02d} | train loss {train_metrics['loss']:.2f} "
          f"(rec {train_metrics['recon']:.2f}, KL {train_metrics['kl']:.2f}) | "
          f"val loss {val_metrics['loss']:.2f} (rec {val_metrics['recon']:.2f}, KL
    ↪ {val_metrics['kl']:.2f})")

print(f"Training time: {(time.time() - start)/60:.1f} minutes")
```

Listing 18.8. VAE cell 8: the training loop.

The standard four-line PyTorch step: clear old gradients, forward pass, loss, `backward`, optimiser step (AdamW). The rest bookkeeps running sums, prints every 25 steps, and stores one line of train and validation numbers per epoch in `history`.

```
# Plot training curves
if history:
    epochs = [h["epoch"] for h in history]
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 3, 1)
    plt.plot(epochs, [h["train_loss"] for h in history], label="train")
    plt.plot(epochs, [h["val_loss"] for h in history], label="val")
    plt.title("total loss"); plt.xlabel("epoch"); plt.legend()
    plt.subplot(1, 3, 2)
    plt.plot(epochs, [h["train_recon"] for h in history], label="train")
    plt.plot(epochs, [h["val_recon"] for h in history], label="val")
    plt.title("reconstruction"); plt.xlabel("epoch"); plt.legend()
    plt.subplot(1, 3, 3)
    plt.plot(epochs, [h["train_kl"] for h in history], label="train")
    plt.plot(epochs, [h["val_kl"] for h in history], label="val")
    plt.title("KL"); plt.xlabel("epoch"); plt.legend()
    plt.tight_layout(); plt.show()
```

Listing 18.9. VAE cell 9: plot the training curves.

```
@torch.no_grad()
def show_reconstructions(model, loader, n=16):
    model.eval()
    x, _ = next(iter(loader))
    x = x[:n].to(DEVICE)
    recon, mu, logvar = model(x)
    stacked = torch.cat([x, recon], dim=0)
    show_images(stacked, "top: original | bottom: reconstruction", nrow=n)

@torch.no_grad()
def show_prior_samples(model, n=32):
    model.eval()
    # Sample from the VAE prior and decode.
    z = torch.randn(n, cfg.LATENT_DIM, device=DEVICE)
    samples = model.decode(z)
    show_images(samples, "samples from z ~ N(0, I)", nrow=8)

show_reconstructions(model, val_loader, n=16)
show_prior_samples(model, n=32)
```

Listing 18.10. VAE cell 10: reconstructions and samples from the prior.

Two different questions. `show_reconstructions` encodes real validation images and decodes them (*can it copy?*). `show_prior_samples` ignores the encoder entirely: it draws $z \sim \mathcal{N}(0, I)$ and decodes (*can it generate?*).

```
@torch.no_grad()
def interpolate_two_images(model, loader, steps=10):
```

```
    model.eval()
    x, _ = next(iter(loader))
    x = x[:2].to(DEVICE)
    mu, logvar = model.encode(x)
    # Interpolate between two latent means.
    z0, z1 = mu[0], mu[1]
    alphas = torch.linspace(0, 1, steps, device=DEVICE).unsqueeze(1)
    z = (1 - alphas) * z0 + alphas * z1
    imgs = model.decode(z)
    show_images(imgs, "latent interpolation", nrow=steps)

interpolate_two_images(model, val_loader, steps=10)
```

Listing 18.11. VAE cell 11: walk in a straight line between two codes.

Encode two images, take their means z_0, z_1, and decode the ten points $(1-a)z_0 + az_1$ for a from 0 to 1. A smooth fade means the code space between the two images is filled with sensible codes.

```
RUN_LINEAR_PROBE = False

class LinearProbe(nn.Module):
    def __init__(self, dim, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(dim, num_classes)
    def forward(self, x):
        return self.fc(x)

@torch.no_grad()
def collect_latents(model, loader, max_batches=None):
    # Freeze VAE; collect encoder means as image features.
    model.eval()
    zs, ys = [], []
    for i, (x, y) in enumerate(loader):
        if max_batches is not None and i >= max_batches:
            break
        x = x.to(DEVICE)
        mu, _ = model.encode(x)
        zs.append(mu.cpu())
        ys.append(y)
    return torch.cat(zs), torch.cat(ys)

if RUN_LINEAR_PROBE:
    z_train, y_train = collect_latents(model, train_loader, max_batches=40)
    z_val, y_val = collect_latents(model, val_loader, max_batches=20)
    probe = LinearProbe(cfg.LATENT_DIM).to(DEVICE)
    # Only the probe is trained; VAE encoder stays frozen.
    opt = torch.optim.AdamW(probe.parameters(), lr=1e-2)
    for ep in range(1, 11):
        probe.train()
        perm = torch.randperm(len(z_train))
        for start_idx in range(0, len(perm), 512):
            idx = perm[start_idx:start_idx+512]
            xb, yb = z_train[idx].to(DEVICE), y_train[idx].to(DEVICE)
```

```
            opt.zero_grad(set_to_none=True)
            loss = F.cross_entropy(probe(xb), yb)
            loss.backward(); opt.step()
        probe.eval()
        with torch.no_grad():
            pred = probe(z_val.to(DEVICE)).argmax(1).cpu()
            acc = (pred == y_val).float().mean().item()
        print(f"probe epoch {ep:02d} | val acc {acc*100:.1f}%")
```

Listing 18.12. VAE cell 12: a linear probe on the frozen mean $\mu(x)$.

Switch `RUN_LINEAR_PROBE` to `True` to run it. `collect_latents` encodes 40 training batches (about 10,000 images) and 20 validation batches and keeps only μ; the VAE itself is frozen. A single linear layer $128 \to 10$ is then trained for ten epochs with cross-entropy on the CIFAR-10 labels, and validation accuracy is printed. It answers week-3 Question 12: are the codes *linearly* useful for recognising the class?

18.1.6 What the run shows

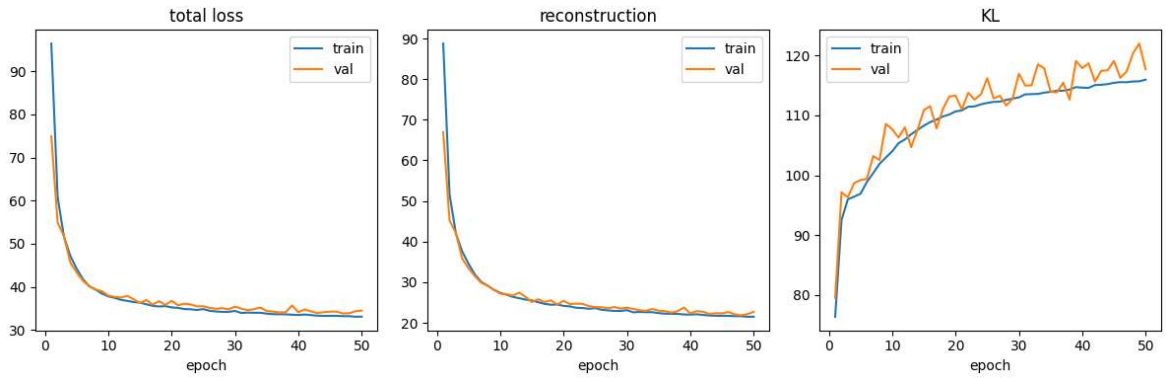


Figure 18.1. VAE training curves over 50 epochs ($\beta = 0.1$, latent 128; saved notebook output). Left: total loss. Middle: reconstruction term. Right: KL term.

How to read it: The middle and right panels move in opposite directions: the model buys better reconstructions by putting more information into z, which the KL term charges for.

The curves (Figure 18.1). After 50 epochs the last line printed is: train loss 33.12 (reconstruction 21.52, KL 115.94), validation loss 34.50 (reconstruction 22.73, KL 117.68).

- *What* 21.5 *means.* It is a sum over 3072 pixel values, so the average squared error per value is $21.5/3072 \approx 0.007$, a typical error of $\sqrt{0.007} \approx 0.084$, about 8% of the brightness range.
- *The KL rises* from about 77 to 116 nats while reconstruction falls: the encoder learns to put more and more information about each image into z. With $\beta = 0.1$, each nat of KL costs only a tenth of a unit of squared error, so this trade is cheap.

- *Train and validation stay close* (22.7 versus 21.5): no sign of memorisation with 20,000 images and this bottleneck (compare week-3 Question 11).

Figure 18.2. Top: sixteen validation images. Bottom: the VAE's reconstructions (saved notebook output).

How to read it: Colours and rough layout survive; edges, faces and legs do not.

Reconstructions (Figure 18.2). Recognisably the same scenes — the white dog, the boats on blue water — but blurred. This is week-3 Question 15 in real pixels: under squared error, detail the code cannot pin down is replaced by its average.

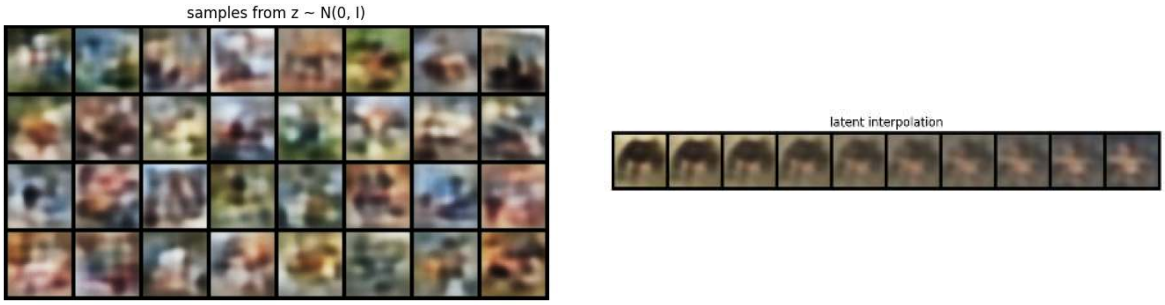


Figure 18.3. Left: 32 images decoded from random $z \sim \mathcal{N}(0, I)$. Right: decoding ten evenly spaced points between the codes of two validation images (saved notebook outputs).

How to read it: Left: plausible colours and "scene-like" blobs, but no objects. Right: a smooth fade, with no broken frames in between.

Samples and interpolation (Figure 18.3). Random codes give images with CIFAR-like colours and composition (sky on top, ground below, an object-sized blob in the middle), never a clear object. That is what a 1-million-parameter VAE on 20,000 images can do; the point is that *every* random code decodes to something image-like, which a plain autoencoder does not manage (play-around D). The interpolation fades smoothly, so the code space is continuous between data points.

The linear probe. The saved notebook has the probe switched off. Re-running the notebook for this book with `RUN_LINEAR_PROBE = True` (same settings, laptop CPU; the final validation loss, 34.43, matches the saved run's 34.50) gives a probe accuracy that rises

from 35.4% after one probe epoch to 38.7% after ten — almost four times chance. The table at the end of the chapter (Table 18.1) puts it next to the MAE's.

18.2 Coding Tutorial 1: the play-around questions, solved

The notebook asks you to change *one* variable at a time, retrain, and record what happens. Here is what you should see and why.

A. Bottleneck and memorisation (`TRAIN_SUBSET = 2_000`, `LATENT_DIM = 256`, 15 epochs). *What happens:* the training reconstruction keeps falling while the validation reconstruction stops improving early and a gap opens between them. *Why:* ten times fewer images and a twice-wider code make it easier to store individual images than to learn what they share — the "too loose" panel of Figure 17.1. *Answer to "learning structure or memorising?":* a growing train–validation gap is the evidence of memorisation (week-3 Question 11). With $\beta > 0$ the KL term resists this, so the gap is smaller than it would be for a plain autoencoder.

B. KL pressure and posterior collapse ($\beta \in \{0, 0.01, 0.1, 1, 5\}$).

1. *KL as β grows:* it falls. At $\beta = 0$ nothing stops it, so it grows large (the codes carry a lot and spread far from $\mathcal{N}(0, I)$). At $\beta = 5$ it is pushed towards zero.
2. *When reconstructions get worse:* gradually from $\beta = 1$ and clearly at $\beta = 5$: images turn into colour blobs, because the codes are no longer allowed to carry enough information. If the KL goes almost to zero while the decoder still outputs "average-looking" images, that is posterior collapse (week-3 Question 14): the decoder has learned to ignore z.
3. *Prior samples:* at $\beta = 0$ they are poor (codes do not fill $\mathcal{N}(0, I)$); they improve as β grows, then become bland and all alike when β is very large. The best samples come from a middle β: the recurring trade-off between a well-shaped code space and informative codes.

C. Blurry reconstructions (`RECON_LOSS = "mse"` versus `"bce"`). Binary cross-entropy punishes confident mistakes near 0 and 1 more strongly, so dark and bright regions often come out with a little more contrast; both remain blurry, because both still average over what the code does not determine. The loss values are on different scales, so compare the *pictures*, not the numbers. *Are sharper reconstructions better representations?* Not necessarily (week-3 Question 12): sharpness is about pixels, a linear probe is about classes. Measure both.

D. Plain autoencoder versus VAE sampling (`BETA = 0.0`). With $\beta = 0$ the KL term is gone, the model is effectively a plain autoencoder (plus a little noise from the reparameterisation), and reconstructions get *sharper* (the codes can carry more). But decoding random $z \sim \mathcal{N}(0, I)$ gives noise-like or garbled images. The reason (week-3 Question 16):

nothing made the codes fill the standard Gaussian, so random points fall where the decoder never trained.

E. Representation check (probe on two models with similar reconstructions). Typically the model with the larger β or smaller latent (more compression) has similar or slightly worse reconstructions yet a probe accuracy that differs by several points — in either direction, depending on the data. What it shows: reconstruction quality does not rank representations. Only a downstream test does.

18.3 Coding Tutorial 2: a tiny MAE, cell by cell

What the notebook teaches. The whole MAE recipe in miniature: *patchify* → *mask most patches* → *encode only the visible ones* → *decode the full grid* → *reconstruct the masked patches.* It links to week-3 Questions 12, 17, 18, 19 and 20. The model is a very small vision transformer (four encoder layers of width 128, two decoder layers of width 96) on 32×32 CIFAR images cut into 4×4 patches.

18.3.1 Setup, configuration and data

Cells 1, 2, 4 and 5 are the same as in the VAE notebook (setup, imports, data subsets, image grid); they are printed again so that the notebook is complete here, but only cell 3 needs reading.

```
# Colab setup + GPU check
# In Colab: Runtime -> Change runtime type -> GPU, then run this cell.
# This installs from requirements.txt if it is available; otherwise it installs the small package list directly.
# This notebook uses simple print logging instead of progress-bar renderers.
!if [ -f ../requirements.txt ]; then pip install -q -r ../requirements.txt; elif [ -f requirements.txt ]; then pip install -q -r requirements.txt; else pip install -q torch torchvision numpy matplotlib; fi

import sys, subprocess
try:
    import torch
    print("Python:", sys.version.split()[0])
    print("PyTorch:", torch.__version__)
    print("CUDA available:", torch.cuda.is_available())
    if torch.cuda.is_available():
        print("GPU:", torch.cuda.get_device_name(0))
        try:
            print(subprocess.check_output(["nvidia-smi", "--query-gpu=name,memory.total,memory.free", "--format=csv,noheader"], text=True).strip())
        except Exception:
            pass
    else:
```

```
        print("No GPU detected. In Colab, enable GPU via Runtime -> Change runtime type
    ↪ -> GPU.")
except ImportError:
    print("PyTorch is not installed yet. In Colab, uncomment and run the %pip install
    ↪ line above, then restart runtime.")
```

Listing 18.13. MAE cell 1: Colab setup (as VAE cell 1).

```
import os, math, random, time
from dataclasses import dataclass

import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
from torchvision.utils import make_grid

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", DEVICE)
if DEVICE.type == "cuda":
    print("GPU:", torch.cuda.get_device_name(0))
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True

def default_data_root():
    from pathlib import Path
    # If run from project root, use ./data symlink. If run from notebooks/, use ../data.
    for candidate in [Path("./data"), Path("../data")]:
        if candidate.exists() or candidate.is_symlink():
            return str(candidate)
    # Colab fallback: download CIFAR10 into the runtime workspace.
    return "./data"
```

Listing 18.14. MAE cell 2: imports, seeds, device (as VAE cell 2).

```
# Self-contained config cell: safe to run even if previous cells were skipped or rerun
    ↪ out of order.
from dataclasses import dataclass

def default_data_root():
    from pathlib import Path
    for candidate in [Path("./data"), Path("../data")]:
        if candidate.exists() or candidate.is_symlink():
            return str(candidate)
```

```
    return "./data"

@dataclass
class CFG:
    DATA_ROOT: str = default_data_root()
    TRAIN_SUBSET: int = 20_000
    VAL_SUBSET: int = 5_000
    BATCH_SIZE: int = 256
    EPOCHS: int = 8
    LR: float = 1.5e-3
    PATCH_SIZE: int = 4
    MASK_RATIO: float = 0.75
    EMBED_DIM: int = 128
    DECODER_DIM: int = 96
    ENCODER_DEPTH: int = 4
    DECODER_DEPTH: int = 2
    NUM_HEADS: int = 4
    NUM_WORKERS: int = 2
    LOSS_ON_MASK_ONLY: bool = True

cfg = CFG()
print(cfg)
```

Listing 18.15. MAE cell 3: the configuration.

The new knobs: `PATCH_SIZE = 4` ($32/4 = 8$, so $8 \times 8 = 64$ patches, each $4 \times 4 \times 3 = 48$ numbers); `MASK_RATIO = 0.75` (48 hidden, 16 visible); `EMBED_DIM = 128` and `ENCODER_DEPTH = 4` for the encoder; a smaller decoder (`DECODER_DIM = 96`, `DECODER_DEPTH = 2`); `LOSS_ON_MASK_ONLY = True`.

```
transform = transforms.ToTensor()
train_full = datasets.CIFAR10(root=cfg.DATA_ROOT, train=True, download=True,
    ↪ transform=transform)
test_full = datasets.CIFAR10(root=cfg.DATA_ROOT, train=False, download=True,
    ↪ transform=transform)

# Fixed random subset keeps the tutorial fast and reproducible.
rng = torch.Generator().manual_seed(SEED)
train_indices = torch.randperm(len(train_full),
    ↪ generator=rng)[:cfg.TRAIN_SUBSET].tolist()
val_indices = torch.randperm(len(test_full), generator=rng)[:cfg.VAL_SUBSET].tolist()
train_ds = Subset(train_full, train_indices)
val_ds = Subset(test_full, val_indices)

pin = DEVICE.type == "cuda"  # faster CPU->GPU transfer when CUDA is used
train_loader = DataLoader(train_ds, batch_size=cfg.BATCH_SIZE, shuffle=True,
                          num_workers=cfg.NUM_WORKERS, pin_memory=pin)
val_loader = DataLoader(val_ds, batch_size=cfg.BATCH_SIZE, shuffle=False,
                        num_workers=cfg.NUM_WORKERS, pin_memory=pin)

print(f"train examples: {len(train_ds)} | val examples: {len(val_ds)}")
```

Listing 18.16. MAE cell 4: data (as VAE cell 4).

```python
def show_images(x, title=None, nrow=8):
    # Detach from graph, move to CPU, and keep valid image range.
    x = x.detach().cpu().clamp(0, 1)
    grid = make_grid(x, nrow=nrow, padding=2)
    plt.figure(figsize=(nrow, nrow))
    plt.imshow(grid.permute(1, 2, 0))
    plt.axis("off")
    if title:
        plt.title(title)
    plt.show()

x0, y0 = next(iter(train_loader))
show_images(x0[:32], "CIFAR10 examples")
```

Listing 18.17. MAE cell 5: image grid helper.

18.3.2 Patchify and unpatchify

```python
def patchify(imgs, patch_size=4):
    # Split each image into non-overlapping flattened patches.
    # imgs: (B, 3, H, W) -> patches: (B, N, patch_dim)
    B, C, H, W = imgs.shape
    p = patch_size
    assert H == W and H % p == 0
    h = w = H // p
    x = imgs.reshape(B, C, h, p, w, p)
    x = x.permute(0, 2, 4, 3, 5, 1).reshape(B, h * w, p * p * C)
    return x

def unpatchify(patches, patch_size=4, img_size=32, channels=3):
    # Inverse of patchify: flattened patches -> image grid.
    # patches: (B, N, patch_dim) -> imgs: (B, 3, H, W)
    B, N, D = patches.shape
    p = patch_size
    h = w = img_size // p
    assert N == h * w
    x = patches.reshape(B, h, w, p, p, channels)
    x = x.permute(0, 5, 1, 3, 2, 4).reshape(B, channels, img_size, img_size)
    return x

# sanity check
patches = patchify(x0[:4], cfg.PATCH_SIZE)
reconstructed = unpatchify(patches, cfg.PATCH_SIZE)
print("patches:", patches.shape, "max reconstruction error:", (x0[:4] -
    ↪ reconstructed).abs().max().item())
```

Listing 18.18. MAE cell 6: cut images into patches and glue them back.

Step by step — the shapes in `patchify`, for one batch of B images

Step 1. Start: $(B, 3, 32, 32)$.

Step 2. `reshape(B, C, h, p, w, p)` with $h = w = 8$, $p = 4$: $(B, 3, 8, 4, 8, 4)$ — each 32 is split into "which patch row (8)" × "which row inside the patch (4)".

Step 3. `permute(0, 2, 4, 3, 5, 1)`: $(B, 8, 8, 4, 4, 3)$ — patch row, patch column first; then the 4×4 pixels and 3 colours of that patch.

Step 4. `reshape(B, 64, 48)`: 64 patches, each flattened to 48 numbers. These are the "words" of the transformer.

`unpatchify` undoes the same steps in reverse. The sanity check prints `patches: torch.Size([4, 64, 48]) max reconstruction error: 0.0` — the round trip is exact.

18.3.3 Random masking and the model

```python
def random_masking(x, mask_ratio):
    # Randomly keep a subset of tokens.
    #
    # x: (B, N, D)
    # returns:
    #   x_keep: visible tokens
    #   mask: (B, N), 0 = visible, 1 = masked, in original patch order
    #   ids_restore: indices that restore shuffled order to original order
    #   ids_keep: visible patch indices
    B, N, D = x.shape
    len_keep = int(N * (1 - mask_ratio))  # number of visible patches
    # Random noise decides which patches are kept.
    noise = torch.rand(B, N, device=x.device)
    ids_shuffle = torch.argsort(noise, dim=1)
    ids_restore = torch.argsort(ids_shuffle, dim=1)  # used to unshuffle later
    ids_keep = ids_shuffle[:, :len_keep]
    # Keep only visible tokens for the encoder.
    x_keep = torch.gather(x, dim=1, index=ids_keep.unsqueeze(-1).expand(-1, -1, D))

    # mask: 0 = visible, 1 = hidden.
    mask = torch.ones(B, N, device=x.device)
    mask[:, :len_keep] = 0
    mask = torch.gather(mask, dim=1, index=ids_restore)
    return x_keep, mask, ids_restore, ids_keep

class TinyMAE(nn.Module):
    def __init__(self, img_size=32, patch_size=4, embed_dim=128, decoder_dim=96,
                 encoder_depth=4, decoder_depth=2, num_heads=4, in_chans=3):
        super().__init__()
        self.img_size = img_size
        self.patch_size = patch_size
        # For CIFAR10 and patch_size=4: 8x8 = 64 patches.
        self.num_patches = (img_size // patch_size) ** 2
        self.patch_dim = in_chans * patch_size * patch_size

        # Linear patch embedding plays the role of ViT patch projection.
```

```
        self.patch_embed = nn.Linear(self.patch_dim, embed_dim)
        self.pos_embed = nn.Parameter(torch.zeros(1, self.num_patches, embed_dim))

        enc_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim, nhead=num_heads, dim_feedforward=embed_dim * 4,
            dropout=0.0, activation="gelu", batch_first=True, norm_first=True)
        self.encoder = nn.TransformerEncoder(enc_layer, num_layers=encoder_depth)
        self.encoder_norm = nn.LayerNorm(embed_dim)

        # Lightweight decoder receives visible features plus mask tokens.
        self.decoder_embed = nn.Linear(embed_dim, decoder_dim)
        self.mask_token = nn.Parameter(torch.zeros(1, 1, decoder_dim))
        self.decoder_pos_embed = nn.Parameter(torch.zeros(1, self.num_patches,
    ↪ decoder_dim))
        dec_layer = nn.TransformerEncoderLayer(
            d_model=decoder_dim, nhead=num_heads, dim_feedforward=decoder_dim * 4,
            dropout=0.0, activation="gelu", batch_first=True, norm_first=True)
        self.decoder = nn.TransformerEncoder(dec_layer, num_layers=decoder_depth)
        self.decoder_norm = nn.LayerNorm(decoder_dim)
        self.decoder_pred = nn.Linear(decoder_dim, self.patch_dim)

        self.initialize_weights()

    def initialize_weights(self):
        # Small initialization keeps early predictions stable.
        nn.init.trunc_normal_(self.pos_embed, std=0.02)
        nn.init.trunc_normal_(self.decoder_pos_embed, std=0.02)
        nn.init.normal_(self.mask_token, std=0.02)
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)
            elif isinstance(m, nn.LayerNorm):
                nn.init.ones_(m.weight)
                nn.init.zeros_(m.bias)

    def forward_encoder(self, imgs, mask_ratio):
        # Convert image to patch tokens and add position information.
        patches = patchify(imgs, self.patch_size)
        x = self.patch_embed(patches) + self.pos_embed
        # Encoder sees only visible patches.
        x_keep, mask, ids_restore, ids_keep = random_masking(x, mask_ratio)
        # Transformer mixes information among visible tokens.
        x_encoded = self.encoder_norm(self.encoder(x_keep))
        return x_encoded, mask, ids_restore

    def forward_decoder(self, x_encoded, ids_restore):
        # Project encoder features to decoder dimension.
        x = self.decoder_embed(x_encoded)
        B, L, D = x.shape
        N = ids_restore.shape[1]
        # Add learned placeholders for missing patches.
        mask_tokens = self.mask_token.repeat(B, N - L, 1)
        x_ = torch.cat([x, mask_tokens], dim=1)
        # Restore original patch order before decoding.
```

```
92          x_ = torch.gather(x_, dim=1, index=ids_restore.unsqueeze(-1).expand(-1, -1, D))
93          x_ = x_ + self.decoder_pos_embed
94          # Decoder predicts pixel values for every patch.
95          x_ = self.decoder_norm(self.decoder(x_))
96          pred = self.decoder_pred(x_)
97          return pred
98
99      def forward(self, imgs, mask_ratio=0.75):
100         # Full MAE pass: mask -> encode visible -> decode full patch grid.
101         x_encoded, mask, ids_restore = self.forward_encoder(imgs, mask_ratio)
102         pred = self.forward_decoder(x_encoded, ids_restore)
103         target = patchify(imgs, self.patch_size)  # ground-truth patch pixels
104         return pred, target, mask
105
106     @torch.no_grad()
107     def encode_full(self, imgs):
108         # Use the encoder without masking for representation probing.
109         patches = patchify(imgs, self.patch_size)
110         x = self.patch_embed(patches) + self.pos_embed
111         x = self.encoder_norm(self.encoder(x))
112         return x.mean(dim=1)  # mean-pool patch tokens into one image feature
113
114 model = TinyMAE(
115     patch_size=cfg.PATCH_SIZE,
116     embed_dim=cfg.EMBED_DIM,
117     decoder_dim=cfg.DECODER_DIM,
118     encoder_depth=cfg.ENCODER_DEPTH,
119     decoder_depth=cfg.DECODER_DEPTH,
120     num_heads=cfg.NUM_HEADS,
121 ).to(DEVICE)
122 print(model)
123 print("parameters:", sum(p.numel() for p in model.parameters()) / 1e6, "M")
```

Listing 18.19. MAE cell 7: random masking and the TinyMAE model.

`random_masking`: the argsort trick (lines 1–24). To hide a random 75% of patches for every image in a batch, in parallel, with no loops:

1. draw one random number per patch (`noise`);
2. sort the patches by it (`ids_shuffle`): a random order;
3. keep the first `len_keep` of that order (`ids_keep`);
4. remember how to undo the shuffle (`ids_restore`, the argsort of the argsort);
5. build the mask in shuffled order (first `len_keep` are 0 = visible, rest 1 = hidden) and unshuffle it.

Worked Example — the argsort trick with 4 patches and 75% masking

`len_keep` $= \text{int}(4 \times 0.25) = 1$. Suppose `noise` $= [0.7, 0.1, 0.9, 0.4]$. `ids_shuffle` $=$ argsort $= [1, 3, 0, 2]$ (patch 1 has the smallest number). `ids_keep` $= [1]$: only patch 1 is visible. `ids_restore` $=$ argsort$([1, 3, 0, 2]) = [2, 0, 3, 1]$. Mask in shuffled order: $[0, 1, 1, 1]$.

Unshuffled with `gather(..., ids_restore)`: position 0 takes element 2 (1), position 1 element 0 (0), position 2 element 3 (1), position 3 element 1 (1): mask $= [1, 0, 1, 1]$ — patch 1 visible, the others hidden, in the original patch order. Correct.

`TinyMAE.__init__` (lines 26–57). A linear `patch_embed` ($48 \to 128$) plays the role of ViT's patch projection; `pos_embed` is a learned position vector for each of the 64 slots. The encoder is four standard transformer layers (`norm_first=True` is the pre-norm variant ViT uses). The decoder has its own width (96), its own position embeddings, the learned `mask_token` (one vector of 96 numbers, shared by all hidden patches), two transformer layers, and `decoder_pred` mapping each token back to 48 pixel values. `initialize_weights` uses small random values so early predictions are stable.

`forward_encoder` (lines 73–81). Patchify, embed, add positions *before* masking (so each visible token remembers where it came from), keep 16 of 64 tokens, run the encoder on those 16 only. This is week-3 Question 5 in code: the encoder never sees a mask token.

`forward_decoder` (lines 83–97). Project the 16 encoded tokens to width 96, append 48 copies of the mask token, put all 64 back in their original order with `ids_restore`, add the decoder's position embeddings (now each mask token knows *which* hole it stands for), run the two decoder layers and predict 48 pixel values for every slot.

`forward` and `encode_full` (lines 99–112). `forward` returns prediction, target (the true patches) and mask. `encode_full` is used only for the probe: it runs the encoder on *all* 64 patches without masking and averages them into one 128-number image feature. The model has 1.05 million parameters.

18.3.4 Loss, visualisation and training

```
def mae_loss(pred, target, mask, mask_only=True):
    # MSE per patch, averaged over pixel values inside each patch.
    loss_per_patch = ((pred - target) ** 2).mean(dim=-1)  # (B, N)
    if mask_only:
        # Standard MAE: train only on hidden patches.
        return (loss_per_patch * mask).sum() / mask.sum().clamp(min=1)
    return loss_per_patch.mean()

@torch.no_grad()
def make_mae_visuals(model, imgs, mask_ratio=0.75):
    model.eval()
    imgs = imgs.to(DEVICE)
    pred, target, mask = model(imgs, mask_ratio=mask_ratio)
    mask_expanded = mask.unsqueeze(-1)  # broadcast mask over patch pixels

```

```
    # Masked input: replace hidden patches by gray.
    gray = torch.ones_like(target) * 0.5
    visible_patches = target * (1 - mask_expanded) + gray * mask_expanded
    visible_img = unpatchify(visible_patches, cfg.PATCH_SIZE)

    # Raw decoder prediction for all patches.
    pred_img = unpatchify(pred, cfg.PATCH_SIZE)

    # Final visualization: original visible patches + predicted hidden patches.
    combined_patches = target * (1 - mask_expanded) + pred * mask_expanded
    combined_img = unpatchify(combined_patches, cfg.PATCH_SIZE)
    return imgs, visible_img, pred_img, combined_img, mask

@torch.no_grad()
def show_mae_reconstructions(model, loader, n=8, mask_ratio=None):
    if mask_ratio is None:
        mask_ratio = cfg.MASK_RATIO
    x, _ = next(iter(loader))
    x = x[:n]
    original, visible, pred, combined, mask = make_mae_visuals(model, x, mask_ratio)
    show_images(torch.cat([original, visible, combined, pred], dim=0),
                "row 1: original | row 2: visible input | row 3: combined reconstruction
    ↪ | row 4: raw prediction",
                nrow=n)
```

Listing 18.20. MAE cell 8: the masked loss and the four-row picture.

`mae_loss` first computes the mean squared error *per patch* ($B \times 64$ numbers), then multiplies by the mask, so only hidden patches count, and divides by the number of hidden patches:

$$\mathcal{L} = \frac{\sum_i m_i \, \|\hat{x}_i - x_i\|^2 / 48}{\sum_i m_i}, \qquad m_i = 1 \text{ if patch } i \text{ is hidden.}$$

`make_mae_visuals` builds four rows: the original; the visible input (hidden patches grey); the *combined* image (true visible patches plus predicted hidden ones — what MAE papers show); and the *raw* prediction for all 64 patches.

```
optimizer = torch.optim.AdamW(model.parameters(), lr=cfg.LR, weight_decay=0.05)
history = []
start = time.time()

@torch.no_grad()
def evaluate(model, loader):
    # Validation pass without gradient updates.
    model.eval()
    total, n = 0.0, 0
    for x, _ in loader:
        x = x.to(DEVICE, non_blocking=True)
        pred, target, mask = model(x, mask_ratio=cfg.MASK_RATIO)
        loss = mae_loss(pred, target, mask, cfg.LOSS_ON_MASK_ONLY)
        bs = x.size(0)
        total += loss.item() * bs
        n += bs
```

```
    return total / n

for epoch in range(1, cfg.EPOCHS + 1):
    model.train()
    total, n = 0.0, 0
    print(f"Starting epoch {epoch}/{cfg.EPOCHS}...")
    for step, (x, _) in enumerate(train_loader, start=1):
        x = x.to(DEVICE, non_blocking=True)
        optimizer.zero_grad(set_to_none=True)  # clear old gradients
        pred, target, mask = model(x, mask_ratio=cfg.MASK_RATIO)  # forward pass
        loss = mae_loss(pred, target, mask, cfg.LOSS_ON_MASK_ONLY)
        loss.backward()   # compute gradients
        optimizer.step()  # update parameters

        bs = x.size(0)
        total += loss.item() * bs
        n += bs
        if step == 1 or step == len(train_loader) or step % 25 == 0:
            print(f"  step {step:03d}/{len(train_loader)} | loss {loss.item():.5f}")

    train_loss = total / n
    val_loss = evaluate(model, val_loader)
    history.append({"epoch": epoch, "train_loss": train_loss, "val_loss": val_loss})
    print(f"Epoch {epoch:02d} | train loss {train_loss:.5f} | val loss {val_loss:.5f}")

print(f"Training time: {(time.time() - start)/60:.1f} minutes")
```

Listing 18.21. MAE cell 9: training loop with validation.

The same loop as the VAE's, with AdamW and weight decay 0.05 (as in the MAE paper). Note that validation also masks randomly, so its loss is itself a little noisy.

```
if history:
    epochs = [h["epoch"] for h in history]
    plt.figure(figsize=(6, 4))
    plt.plot(epochs, [h["train_loss"] for h in history], label="train")
    plt.plot(epochs, [h["val_loss"] for h in history], label="val")
    plt.xlabel("epoch"); plt.ylabel("patch MSE")
    plt.title("MAE training curve")
    plt.legend(); plt.grid(alpha=0.3); plt.show()

show_mae_reconstructions(model, val_loader, n=8, mask_ratio=cfg.MASK_RATIO)
```

Listing 18.22. MAE cell 10: training curve and reconstructions.

```
for ratio in [0.25, 0.50, 0.75, 0.90]:
    print(f"Mask ratio: {ratio}")
    show_mae_reconstructions(model, val_loader, n=8, mask_ratio=ratio)
```

Listing 18.23. MAE cell 11: show the same model at four mask ratios.

```
RUN_LINEAR_PROBE = False
```

```

class LinearProbe(nn.Module):
    def __init__(self, dim, num_classes=10):
        super().__init__()
        self.fc = nn.Linear(dim, num_classes)
    def forward(self, x):
        return self.fc(x)

@torch.no_grad()
def collect_features(model, loader, max_batches=None):
    # Freeze MAE; collect encoder features for classification probe.
    model.eval()
    feats, labels = [], []
    for i, (x, y) in enumerate(loader):
        if max_batches is not None and i >= max_batches:
            break
        x = x.to(DEVICE)
        feats.append(model.encode_full(x).cpu())
        labels.append(y)
    return torch.cat(feats), torch.cat(labels)

if RUN_LINEAR_PROBE:
    feat_train, y_train = collect_features(model, train_loader, max_batches=40)
    feat_val, y_val = collect_features(model, val_loader, max_batches=20)
    probe = LinearProbe(cfg.EMBED_DIM).to(DEVICE)
    # Only train the linear classifier, not the MAE encoder.
    opt = torch.optim.AdamW(probe.parameters(), lr=1e-2)
    for ep in range(1, 11):
        probe.train()
        perm = torch.randperm(len(feat_train))
        for start_idx in range(0, len(perm), 512):
            idx = perm[start_idx:start_idx+512]
            xb, yb = feat_train[idx].to(DEVICE), y_train[idx].to(DEVICE)
            opt.zero_grad(set_to_none=True)
            loss = F.cross_entropy(probe(xb), yb)
            loss.backward(); opt.step()
        probe.eval()
        with torch.no_grad():
            pred = probe(feat_val.to(DEVICE)).argmax(1).cpu()
            acc = (pred == y_val).float().mean().item()
        print(f"probe epoch {ep:02d} | val acc {acc*100:.1f}%")
```

Listing 18.24. MAE cell 12: linear probe on the frozen encoder.

The probe is the VAE's, with `encode_full` (the mean of the 64 unmasked patch tokens, 128 numbers) as the feature.

18.3.5 What the run shows

The curve (Figure 18.4). Train loss 0.115 → 0.063 → 0.063 → 0.060 → 0.048 → 0.045 → 0.042 → 0.040; validation ends at 0.042. The plateau is worth understanding. The cheapest guess for a hidden patch is "the average colour of this image" — it needs no understanding and already cuts the loss in half. The model finds it in one epoch and sits there. Only from

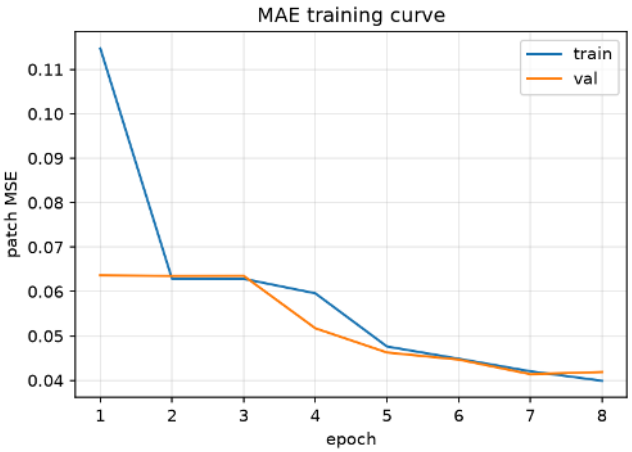


Figure 18.4. MAE loss per epoch (mean squared error on hidden patches; run for this book with the notebook's defaults, 8 epochs).

How to read it: Find the flat stretch at about 0.063 *in epochs 2–3, then the second drop.*

epoch 4 does it start to use the visible patches' *positions and content* to predict what is where, and the loss falls again.

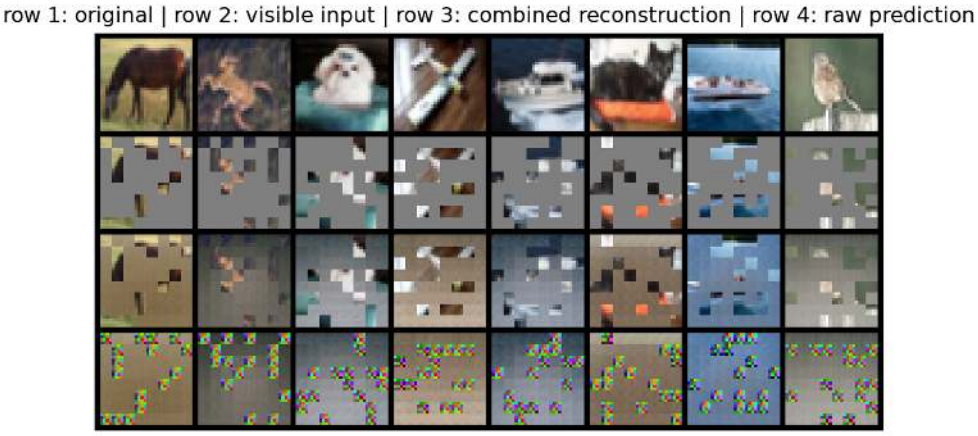


Figure 18.5. Tiny MAE at its training mask ratio (75%), after 8 CPU epochs. Rows: original; visible input (hidden patches grey); combined reconstruction; raw prediction for all patches.

How to read it: Row 3: hidden patches become smooth colour that matches the surroundings. Row 4: the patches that were *visible come out as coloured noise.*

Reconstructions (Figure 18.5). Two things to notice. *Row 3:* the filled-in patches are smooth, correctly coloured patches — sky is blue, the horse's field is brown-green — but with no detail: after 8 epochs the model predicts "what colour belongs here", not "there is a leg here". Real MAEs train for hundreds of epochs on far larger models. *Row 4:* the raw prediction at visible positions is garbage (coloured speckle). This is not a bug: with

`LOSS_ON_MASK_ONLY = True` the model is never asked to predict visible patches, so it does not bother. It is the clearest possible picture of what "loss on masked patches only" means.

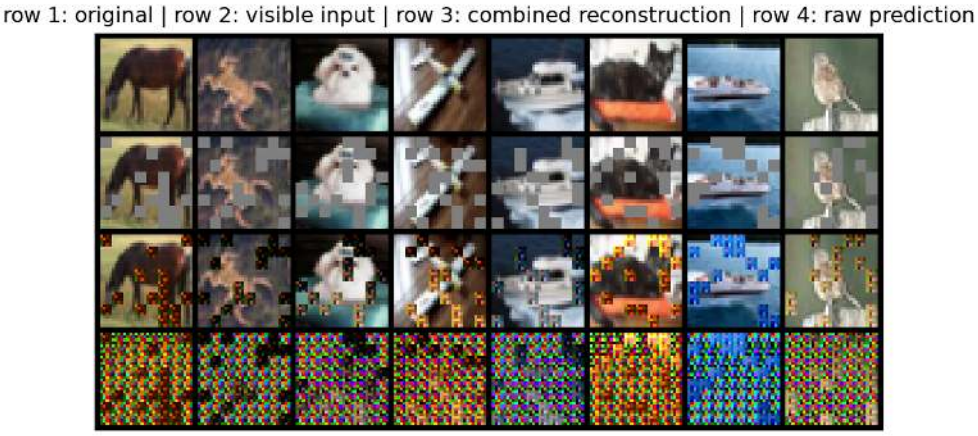


Figure 18.6. The same model, trained at 75%, shown 25% masking (cell 11). Row 3 is *worse* than at 75%.

How to read it: Compare row 3 with Figure 18.5: more visible patches, yet the filled-in patches are speckled.

The mask-ratio sweep (cell 11). At 90% masking the combined images are a grey-brown wash with a few true patches: almost no context. At 25% (Figure 18.6) one would expect the *best* reconstructions — there is the most context — but the filled-in patches come out as speckle. The reason is the notebook's own warning in the markdown above cell 11: "for a fair comparison, retrain separate models". This model has only ever seen exactly 16 visible and 48 masked tokens. With 48 visible and 16 masked, both encoder and decoder face a situation they never trained on. The lesson generalises: a model evaluated off its training distribution tells you about the distribution shift, not about the task.

Table 18.1. Linear probes on frozen features (CIFAR-10, 10 classes, chance 10%; runs for this book on a laptop CPU with the notebooks' default settings).

model	feature	probe accuracy after 10 epochs
VAE, 50 epochs, $\beta = 0.1$	$\mu(x)$, 128 numbers	38.7%
tiny MAE, 8 epochs, 75%	mean patch token, 128 numbers	31.7%

The probe. The MAE's probe climbs from about 28% to 31.7% in ten epochs — three times chance, from 8 epochs of self-supervision on 20,000 images and no labels until the probe. It is *below* the VAE's 38.7%, and that is worth reading carefully rather than concluding "VAEs beat MAEs". The comparison is not fair: the VAE trained for 50 epochs, the MAE for 8, and the MAE's loss curve was still falling. More importantly, MAE features are known to be

weak under *linear* probing and strong under fine-tuning (Tutorium 6, Question 12; §5.6); its advantage appears when the encoder is adapted, which this notebook does not do. For scale: the MAE paper's linear probe on ImageNet with a ViT-Large at 75% masking is about 73.5% (§5.7); the tiny model is not meant to compete, only to show the pipeline end to end.

18.4 Coding Tutorial 2: the play-around questions, solved

A. High versus low masking ratio (retrain at $0.25, 0.5, 0.75, 0.9$).

1. *Lowest reconstruction loss:* 0.25 — the most context, and hidden patches are surrounded by visible neighbours.
2. *Most global context:* 0.9 (and 0.75) — whole regions are missing, so neighbours are not enough (Figure 17.3).
3. *Why the best loss need not mean the best representation:* a low ratio is solvable by local blending, which teaches the encoder little about objects (week-3 Question 18). The loss ranks *how easy the task was*, not *what was learned*. The MAE paper's sweet spot was 75%; at 90% there is sometimes too little left to infer anything.

B. Compute budget and the asymmetric design. Deepening the encoder costs more per step than deepening the decoder *per layer*, but it spends compute where it is kept: the decoder is thrown away after pretraining (week-3 Question 17). Runtime grows roughly linearly with depth. Because the encoder sees only 16 tokens and the decoder 64, a decoder layer here is actually *more* expensive than an encoder layer at the same width — one reason the decoder is kept narrow (96) and shallow (2).

C. Masked-only loss versus all-patch loss (`LOSS_ON_MASK_ONLY = False`). The loss becomes lower (visible patches are easy to copy, since the encoder has seen them) and row 4 of the picture turns from speckle into a faithful copy at visible positions. But part of the training signal is now spent on copying rather than inferring. The MAE paper found masked-only loss gives better representations; that is why it is the default.

D. MAE versus full reconstruction (`MASK_RATIO = 0.0`, `LOSS_ON_MASK_ONLY = False`). Nothing is hidden: the model becomes an autoencoder *without* a bottleneck — each token of width 128 carries its own 48 pixel values, so it can copy them straight through. The loss drops close to zero quickly and the task becomes trivial. An easier pretext task gives a weaker learning signal: nothing forces the encoder to relate patches to each other (week-3 Question 19).

E. Representation check (probe a low-mask and a high-mask model). Expect the high-mask model to probe better even though its reconstruction loss is higher — the ranking by

probe is the opposite of the ranking by loss. With only 8 epochs on a CPU the gap can be small or noisy; train longer (`EPOCHS`) to see it clearly. This is the chapter's recurring lesson: *reconstruction loss ranks tasks by difficulty; only downstream tests rank representations.*

Key takeaway — both notebooks

The same two lines explain both notebooks. *The loss decides what the model spends effort on*: KL weight trades sharp copies against a usable code space; masked-only loss spends everything on inference instead of copying. *The loss does not measure what you want*: the VAE's blur, the MAE's speckle and the probe numbers only become visible when you look at outputs and test features.

CHAPTER 19

Tutorial Quiz: Generative Models — KDE, Flows and GANs

In this chapter

19.1 The one table the quiz tests 805
19.2 The ten questions 806
19.3 The quiz on one page 812

Chapter roadmap

The quiz. Ten single-choice questions about the first generative models lecture: kernel density estimation, autoencoders as generators, normalising flows, GANs, StyleGAN and conditional GANs (Pix2Pix).

What you need. Chapter 6. The one thing that makes the whole quiz easy is the table below: *what each model family can and cannot do*. Nearly every question is "which family has this ability?" or "what goes wrong when you ask a family for something it cannot do?".

19.1 The one table the quiz tests

A generative model can offer up to three services. It can *generate* new samples; it can *score* a given sample (tell you its probability $p(x)$); and it can give you a *latent code* for a sample. The families differ in which services they provide.

Family	generate	exact $p(x)$	latent code	weak spot
KDE	yes (pick a point, add kernel noise)	yes	no	high dimensions; bandwidth choice
Plain autoencoder	poorly	no	yes	code space has no known shape
VAE	yes	a lower bound only	yes	blurry samples
Normalising flow	yes	yes	yes (same size as x)	must stay invertible; restricted layers
GAN	yes, very sharp	no	no encoder by default	mode collapse; unstable training
Conditional GAN	yes, guided by an input	no	no	needs paired data (Pix2Pix)

In plain words — the three services

Generate: "draw me a new face". *Score*: "how typical is this face?" — useful for spotting anomalies, like an unusual X-ray. *Latent code*: "give me a short description of this face that I can edit". No single classic family does all three well, which is why the course keeps introducing new ones.

19.2 The ten questions

Q1 KDE with a very large bandwidth

You observed only the 1D samples $\{2.0, 2.1, 2.2, 8.0, 8.1, 8.2\}$ and fit a KDE with a very large kernel bandwidth. What is the likely consequence?

A. The density becomes smoother and may place high probability between the two groups.

B. The density becomes more accurate, because a larger bandwidth always improves generalisation.

C. KDE will automatically discover two clusters.

D. The model becomes equivalent to a GAN.

Answer: A — over-smoothing fills the gap between the groups.

The idea. KDE puts a little bump (a Gaussian of width h, the *bandwidth*) on every data point and adds the bumps up. With a small h, the six bumps form two hills, at 2 and at 8, and the valley at 5 is almost empty — correct, no data was seen there. With a huge h, every bump is wider than the gap, the bumps from both groups overlap, and the sum becomes one broad hill whose *top* can sit right at 5, where no sample ever was (Figure 19.1). *Numbers* (each bump has height $e^{-d^2/2h^2}$ at distance d): with $h = 5$, at $x = 5$ all six bumps are about

3 away and each contributes $e^{-9/50} \approx 0.84$, total ≈ 5.0. At $x = 2$ three bumps are 0 away (3×1) and three are 6 away ($3 \times e^{-36/50} \approx 3 \times 0.49$), total ≈ 4.5. The empty middle now scores *higher* than the data.

Why not the others. B: too large a bandwidth *oversmooths*; there is a best h in between. C: KDE estimates a density, it does not label clusters (although a well-chosen h shows two hills). D: a GAN is a trained neural generator; KDE has no training at all.

In the book. §6.2.

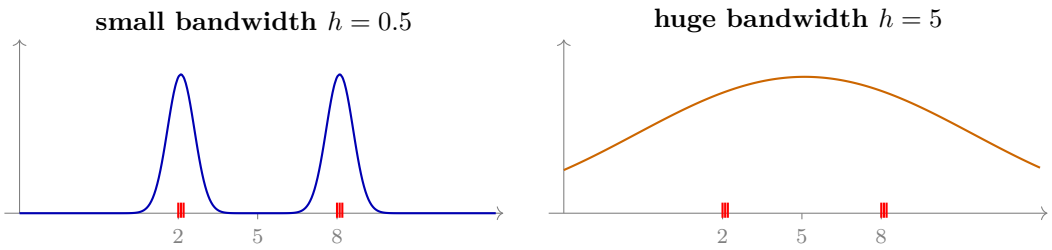


Figure 19.1. The six samples of Question 1 (red ticks) under two bandwidths. Left: two hills, an empty valley at 5. Right: one broad hill whose peak is near 5, where no data was seen.

How to read it: Compare the height of each curve at $x = 5$ with its height at the data.

Q2 KDE for face images

You want to generate realistic face images. What is the strongest argument against using KDE directly?

A. KDE cannot estimate densities.
B. KDE scales poorly in very high-dimensional spaces.
C. KDE requires labels.
D. KDE cannot generate samples.

Answer: B — the curse of dimensionality.

The idea. A 64×64 colour face is a point in 12,288 dimensions. KDE only knows "near a training face = likely". In such a huge space every face is far from every other face, so a KDE sample is just a training face with pixel noise sprinkled on it — never a new face. To cover the space densely enough for smooth interpolation you would need astronomically many examples.

Why not the others. A, C are false. D: KDE *can* sample (pick a training point at random and add Gaussian noise of width h); the samples are just not useful here.

In the book. §6.2.

Q3 Is a perfect autoencoder a good generator?

An autoencoder reconstructs images perfectly. Can it automatically be used as a good generative model?

A. Yes, because perfect reconstruction implies a meaningful latent distribution.
B. Yes, because the decoder can generate arbitrary images.
C. No, because reconstruction quality does not guarantee that latent codes follow a simple distribution.
D. No, because autoencoders cannot use latent variables.

Answer: C — you do not know where in code space to sample.

The idea. Generating means "pick a code, decode it". To pick a code you need to know which codes are valid, and a plain autoencoder never promised anything about that. This is the same point as Question 16 of the week-3 quiz (§17.3), and the reason the VAE exists.

Why not the others. A: good reconstruction says nothing about the *shape* of the code cloud. B: the decoder only knows the codes it was trained on. D: autoencoders are built around a latent code.

Q4 Great images, but no probability

A model generates excellent images, but you cannot compute the probability of an image under it. Which family was largely designed to solve this?

A. Normalising flows.
B. GANs.
C. k-means.
D. DBSCAN.

Answer: A — normalising flows.

The idea. A flow is an invertible chain of transformations from data x to a simple latent z (a standard Gaussian). Because every step can be undone, the *change-of-variables* formula gives the exact probability:

$$\log p(x) = \log p_Z(f(x)) + \log \left| \det \frac{\partial f(x)}{\partial x} \right| .$$

In words: how likely is the code, plus a correction for how much the transformation stretched or squeezed space around x. The model described in the question sounds like a GAN — the family that cannot do this.

Why not the others. B is the problem, not the solution. C, D are clustering algorithms, not generative models.

In the book. §6.3.

Q5 A flow whose inverse is expensive

A flow transformation is extremely expressive, but computing its inverse needs expensive numerical optimisation. Why is this a problem?

A. Sampling and likelihood computation become difficult.
B. The model becomes supervised.
C. The latent space disappears.
D. The model can no longer generate samples.

Answer: A — both directions of the flow are used all the time.

The idea. A flow uses one direction for *scoring* ($x \to z$, to evaluate $\log p(x)$ in every training step) and the other for *generating* ($z \to x$). If one direction needs an optimisation loop each time, one of the two services becomes slow. That is why flows use layers like the *affine coupling layer*: split x into two halves, leave one half alone and shift-and-scale the other using numbers computed from the first. Undoing it is just "unshift and unscale", and the Jacobian determinant is the product of the scales — both cheap.

Why not the others. B, C are unrelated. D is too strong: sampling stays possible, it just becomes expensive.

In the book. §6.3.

Q6 A medical imaging system

Doctors require (1) realistic samples, (2) exact likelihoods and (3) a meaningful latent representation. Which family is most suitable?

A. GAN.
B. Normalising flow.
C. KDE.
D. k-means.

Answer: B — normalising flow — the only row of the table with all three ticks.

The idea. Read requirement (2) first: it rules out GANs (and VAEs, which give only a bound). KDE has exact likelihood but no latent code and fails in image dimensions. k-means is not generative.

A caveat worth knowing. A flow's latent has the *same size* as the image, so it is not a compressed summary in the way a VAE code is. "Meaningful" here means "invertible and Gaussian-shaped": every image has exactly one code and every code one image. Flow samples are also usually less sharp than GAN samples. The answer is "best fit for these three requirements", not "best at everything".

Q7 Sketch to shoe

You want to convert a sketch of a shoe into a realistic shoe image. Which approach is most appropriate?

A. KDE.
B. A standard GAN.
C. A conditional GAN.
D. DBSCAN.

Answer: C — a conditional GAN (Pix2Pix).

The idea. The output must depend on the *input sketch*. A conditional GAN feeds the sketch to the generator ("draw a shoe that fits this outline") and also to the discriminator, which judges pairs: "is this a real (sketch, photo) pair or a fake one?". Pix2Pix adds an L1 loss to the true photo, which is possible because its training data are *paired*: each sketch comes with its photo.

Why not the others. B: a standard GAN starts from random noise and has no way to take the sketch. A, D: not image generators.

In the book. §6.4.

Q8 Every face looks the same

A GAN generates very realistic images, but every generated face looks almost identical. What is the most likely problem?

A. Overfitting.
B. Posterior collapse.
C. Mode collapse.
D. Underclustering.

Answer: C — mode collapse.

The idea. The generator is rewarded only for fooling the discriminator. If one kind of face fools it reliably, the easiest strategy is to produce that face for every noise input. The discriminator then learns to reject it, the generator jumps to another single face, and so on. Real-looking but no variety: the generator covers one *mode* (peak) of the data distribution instead of all of them.

Why not the others. A: overfitting would reproduce many *training* faces, which is still varied. B: posterior collapse is the VAE failure where the decoder ignores z (week 3, Question 14). D is not a GAN term.

In the book. §6.4.

Q9 StyleGAN without its noise inputs

Suppose you remove all stochastic noise inputs from StyleGAN. Which effect is most likely?

A. Global identity changes dramatically.
B. Fine details such as freckles, pores and hair texture become less diverse.
C. The model stops generating images.
D. The latent space disappears.

Answer: B — only the fine random details lose their variety.

The idea. StyleGAN splits the job. The latent style vector, injected at every resolution, decides the *big* things: identity, pose, face shape, hair style. Separate per-pixel noise, added at every layer, decides the *random small* things: exactly where each hair strand, freckle or pore goes — details that vary even between two photos of the same person. Remove the noise and the faces are the same people, but their skin and hair look smooth and repetitive, "painted".

Why not the others. A: identity comes from the style vector. C, D: the generator and latent space are untouched.

In the book. §6.4.

Q10 Segmentation maps to street views

You train a GAN to convert semantic segmentation maps into street-view images. Why is a conditional GAN more appropriate than a standard GAN?

A. It can use the segmentation map as additional information.
B. It removes the discriminator.
C. It guarantees perfect images.
D. It no longer needs a generator.

Answer: A — the map tells it where the road, cars and trees must go.

The idea. Same as Question 7 with a different input: the condition can be a label, a sketch, a segmentation map or a text description. The condition fixes the layout; the generator only has to fill in realistic appearance.

Why not the others. B, D: a conditional GAN still has both players. C: conditioning improves control, not perfection.

19.3 The quiz on one page

Q	Ans.	The one-line lesson
1	A	Too wide a kernel smears the groups together and fills empty gaps.
2	B	KDE cannot fill a space with thousands of dimensions.
3	C	Good reconstruction $\neq$ a code space you can sample from.
4	A	Flows give exact $\log p(x)$ by change of variables.
5	A	A flow needs both directions cheap: coupling layers do that.
6	B	Only flows tick generate + exact likelihood + latent.
7	C	Output must follow an input $\Rightarrow$ conditional GAN.
8	C	Realistic but identical samples = mode collapse.
9	B	StyleGAN noise controls random fine detail, not identity.
10	A	The condition supplies the layout.

Key takeaway — no best generative model

The answer sheet's own summary: *density estimation* $\rightarrow$ KDE (low dimensions) or flows; *representation learning* $\rightarrow$ autoencoders and VAEs; *realistic synthesis* $\rightarrow$ GANs and StyleGAN; *controlled generation* $\rightarrow$ conditional GANs and Pix2Pix. When a question names a requirement, find the row of the table in §19.1 that has it.

CHAPTER 20

Tutorial Quiz: Denoising, Diffusion and Flow Matching

In this chapter

20.1 Four facts about diffusion 813
20.2 The sixteen questions 814
20.3 The quiz on one page 823

Chapter roadmap

The quiz. Sixteen questions from Tutorial 5 (12 June 2026). Three kinds appear: *reason* (why is this true?), *debug* (a few lines of code are wrong — which change fixes them?) and *complete* (fill in the missing line). Each answer slide in the tutorial deck is followed by a "deeper" slide; its content is folded into the explanations here.
What you need. Chapter 7. The four facts in the next section answer twelve of the sixteen questions.

20.1 Four facts about diffusion

Fact 1: the forward process is a fixed recipe, with a shortcut. Noising is not learned. Starting from a clean image x_0, the noisy image at step t is

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I), \qquad \bar{\alpha}_t = \prod_{s \le t} (1-\beta_s). \tag{20.1}$$

$\bar{\alpha}_t$ is a number that slides from about 1 ($t = 0$, all image) to about 0 ($t = T$, all noise). Because it is a formula, you can jump to any t in one line.

Fact 2: training is "guess the noise I added". Pick an image, a random t and random noise ϵ; build x_t with Equation (20.1); ask the network $\epsilon_\theta(x_t, t)$ for ϵ; loss $\|\epsilon - \epsilon_\theta(x_t, t)\|^2$. The label is the noise the recipe itself added — that is why it is self-supervised.

Fact 3: generating walks the recipe backwards. Start from pure noise and repeatedly remove a little predicted noise. For flow matching, the network learns a velocity v pointing along the path; to generate you move *against* the direction it was trained to point, if that direction was data → noise.

Fact 4: who can score $p(x)$. Flows: exactly. VAEs: a lower bound (the ELBO). GANs and diffusion: only sampling is cheap — no cheap exact $p(x)$.

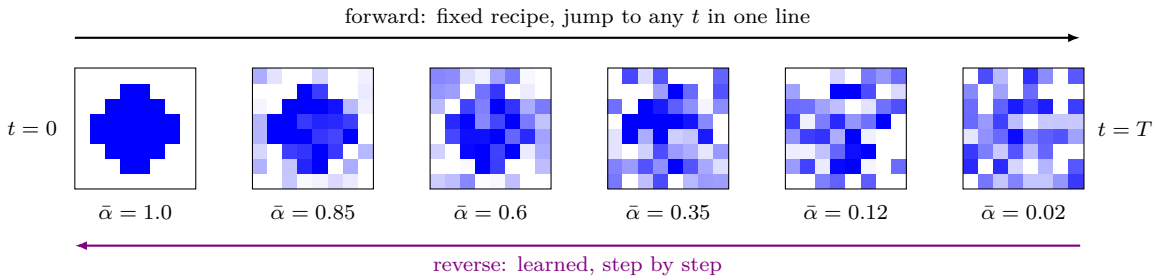


Figure 20.1. Fact 1 in pictures: a blue disc under the forward recipe. As $\bar{\alpha}_t$ falls from 1 to 0, the image fades and the noise grows. Only the violet direction is learned.

How to read it: Follow the disc from left to right; by the last frame there is almost nothing left of it (Question 15).

20.2 The sixteen questions

Q1 Why you can skip to any step

You can draw x_t for any timestep t in one step, without simulating the t intermediate noising steps. Why?

A. Composing Gaussian noising steps stays Gaussian, giving a closed-form marginal $q(x_t \mid x_0)$.

B. Because the reverse process is deterministic.

C. Because the network memorises each intermediate x_t.

D. Because β is constant, so every step is identical.

Answer: A — Gaussian plus Gaussian is Gaussian.

The idea. Each forward step shrinks the image a little and adds a little fresh Gaussian noise. Two independent Gaussian noises added together are again one Gaussian noise (their variances add). So after t steps the image has been shrunk by $\sqrt{\bar{\alpha}_t}$ and carries one Gaussian noise of variance $1 - \bar{\alpha}_t$ — Equation (20.1). In training, $\bar{\alpha}_t$ is precomputed as a table, so a random t costs nothing. *Numbers:* $\beta_1 = \beta_2 = 0.1$ gives $\bar{\alpha}_2 = 0.9 \times 0.9 = 0.81$: $x_2 = 0.9\,x_0 + \sqrt{0.19}\,\epsilon \approx 0.9\,x_0 + 0.44\,\epsilon$.

Why not the others. B: the reverse process is learned (and usually random); it has nothing to do with the forward shortcut. C: the forward process has no parameters. D: the formula holds for any β schedule — the product just uses different factors.

In the book. §7.4.

Q2 Three ways to parameterise

A diffusion model can predict the noise ϵ, the clean image x_0, or a velocity v. Which statement is correct?

A. Given x_t and t, these are linear reparameterisations of each other, so any one converts into the others.

B. Predicting x_0 needs labels, while predicting ϵ does not.

C. Only ϵ-prediction can generate samples.

D. Predicting x_0 turns the model into a normalising flow.

Answer: A — one prediction, three ways of writing it.

The idea. Equation (20.1) has three unknowns tied by one line: once you know x_t and t, knowing ϵ tells you x_0 and vice versa:

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon_\theta}{\sqrt{\bar{\alpha}_t}}.$$

So "predict the noise" and "predict the clean image" are the same model seen two ways; one picks the form that is numerically nicest. Note the $1/\sqrt{\bar{\alpha}_t}$: at large t it is huge, so the $\hat{x}_0$ read out early in sampling is rough and sharpens late.

Why not the others. B: all three targets come from the recipe, none needs a label. C: any form can sample. D: the family does not change.

Q3 Fix the training loop

ϵ_θ is meant to predict the added noise. Which single change makes the loss below match the noise-prediction objective?

```
t   = randint(1, T)
eps = randn_like(x0)
x_t = sqrt(abar[t])*x0 + sqrt(1-abar[t])*eps
loss = mse( eps_theta(x_t, t), x0 )
```

A. Compare against ϵ, not x_0: `mse(eps_theta(x_t,t), eps)`.

B. Sample ϵ from a uniform distribution, not a Gaussian.

C. Remove the time input t from the model.

D. Pass x_0 into ϵ_θ instead of x_t.

Answer: A — the target must be the noise.

The idea. Read the names: `eps_theta` is supposed to output noise, but the last line compares it with the clean image. The network would be asked to be a noise predictor and an image predictor at once. The rule: *the target must match what the output means.* Everything else in the snippet is correct.

Why not the others. B: the recipe (and its closed form) is Gaussian. C: the network must know t — the amount of noise to remove depends on it. D: feeding the clean image defeats the purpose; the network must see what it will face at generation time, a noisy x_t.

Q4 A sign error in sampling

v_θ was trained to predict $z - x$, with data x at $t = 0$ and noise z at $t = 1$. The sampler starts from noise but still returns noise. What is the mistake?

```
x = randn()                      # start at noise
for t in linspace(1, 0, N):
    x = x + v_theta(x, t)*dt     # dt > 0
```

A. The trained velocity points from data towards noise, so to generate you must subtract it and move towards data.

B. The loop should start from a real image, not from noise.

C. v_θ should be evaluated at $1 - t$ instead of t.

D. Euler integration cannot solve flow ODEs at all.

Answer: A — walk against the arrow: `x = x - v_theta(x,t)*dt`.

The idea. $v = z - x$ is an arrow *from* the data *to* the noise. The loop starts at the noise and adds the arrow — so it keeps walking further into noise. To go home to the data, step against the arrow. 1D example: data $x = 2$, noise $z = 5$, so $v = 3$. Starting at 5 with $dt = 0.5$: the buggy loop goes $5 \to 6.5 \to 8$; the fixed one goes $5 \to 3.5 \to 2$ — exactly to the data. The first thing to check in a broken sampler is the sign.

Why not the others. B: generation starts from noise — correct as written. C: the time runs from 1 to 0 correctly. D: Euler works, especially on straight paths (Question 10).

In the book. §7.5.

Q5 Noise-level mismatch

A denoising autoencoder is trained to remove Gaussian noise at $\sigma = 0.1$. You feed it images corrupted at $\sigma = 0.6$. The most likely result?

A. Perfect reconstruction, since a DAE generalises to any noise level.

B. It under-cleans and leaves visible corruption, having learned a denoiser tuned to small perturbations.

C. It outputs pure noise.

D. It returns memorised training images unchanged.

Answer: B — it cleans a bit, but far from enough.

The idea. A denoiser learns "the real image is a small step away in this direction". Trained at $\sigma = 0.1$, the steps it knows are small; an image at $\sigma = 0.6$ is six times further from the clean images, in territory it never saw. It moves the right way but stops far short. This is exactly why diffusion trains *one* network at *every* noise level (by giving it t): a denoiser that is good at every scale can walk all the way from pure noise to an image.

Why not the others. A: no free generalisation across scales. C: it still moves towards the data. D: a DAE is a map, not a lookup of training images.

In the book. §7.2.

Q6 The unconditional term in guidance

Classifier-free guidance mixes a conditional prediction ϵ_c and an unconditional one ϵ_u: $\tilde{\epsilon} = \epsilon_u + w(\epsilon_c - \epsilon_u)$. What is the role of ϵ_u?

A. It is the baseline to push away from: $(\epsilon_c - \epsilon_u)$ is the direction towards the condition, scaled by w.

B. It injects random noise to raise diversity.

C. It is a separately trained classifier.

D. It is used only during training, never at sampling.

Answer: A — the "generic" reference point.

The idea. Ask the same network twice: once with the prompt ("a cat"), once with the prompt blanked. The difference is the part of the prediction that is *due to the prompt*. Guidance takes the generic answer and adds w times that difference (Figure 20.2): $w = 0$ is unconditional, $w = 1$ plain conditional, $w > 1$ exaggerates the prompt. One network can answer both ways because during training the prompt was randomly dropped (say 10% of the time). Cost: two forward passes per step.

Why not the others. B: ϵ_u is a prediction, not injected noise. C: "classifier-free" — no classifier. D: both predictions are computed at every sampling step.

In the book. §7.7.

Q7 Turning guidance up

As you raise the guidance scale w well above 1, what trade-off do you expect?

A. More diversity and weaker prompt adherence.

B. Stronger prompt adherence but less diversity, sometimes with oversaturation.

C. No change; w only affects sampling speed.

D. Exact data likelihoods become available.

Answer: B — sharper match, less variety, artefacts if too high.

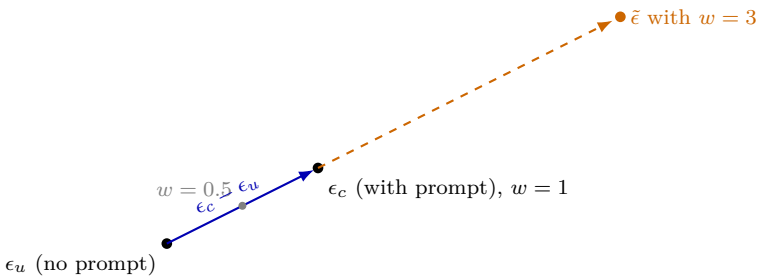


Figure 20.2. Classifier-free guidance (Questions 6 and 7). The arrow from the unconditional to the conditional prediction is "what the prompt adds". The guidance scale w slides along that line; $w > 1$ goes past the conditional prediction.

How to read it: Everything lies on one line through ϵ_u; w only says how far along it to go.

The idea. Pushing further along "what the prompt adds" (Figure 20.2) makes every sample a very typical, very on-prompt image — so samples look alike. Push too far and you leave the region of real images: colours oversaturate, textures look burnt. Useful values for images are moderate, often about 3–8. It is a fidelity versus diversity dial.

Note on the answer slide. The slide's "why it is right" and "why the others are wrong" texts are swapped for this question; the answer itself (B) is correct.

Why not the others. A is backwards. C: w changes the samples, not the speed. D: guidance does not give likelihoods.

Q8 Complete the latent sampler

A latent-diffusion sampler runs the loop below. What is the missing line?

```
z = randn(latent_shape)
for t in reversed(range(T)):
    z = denoise(z, t, cond)
# MISSING LINE
return image
```

A. `image = decoder(z)` — the autoencoder decoder maps the latent to pixels.
B. `image = encoder(z)` — encode the latent again.
C. `image = z` — the latent already is the image.
D. `image = denoise(image, 0, cond)`.

Answer: A — decode the latent back to pixels.

The idea. Latent diffusion runs in three stages: a frozen autoencoder compresses images to a small latent (e.g. $64 \times 64 \times 4$ instead of $512 \times 512 \times 3$, about 48× fewer numbers); diffusion runs there, cheaply; the decoder turns the final latent into an image. The loop has done stage 2; the missing line is stage 3.

Why not the others. B goes the wrong way (image → latent). C: a latent is not a viewable picture. D: `image` does not exist yet.

In the book. §7.8.

Q9 Which noise level for matching two different chairs

You want pixel correspondences between two *different* chairs using a pretrained diffusion U-Net, reading features at a low or a higher noise level. Which gives more robust semantic matches?

A. A higher noise level: coarser features encode semantics that transfer across instances.
B. A very low noise level: fine textures are identical across different chairs.
C. It makes no difference.
D. Neither; diffusion features cannot be used for correspondence.

Answer: A — more noise ⇒ features about parts and layout.

The idea. To denoise a nearly clean image, the network only needs local texture ("this wood grain continues"). To denoise a very noisy one, it must know what the object is and where its parts go ("the seat is here, so the legs are below"). So features read at high t describe *parts and layout*, which two different chairs share; at low t they describe *texture*, which they do not. The procedure (DIFT): add noise to both images at the chosen t, run the U-Net once, take a feature map, and match each pixel to the most similar (cosine) feature in the other image.

Why not the others. B: two different chairs have different wood and fabric. C: t changes the features a lot. D: this is a known, label-free correspondence method.

In the book. §7.10, §13.1.1.

Q10 Four steps versus a thousand

Sampler A uses 1000 Euler steps on a curved probability-flow ODE. Sampler B uses 4 Euler steps on a well-rectified, nearly straight flow and matches A's quality. The main reason?

A. Straight paths have near-constant velocity, so large Euler steps add little discretisation error.
B. Fewer steps always improve quality.
C. Rectified flows skip the noise prior.
D. Four steps suffice for any ODE regardless of curvature.

Answer: A — on a straight road you can take big strides.

The idea. An Euler step says "look at the direction here, and go straight that way for a while". On a curved path the direction changes during the stride, so you drift off the path and need tiny strides. On a straight path the direction never changes, so even one big stride lands exactly right. Rectified flow and flow matching train the paths to be nearly straight precisely so that sampling needs few steps.

Why not the others. B: fewer steps usually hurt. C: B still starts from noise. D: on a curved ODE four steps are far too coarse.

In the book. §7.5.

Q11 Can you trust a higher ELBO?

A team changes only their encoder, and the ELBO goes up. Can they conclude that the model assigns higher true likelihood $\log p(x)$?

- **A.** Not necessarily: the ELBO is a lower bound, and it can rise because the bound got tighter, not because $\log p(x)$ grew.
- **B.** Yes; the ELBO equals $\log p(x)$.
- **C.** No; the ELBO is an upper bound on $\log p(x)$.
- **D.** Only if the decoder is linear.

Answer: A — the bound may simply have become tighter.

The idea. The exact relation is

$$\log p(x) = \text{ELBO} + \text{KL}(q(z \mid x) \,\|\, p(z \mid x)), \qquad \text{KL} \geq 0.$$

Think of $\log p(x)$ as a ceiling, the ELBO as a floor, and the KL as the gap. $\log p(x)$ depends only on the *decoder*. Changing only the encoder cannot move the ceiling; it can only close the gap. *Numbers:* $\log p(x) = -100$, ELBO $= -110$, gap 10; a better encoder gives gap 4, ELBO -104, while $\log p(x)$ is still -100.

Why not the others. B: equality only when the gap is zero. C: it is a *lower* bound. D is irrelevant.

In the book. §4.5.1.

Q12 Spot the false claim

Which statement about diffusion models versus normalising flows is *false*?

- **A.** A normalising flow can return an exact log-density for a test point.
- **B.** A standard diffusion model exposes an exact, cheap $\log p(x)$ at sampling time.
- **C.** Diffusion generates by iterating a learned denoiser.
- **D.** A flow needs an invertible map with a tractable Jacobian.

Answer: B — diffusion does not offer cheap exact likelihoods.

The idea. Fact 4. A flow scores by change of variables (A and D are true). Diffusion samples by chaining denoising steps (C is true). A diffusion model's likelihood can be computed or bounded, but only at great cost (e.g. by integrating the probability-flow ODE with a trace estimate); it is not a cheap basic interface.

Note on the answer slide. The slide's explanation calls the false statement "A" by mistake; its "Correct: B" is right.

Q13 The score from a denoiser (Tweedie)

An optimal Gaussian denoiser returns $D(x_t) = \mathbb{E}[x_0 \mid x_t]$ at noise level σ. By Tweedie's identity, which expression equals the score $\nabla \log p(x_t)$?

A. $(D(x_t) - x_t)/\sigma^2$
B. $(x_t - D(x_t))/\sigma^2$
C. $(D(x_t) - x_t) \cdot \sigma^2$
D. $\sigma^2 \cdot x_t$

Answer: A — denoised minus noisy, divided by σ^2.

The idea. Tweedie's identity reads $\mathbb{E}[x_0 \mid x_t] = x_t + \sigma^2 \nabla \log p(x_t)$; solve for the score. In words: the *score* is the arrow pointing towards where noisy images are more likely — towards the data. The denoiser's correction $D(x_t) - x_t$ is exactly such an arrow ("the clean image is over there"). Dividing by σ^2 just fixes the scale. So training a denoiser *is* learning the score; that is why a denoiser can generate, and it unifies denoising autoencoders, score matching and diffusion. *1D check:* data are all at 0, $\sigma = 1$. Then $p(x_t)$ is $\mathcal{N}(0, 1)$ with score $-x_t$, and the best denoiser returns 0. At $x_t = 2$: $(0 - 2)/1 = -2$ — matches.

Why not the others. B: sign flipped, points away from the data. C: wrong scale. D ignores the denoiser.

Q14 One network or a thousand

You replace the single time-conditioned denoiser $\epsilon_\theta(x_t, t)$ by 1000 independent networks, one per timestep, trained on the same total data. The most likely consequence?

A. Each network sees far less data and shares nothing across noise levels, so quality drops and cost explodes.
B. Identical quality at lower cost.
C. Exact likelihoods become computable.
D. The forward noising process changes.

Answer: A — 1/1000 of the data each, 1000× the parameters.

The idea. Denoising at $t = 500$ and at $t = 501$ is almost the same job. One network with a time input shares what it learns across all noise levels (*amortisation*); 1000 separate networks each learn from a thousandth of the examples and store their own copy of everything. A shared network also lets you change the number of sampling steps later.

Why not the others. B: cost rises, quality falls. C: unrelated. D: the forward recipe is fixed whatever the network looks like.

Q15 Where x_t knows least about x_0

In a variance-preserving schedule, $\bar{\alpha}_t$ falls from about 1 at $t = 0$ to about 0 at $t = T$. At which t does x_t carry the least information about x_0?

A. t near T, where $\bar{\alpha}_t \approx 0$ and x_t is almost pure noise.
B. t near 0, where $\bar{\alpha}_t \approx 1$.
C. Always the midpoint.
D. Information about x_0 is constant across t.

Answer: A — at the end of the schedule.

The idea. The image's share of x_t is $\sqrt{\bar{\alpha}_t}$, the noise's share $\sqrt{1-\bar{\alpha}_t}$. Their ratio, the signal-to-noise ratio $\mathrm{SNR}(t) = \bar{\alpha}_t/(1-\bar{\alpha}_t)$, only falls with t (Figure 20.1). The schedule is chosen so that the work is spread evenly across the steps.

Why not the others. B is where it knows the *most*. C, D: the information decreases steadily.

Q16 Is it really self-supervised?

A paper trains a diffusion model to imitate expert robot trajectories and calls the method "self-supervised". The most accurate critique?

A. It is supervised imitation learning: the targets are expert action labels, not signals derived from the data itself.
B. It is self-supervised, because diffusion models are always self-supervised.
C. It is unsupervised, because there is no classifier.
D. It is reinforcement learning, because a robot is involved.

Answer: A — the targets come from an expert.

The idea. The question to ask is *where does the target come from?* Made from the data itself (hide a part and predict it, denoise, next frame) ⇒ self-supervised. Supplied by a person or expert (labels, demonstrations of the right action) ⇒ supervised. A diffusion policy trained on expert actions has the second kind of target, whatever the model is called. The model class never decides the supervision type.

Why not the others. B confuses model and signal. C: having no classifier does not remove the labels. D: copying fixed demonstrations involves no reward.

In the book. §13.2.3 asks the same question for world models.

20.3 The quiz on one page

Q	Ans.	The one-line lesson
1	A	Gaussians compose, so x_t has a one-line formula.
2	A	ϵ-, x_0- and v-prediction convert into each other.
3	A	The loss target must mean what the output means.
4	A	Generate by stepping against the trained arrow.
5	B	One noise level is not enough; diffusion trains them all.
6	A	ϵ_u is the generic baseline; guidance pushes away from it.
7	B	High w: on-prompt but samey, sometimes burnt.
8	A	Latent diffusion ends with the decoder.
9	A	Noisier features describe parts, which transfer across instances.
10	A	Straight paths let a few big Euler steps suffice.
11	A	A better encoder tightens the bound; $\log p(x)$ is unchanged.
12	B	Diffusion has no cheap exact likelihood.
13	A	A denoiser's correction, divided by σ^2, is the score.
14	A	One time-conditioned network amortises across noise levels.
15	A	Least information at $t \approx T$.
16	A	The label source, not the model, decides the supervision type.

Key takeaway — the debugging habit

Three of these questions were broken code (Q3, Q4, Q8). All three bugs are found by the same question: *what does this variable mean?* What does the network's output mean (noise, not image)? Which way does the trained arrow point (towards noise)? What space does z live in (latents, not pixels)? Name the meaning, and the bug is visible.

CHAPTER 21

Tutorial Quiz: Pretext Tasks, PIRL and Contrastive Learning

In this chapter

21.1 The map of the quiz 825
21.2 Questions 1–7: predicting a transformation 825
21.3 Questions 8–10: PIRL 828
21.4 Questions 11–12: evaluating features 829
21.5 Questions 13–18: contrastive learning 830
21.6 The quiz on one page 833

Chapter roadmap

The quiz. Tutorium 6: eighteen single-choice questions. They follow the story of Lectures 06–07: predict a transformation (RotNet, Jigsaw) → stop predicting it and become invariant to it instead (PIRL) → how to evaluate features → contrastive learning (InfoNCE, temperature, SimCLR, MoCo) and its main failure (false negatives).

What you need. Chapters 8 and 9. One distinction runs through half the quiz: *covariance* (features change in a known way when the input is transformed) versus *invariance* (features do not change at all). Section 21.1 draws it.

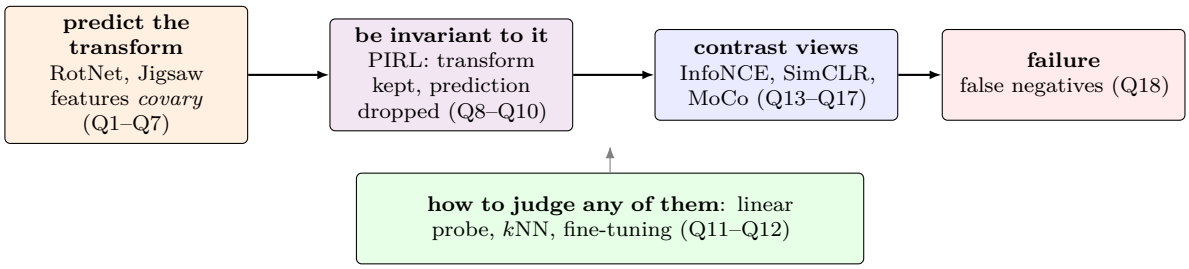


Figure 21.1. The route the eighteen questions take, with the question numbers at each stop.

How to read it: Left to right is the historical order: each box fixes a weakness of the box before it.

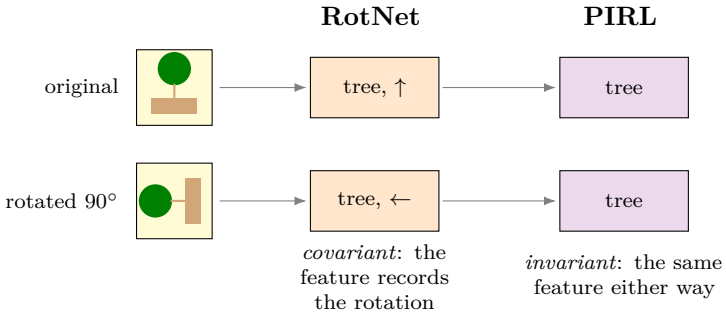


Figure 21.2. The distinction behind Questions 3, 7, 8 and 15. To *predict* a rotation, the features must record it (covariant). To be useful for "what is in the picture", they should not care (invariant).

How to read it: Compare the two boxes in each column: RotNet's differ, PIRL's are identical.

21.1 The map of the quiz

21.2 Questions 1–7: predicting a transformation

Q1 Which SSL family?

A model is trained to predict which augmentation was applied to an image. Which SSL family is this?

A. Contrastive learning.
B. Reconstruction-based SSL.
C. Pretext task prediction.
D. A generative model.

Answer: C — pretext task prediction.

The idea. The course sorts image SSL into three families by *what the network outputs*: *reconstruct* pixels (autoencoders, inpainting, MAE); *predict the transformation* (RotNet: "which of four rotations?", Jigsaw: "which permutation?"); *contrast* two views (make two

views of one image similar, others different). "Which augmentation was applied?" is a classification of the transformation.

Why not the others. A compares two views; B rebuilds pixels; D models the data distribution.

In the book. §8.6.

Q2 Why a pretext task works without labels

Why can a pretext task give useful features without any human labels?

A. Because the pretext label is the class labels in disguise.
B. Because larger batches always help.
C. Because the loss equals supervised training.
D. Because the task cannot be solved without modelling real object structure.

Answer: D — the task is designed to need real structure.

The idea. To tell a photo of a dog from the same photo turned 90°, you must know that dogs stand on their legs, that sky is at the top, that heads are above bodies. The "label" (the rotation) is free, but the only way to predict it is to learn what things look like. The supervision comes from the *difficulty the designer built in*, not from the loss formula (which is ordinary cross-entropy).

Why not the others. A: rotation labels carry no class information. B, C are not reasons.

Q3 RotNet and PIRL: covary or invariant?

How do RotNet and PIRL features relate to the applied transformation?

A. RotNet covaries with the transform and PIRL is invariant to it.
B. Both are invariant to the transform.
C. RotNet is invariant and PIRL covaries.
D. Both covary but use different losses.

Answer: A — RotNet covaries, PIRL is invariant.

The idea. Figure 21.2. If the output must name the rotation, the features must contain it: rotating the input changes the feature in a predictable way. PIRL pulls the features of an image and its transformed version together, so the transformation is removed and only the content remains.

In the book. §8.4.

Q4 Four rotations, not eight

Why does RotNet use four rotations rather than eight?

A. Eight classes overfit the data.
B. Eight rotations make the softmax unstable.
C. Adjacent angles are harder to tell apart and odd angles need warping.
D. With eight rotations the network cannot learn edges.

Answer: C — 45° steps need resampling and are hard to tell apart.

The idea. A 90° turn just rearranges the pixel grid: no pixel is invented, no corner is cut. A 45° turn needs interpolation and leaves empty triangular corners. Those artefacts tell the network the angle without any understanding (a *shortcut*, Question 6), and 0° versus 45° is also a much subtler visual difference. The RotNet paper found that eight rotations did not help.

Why not the others. A, B, D are not the reason.

Q5 Where rotation gives a weak signal

On which images does rotation prediction give the weakest signal?

A. Upright photos of animals and vehicles.
B. Street photos of people and buildings.
C. Indoor scenes with furniture.
D. Top-down aerial images and symmetric objects.

Answer: D — images with no natural "up".

The idea. RotNet relies on photos having a canonical up direction. A satellite view of a field, a plate seen from above, or a star-shaped flower looks equally right in all four orientations: the task becomes unsolvable, or solvable only by accident, and teaches little.

Q6 Preventing a shortcut

Which choice prevents a shortcut in a pretext task?

A. Restrict RotNet to right-angle turns so no border artefacts appear.
B. Use the largest possible batch size.
C. Increase the projection head dimension.
D. Train for more epochs with a cosine schedule.

Answer: A — remove the low-level cue that gives the answer away.

The idea. A *shortcut* is a cheap clue that solves the pretext task without the understanding it was designed to teach: interpolation blur, black corners, chromatic aberration at image borders, the exact alignment of Jigsaw tiles. Good pretext design removes the clue (right angles only; gaps and jitter between Jigsaw tiles), so that the real task is the easiest way left.

Why not the others. B, C, D change optimisation, not the clue.

Q7 Jigsaw features: best in the middle

Jigsaw features are best at a middle block and worse at the last block. Why?

A. The last block has too few parameters.
B. Batch norm corrupts the last block.
C. The last block stays randomly initialised.
D. The last block specialises to the pretext and loses semantics.

Answer: D — the last layers become "permutation detectors".

The idea. Layers close to the output serve the output. For Jigsaw the output is "which of 1000 permutations?", so the last block encodes tile order — useful for the puzzle, useless for "is this a cat?". Middle layers hold general visual features. This is the practical cost of covariance, and PIRL fixes it: trained for invariance, its last block stays the best one.

In the book. §8.8, §8.9.

21.3 Questions 8–10: PIRL

Q8 What PIRL keeps and drops

What does PIRL keep and what does it drop from classification Jigsaw?

A. Keeps the jigsaw transform and replaces prediction with an invariance objective.
B. Keeps the permutation prediction head and drops the transform.
C. Keeps both and adds reconstruction.
D. Drops both and trains a generative model.

Answer: A — keep the shuffle, stop predicting it.

The idea. The Jigsaw shuffle is a good *augmentation*: it destroys the global layout and keeps local content. What was harmful was *predicting* it (Question 7). PIRL (Pretext-Invariant Representation Learning) shuffles the tiles, encodes the shuffled image, and asks that the result be close to the encoding of the original — and far from other images.

Q9 Why a memory bank

Why does PIRL use a memory bank?

A. To store ground-truth class labels.
B. To make features invariant by averaging them.
C. To hold many cheap negatives as cached image features.
D. To reduce the negatives to one.

Answer: C — thousands of negatives without computing them.

The idea. A contrastive loss works better with many negatives, but a batch holds only a few hundred images. The memory bank keeps one stored feature per training image (updated slowly as a running average whenever that image is seen), so each step can compare against thousands of stored features at no extra forward-pass cost.

Why not the others. A: no labels exist. B: the running average makes the stored features *stable*; the invariance comes from the loss. D is backwards.

Q10 What the λ term adds

PIRL mixes two NCE terms with weight λ; $\lambda = 0$ gives plain instance discrimination. What does the λ term add?

A. A supervised classification signal.
B. More negatives in the denominator.
C. A switch to Euclidean distance.
D. An invariance constraint tying the transformed view to the memory feature.

Answer: D — "the shuffled image must look like this image's stored feature".

The idea. PIRL's loss has two parts. One term compares the *untransformed* image with its memory-bank feature: "recognise yourself among other images" (instance discrimination). The λ-weighted term compares the *transformed* (jigsaw) image with the same memory feature: "recognise yourself even when shuffled" — that is the invariance. Setting $\lambda = 0$ drops the shuffle entirely, and PIRL's reported advantage over plain instance discrimination comes from this term.

In the book. §9.1.

21.4 Questions 11–12: evaluating features

Q11 Linear probe, kNN and fine-tuning

How do linear probe, kNN and fine-tuning differ as evaluations?

A. All three measure the same thing.
B. Fine-tuning reflects the initialisation, while probe and kNN test the frozen features.
C. kNN needs a trained classifier like the linear probe.
D. Fine-tuning is the only valid measure.

Answer: B — three protocols, three questions.

The idea.

protocol	what is trained	question it answers
kNN	nothing	are same-class images neighbours in feature space?
linear probe	one linear layer; backbone frozen	can one straight cut separate the classes?
fine-tuning	every weight	is this a good starting point for training?

Why not the others. A: see the table. C: kNN trains nothing. D: each is valid for its own question.

Q12 Weak probe, strong fine-tune

A model has low linear-probe accuracy but matches a supervised baseline after fine-tuning. What does this show?

A. The weights are a good initialisation even if frozen features are not separable.
B. The frozen features are linearly separable.
C. Probe and fine-tuning must always agree.
D. The model memorised the training set.

Answer: A — good starting point, not good frozen features.

The idea. Fine-tuning can rearrange the features, so it only needs the weights to be a good place to start. MAE is the standard example: its linear-probe accuracy is modest but it fine-tunes very well (§5.6). Always say which protocol a number comes from.

21.5 Questions 13–18: contrastive learning

Q13 What InfoNCE does

InfoNCE scores one positive key against many negatives. What is it doing?

A. Maximising similarity of the query to every key.
B. Minimising Euclidean distance between all keys.
C. Classifying which key is the positive among the negatives.
D. Maximising the entropy of the scores.

Answer: C — a classification: "which one is my partner?".

The idea. Compute the similarity of the query q with every key, divide by the temperature τ, apply softmax, and use cross-entropy with the positive as the correct "class":

$$\mathcal{L} = -\log \frac{\exp(q \cdot k_+/\tau)}{\exp(q \cdot k_+/\tau) + \sum_i \exp(q \cdot k_i^-/\tau)}.$$

Numbers ($\tau = 1$): similarity 0.9 to the positive, 0.2 and 0.1 to two negatives: softmax probability of the positive $= e^{0.9}/(e^{0.9} + e^{0.2} + e^{0.1}) = 2.46/(2.46 + 1.22 + 1.11) \approx 0.51$, loss ≈ 0.67.

Why not the others. A: raising similarity to *every* key would raise the negatives too and increase the loss. B, D: not what the formula does.

In the book. §9.3.

Q14 Lowering the temperature

What does lowering the temperature in InfoNCE mainly do?

A. It weights all negatives equally.
B. It focuses the gradient on the hardest negatives and can hurt if too low.
C. It removes the need to normalise embeddings.
D. Lower temperature always improves accuracy.

Answer: B — a hard-negative knob, not a "smaller is better" dial.

The idea. Dividing by a small τ magnifies differences. Take two negatives with similarities 0.5 (hard, close to the query) and 0.1 (easy). Their weights in the softmax are in the ratio $e^{(0.5-0.1)/\tau}$: about 1.5 at $\tau = 1$, but $e^4 \approx 55$ at $\tau = 0.1$. So a low temperature makes the loss almost entirely about the closest negatives. Too low, and it fixates on a few — which may be *false* negatives, images of the same class (Question 18).

In the book. §9.4.

Q15 Loss after the head, features before it

SimCLR computes the loss after a projection head but uses the features before it. Why?

A. The output has too few dimensions.
B. The output is not normalised.
C. The head is never trained.
D. The head discards augmentation-related information that downstream tasks need.

Answer: D — the head throws away what the loss told it to ignore.

The idea. The contrastive loss demands invariance to colour, crop and blur. The small head g can absorb that demand: its output z forgets colour, but the backbone feature h below it keeps it. Downstream, colour often matters ("is this a ripe banana?"), so we keep h and throw g away. SimCLR reports linear-probe accuracy over 10 points higher on h than on z.

In the book. §9.7.

Q16 The critical augmentation pair

SimCLR found one augmentation pair critical. Which one, and why?

A. Rotation and cutout, for spatial invariance.
B. Crop and colour distortion, because crops share a colour-histogram shortcut.
C. Blur and greyscale, to remove texture.
D. Colour distortion alone, since colour is the key cue.

Answer: B — crop plus colour.

The idea. Two random crops of the same photo have almost the same colour distribution (both mostly green grass and blue sky). Matching them by colour histogram alone solves the task — a shortcut. Randomly distorting colour breaks that clue, so the network must match them by *content*. Either augmentation alone is much weaker than the pair.

In the book. §8.2, §9.7.

Q17 MoCo's momentum

What is MoCo's momentum encoder for, and what happens at $m = 0$?

A. It keeps queued keys consistent; at $m = 0$ they change too fast and training fails.
B. It adds negatives; at $m = 0$ the queue grows unbounded.
C. It adds noise; at $m = 0$ accuracy is best because keys are fresh.
D. It replaces normalisation; at $m = 0$ nothing changes.

Answer: A — consistency of the keys in the queue.

The idea. MoCo stores keys from many past batches in a queue (the source of many negatives). If the key encoder changed quickly, keys computed a few hundred steps ago would come from a very different network than today's: comparing against them would be comparing apples to old oranges. So the key encoder is a slow average of the query encoder: $\theta_k \leftarrow m\,\theta_k + (1 - m)\,\theta_q$ with $m = 0.999$. With $m = 0$ the key encoder is just a copy of the query encoder that changes every step, and the MoCo paper reports that training fails. *Queue buys many negatives; momentum buys their consistency.*

In the book. §9.8.

Q18 False negatives

What is the false-negative (class collision) problem?

A. The two views of one image are treated as negatives.
B. The positive pair has low similarity early on.
C. Other images of the same class are used as negatives and pushed apart.
D. The softmax denominator overflows.

Answer: C — two different cat photos are treated as "different".

The idea. Instance discrimination calls *every* other image a negative. With 10 classes and a batch of 256, about 25 of the 255 negatives of a cat image are also cats — and the loss pushes them away. The larger the dictionary and the lower the temperature, the more these collisions matter. Non-contrastive methods (next chapter's quiz) and supervised-contrastive variants are two ways around it.

In the book. §9.6.

21.6 The quiz on one page

Q	Ans.	The one-line lesson (the tutorial's own takeaways)
1	C	Three families: reconstruct, predict the transform, contrast views.
2	D	The designed difficulty is the supervision.
3	A	Predicting a transform forces covariance; semantics want invariance.
4	C	Right-angle turns leave no artefact to exploit.
5	D	RotNet needs a canonical "up".
6	A	Make the shortcut harder than the real task.
7	D	The layer next to a pretext head over-specialises.
8	A	PIRL keeps the transform, drops its prediction.
9	C	The memory bank is an efficiency trick for many negatives.
10	D	PIRL = instance discrimination + an invariance term.
11	B	Probe, kNN and fine-tuning answer different questions.
12	A	A strong fine-tune rates the initialisation.
13	C	Contrastive learning is picking the positive among many.
14	B	Temperature is a hard-negative knob.
15	D	Keep the backbone, drop the projection head.
16	B	Crop + colour breaks the colour-histogram shortcut.
17	A	Queue: many negatives; momentum: consistent keys.
18	C	Every other image as negative repels same-class images.

CHAPTER 22

Tutorial Quiz and Lab: Representation Learning, SimSiam and DINO

In this chapter

22.1 The quiz 835
22.2 The lab: making collapse visible 841 • The notebook, cell by cell • What the lab shows • The reflection prompts, answered

Chapter roadmap

The material. A fifteen-question quiz and a small CPU lab that makes its ideas visible. The quiz covers what a representation is, how to evaluate one, SimSiam, BERT, DINO and DINOv2, collapse, data curation, and three "what would you do?" case studies.
The one idea to carry through. *A self-supervised loss can be excellent while the features are useless* (collapse). Questions 6 and 12 test it directly, and the lab demonstrates it with real numbers: a model that outputs the same vector for every input scores a *better* loss than a genuinely trained SimSiam, and is worthless.
What you need. Chapters 10 and 11.

22.1 The quiz

Q1 The goal of representation learning

Which statement best describes the goal of representation learning?

A. Assign every training example a unique identifier so the model can memorise the dataset.

B. Replace all downstream evaluation with the self-supervised training loss.

C. Map raw data into a feature space that preserves relevant structure and removes nuisance variation.

D. Force all augmented views and all unrelated samples to have identical representations.

Answer: C — keep what matters, drop what does not.

The idea. An encoder maps raw data X (pixels) to features Z that are easier to use: two photos of the same kind of object should land near each other even if lighting and angle differ (*invariance* to nuisances), while properties you need, such as position for a robot, are kept (*covariance*).

Why not the others. A is memorisation. B: the loss does not tell you whether features are useful (Question 6). D describes collapse.

Q2 How features are evaluated

Which set contains common methods for evaluating learned representations?

A. PCA or t-SNE inspection, attention masks, clustering, kNN, linear probing and fine-tuning.

B. Only the number of model parameters and the number of training epochs.

C. Only manual inspection of the raw input images before training.

D. Only the self-supervised pretraining loss, because downstream tests introduce labels.

Answer: A — look at the features and test them.

The idea. Two kinds of evaluation. *Look*: project features to 2D (PCA, t-SNE) and see whether classes form groups; inspect attention maps. *Test*: kNN, linear probe, a small head, fine-tuning (§21.4). Using a few labels for evaluation is normal; they are never used for pretraining.

Why not the others. B, C are not about the features. D: the loss alone can hide collapse.

Q3 SimSiam's components

Which components are central to SimSiam?

A. A decoder, pixel reconstruction loss, KL term and Gaussian prior.

B. A density threshold, ε-neighbourhoods, core points and outlier labels.

C. A masked-language-model head, next-sentence classifier, token embeddings and word-piece vocabulary.

D. A backbone, projection MLP, prediction MLP, symmetric loss, and stop-gradient on the target branch.

Answer: D — backbone, projector, predictor, symmetric loss, stop-gradient.

The idea. Two augmented views go through the same backbone and projector, giving z_1 and z_2. A predictor MLP turns each into p_1 and p_2. The loss makes p_1 point in the same direction as z_2 and p_2 as z_1 (negative cosine similarity, averaged over both ways), and the target z in each term is *detached*: no gradient flows into it. The lab's code for exactly this is cell 4 in §22.2.1.

Why not the others. A is a VAE, B is DBSCAN, C is BERT.

Q4 BERT's two objectives

Which pair of pretraining objectives is associated with BERT?

A. Pixel reconstruction and KL regularisation.

B. Masked word prediction and next-sentence prediction.

C. k-means assignment and DBSCAN outlier detection.

D. Image crop prediction and object detection with bounding boxes.

Answer: B — masked words and next sentence.

The idea. Hide 15% of the words and predict them from both sides of the context; and given two sentences, predict whether the second really followed the first. Both labels come from the text itself.

In the book. §11.7.

Q5 DINOv2's training signals

Which combination matches DINOv2's training signals?

A. Pixel reconstruction, a KL term to a Gaussian prior, and decoder-only fine-tuning.

B. Cross-entropy on human labels, hard-negative mining and supervised segmentation masks.

C. DINO loss on the CLS token, iBOT loss on masked patch tokens, and a KoLeo regulariser.

D. A DBSCAN clustering loss, nearest-neighbour density estimation and class-balanced reweighting.

Answer: C — DINO + iBOT + KoLeo.

The idea. Three terms, three jobs. *DINO loss* on the CLS token: the student's whole-image summary must match the teacher's (image level). *iBOT loss* on masked patches: the student sees an image with some patches masked and must predict the teacher's features for those patches (patch level, like a feature-space MAE). *KoLeo*: a regulariser that pushes each feature away from its nearest neighbour in the batch, so features spread out evenly instead of bunching up.

In the book. §11.5.

Q6 Low loss, useless features

Why can a self-supervised model have low training loss but still learn useless features?

A. The model can collapse to nearly constant representations that satisfy the agreement loss but do not separate examples.

B. Collapse only happens in supervised learning.

C. A low self-supervised loss always means the downstream task has been solved.

D. kNN accuracy is unrelated to feature quality and should never be checked.

Answer: A — collapse.

The idea. An agreement loss says "two views of the same image must get the same feature". Output the same vector for *every* image and every pair agrees perfectly. Loss: minimal. Information: none. The lab measures this exactly: the constant model's loss is -1.000, the best possible, while SimSiam's is about -0.91 (Figure 22.1).

In the book. §10.1.

Q7 Stop-gradient and the predictor

What is the main conceptual role of stop-gradient together with the asymmetric predictor in SimSiam?

A. It replaces augmentation, so both branches receive identical raw inputs.

B. It freezes the entire network after initialisation.

C. It creates explicit negative pairs from unrelated images.

D. It makes one branch act as a fixed target during each loss term, while the predictor introduces asymmetry that helps avoid trivial collapse.

Answer: D — one side is a fixed target; the predictor makes the sides different.

The idea. Without stop-gradient, both branches are free to move towards each other, and the quickest way to agree is for both to shrink to the same constant. With stop-gradient, in each term only the predicting side moves, towards a target that holds still for that step. The predictor makes the two sides structurally different, so "output the same constant" is no

longer a stable solution. SimSiam's paper showed that removing stop-gradient collapses the model immediately.

Why not the others. A: two views are still needed. B: only the target *gradient* is stopped; all weights train. C: SimSiam has no negatives.

In the book. §10.3.

Q8 DINO's anti-collapse tricks

Why does DINO use teacher centring, sharpening and EMA teacher updates?

A. They make student and teacher parameters identical after every minibatch.

B. They stabilise teacher targets, prevent one output dimension from dominating, and make the teacher distribution informative without explicit negatives.

C. They convert the model into a supervised classifier trained on ImageNet labels.

D. They remove the need for augmentations and multi-crop training.

Answer: B — three counterweights against two kinds of collapse.

The idea. DINO's output is a probability distribution over, say, 65,536 "prototypes". It can collapse in two ways: every image puts all its weight on *one* prototype, or every image outputs the *flat* uniform distribution. *Centring* (subtract the running mean output before the softmax) stops one prototype from winning for everyone. *Sharpening* (a low teacher temperature) stops the flat answer. The two pull in opposite directions and balance. *EMA* makes the teacher a slow average of the student, so the targets are stable.

In the book. §11.3.

Q9 Patch features from an image-level loss

How can DINO produce useful patch-level features although the loss is applied to the CLS token?

A. The CLS token is never connected to patch tokens.

B. Patch-level structure appears only because segmentation labels are used.

C. In a vision transformer, patch tokens interact through self-attention, so optimising the global token forces patch tokens to encode object and region structure.

D. Patch tokens are discarded before the transformer.

Answer: C — the CLS token is built from the patches.

The idea. The CLS token has no information of its own; at every layer it collects information from the patch tokens by attention. To make the CLS summary of a small crop match the summary of the whole image, the patches must carry "this is part of a dog" — which is why DINO's attention maps outline objects without ever seeing a mask.

Q10 Hierarchical k-means for data balancing

Why is hierarchical k-means useful for large-scale SSL data preparation?

A. It supports balanced sampling across high-level concepts and sub-concepts in long-tailed web data.

B. It guarantees that every image has a human-written label.

C. It removes the need for deduplication, filtering or test-leakage checks.

D. It only makes training faster and has no effect on the data distribution.

Answer: A — balance many common and rare concepts.

The idea. A web crawl has millions of cat photos and a handful of photos of a rare bird. Train on it as is and the model is mostly a cat expert. Cluster the embeddings (first coarse clusters, then clusters inside each), then sample evenly across clusters: rare concepts get their fair share. The lab does a flat version in miniature (Figure 22.3).

In the book. §11.6.

Q11 Two million plant photos, 2000 labels

You have 2,000,000 unlabelled plant images, do not know the number of species, and can label only 2,000. Which strategy is most appropriate?

A. Train a large supervised classifier from scratch on the 2,000 labels and ignore the rest.

B. Pretrain with SSL, cluster or retrieve neighbours in feature space, label a selected subset, then train a classifier or prototype model.

C. Label the first 2,000 images in file order and deploy without checking class coverage.

D. Run DBSCAN on raw pixels and use its cluster IDs as species labels.

Answer: B — use the unlabelled data first, then spend labels wisely.

The idea. A recipe in four steps. (1) Pretrain on all two million images: the unlabelled data are the most valuable thing you have. (2) Cluster the features: each cluster is roughly a species or a look-alike group, and the number of clusters hints at how many species there are. (3) Spend the 2,000 labels *across* clusters (say 20 per cluster) instead of on whatever comes first. (4) Train a light classifier on the frozen features, and check it on held-out labels.

Why not the others. A wastes 99.9% of the data. C risks labelling only a few species. D: raw-pixel distance does not know species.

Q12 Diagnosing collapse

A SimSiam-like model reaches very low training loss, but kNN accuracy on frozen features is near random and a PCA plot shows almost all examples at one point. Best diagnosis and next step?

A. The model is ready; low loss is the only metric that matters.
B. The labels must be wrong; PCA cannot show collapse.
C. Remove all augmentations so both branches see the same input.
D. It likely collapsed; check feature variance and fix the anti-collapse design (stop-gradient, predictor asymmetry) and representation-quality evaluation.

Answer: D — collapse: check per-dimension spread, then fix the anti-collapse parts.

The idea. Low loss + chance-level kNN + one dot in PCA is the textbook signature — exactly the lab's "collapsed constant" row (Table 22.1, Figure 22.2). First check: the standard deviation of each feature dimension across the dataset; if it is near zero, all inputs give the same feature. Then check that the stop-gradient really is there (a missing `.detach()` is a classic bug) and that the predictor exists.

Why not the others. C makes agreement trivial and collapse *easier*.

Q13 Evaluating robotics features with few labels

You want to evaluate SSL features for a robot perception module, with only a few task labels. Which plan is strongest?

A. Use the SSL loss as the only evaluation.
B. Fine-tune the whole backbone immediately and judge by training accuracy.
C. Use frozen-feature kNN or retrieval, train a low-capacity probe on the small labels, test invariances, and only then consider fine-tuning.
D. Only visualise raw camera images.

Answer: C — cheap frozen tests first, fine-tuning last.

The idea. With few labels, fine-tuning a big backbone overfits and tells you little. Frozen tests are cheap and honest: do nearby features belong to the same situation (kNN)? can a small head read the task out (probe)? does the feature stay the same under a change of lighting you do not care about (invariance test)?

Q14 A web-scale data engine

You build a DINOv2/v3-style training set from web images plus curated seed datasets. Which pipeline best reduces data-quality and evaluation risks?

A. Train on the raw crawl; SSL automatically removes duplicates and unsafe content.

B. Deduplicate, filter unsafe or sensitive content, remove near-duplicates of test data, embed and cluster, retrieve near curated seeds, and sample with balancing.

C. Remove images after evaluation if they turn out to duplicate the test set.

D. Use the largest model and skip filtering; scale dominates quality.

Answer: B — clean, de-leak, embed, retrieve, balance — in that order.

The idea. Each step removes one risk: duplicates (wasted compute and memorisation), unsafe content, *test leakage* (a near-copy of a test image in training inflates the score), imbalance (Question 10), and drift away from useful images (retrieval near curated seed sets keeps the pool relevant). Leakage must be removed *before* training, not discovered after.

Q15 Choosing an evaluation method

You have a frozen SSL encoder and want to know whether its features are useful before expensive fine-tuning. Which choice best matches the goals?

A. kNN for neighbourhood quality, linear probing for linear separability, fine-tuning later for final performance.

B. Training loss only.

C. Fine-tuning only.

D. PCA only, treating visually separated clusters as proof of accuracy.

Answer: A — each method answers its own question.

The idea. Same table as Tutorium 6, Question 11. PCA is a picture, useful for spotting collapse, but a 2D projection can hide or invent structure; it is not a measurement.

22.2 The lab: making collapse visible

The lab runs on a CPU in seconds. It uses the scikit-learn *digits* data set: 1,797 handwritten digits, each an 8×8 grey image flattened to 64 numbers in $[0, 1]$. It is split into 1,347 training and 450 test images. Labels are used only for evaluation. Three feature spaces are compared:

1. **raw pixels** — the 64 numbers themselves, a sanity baseline;
2. **collapsed constant** — a fake "model" that maps every image to the same unit vector: a deliberately collapsed representation;
3. **SimSiam-style** — a small MLP encoder trained with SimSiam for 35 epochs.

22.2.1 The notebook, cell by cell

Every code cell of the lab follows, in order and unchanged. The notebook's comments already name the quiz question each piece serves.

```python
# Core imports for a small CPU-only representation-learning experiment.
# The notebook uses sklearn digits so it runs quickly in Colab without downloads.
import random
import warnings
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

# Some local sklearn/numpy stacks emit noisy matmul RuntimeWarnings even when the result
    ↪ is fine.
warnings.filterwarnings("ignore", category=RuntimeWarning, module=r"sklearn\..*")

# Fixed seeds make the lab repeatable enough for discussion.
SEED = 7
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

# Force CPU so the notebook behaves the same on most student machines and Colab runtimes.
device = torch.device("cpu")
print(f"Using device: {device}")
```

Listing 22.1. Week-8 lab cell 1: imports and seeds. Everything runs on the CPU.

Standard imports. The scikit-learn tools do the evaluation: `KMeans` for clustering, `LogisticRegression` as the linear probe, `KNeighborsClassifier` for kNN. `device` is forced to the CPU so every student gets the same behaviour.

```python
# Q11 and Q13 discuss large visual datasets, but this lab needs a tiny built-in proxy.
# Each digit image is an 8x8 grayscale image flattened into 64 raw pixel values.
digits = load_digits()
X = digits.data.astype(np.float32) / 16.0
Y = digits.target.astype(np.int64)

# Stratification keeps all digit classes represented in both train and test splits.
X_train, X_test, y_train, y_test = train_test_split(
    X, Y, test_size=0.25, random_state=SEED, stratify=Y
)

```

```
# Torch tensors are used for SSL-style training; numpy arrays are kept for sklearn evaluation.
X_train_t = torch.tensor(X_train, dtype=torch.float32, device=device)
X_test_t = torch.tensor(X_test, dtype=torch.float32, device=device)

print("train:", X_train.shape, "test:", X_test.shape)
print("classes:", sorted(set(Y.tolist())))
```

Listing 22.2. Week-8 lab cell 2: the digits data.

Pixel values in the digits set run from 0 to 16, so dividing by 16 puts them in $[0, 1]$. A stratified 75/25 split keeps all ten digits in both halves. Each image stays a flat vector of 64 numbers, which is why an MLP (not a CNN) is enough.

```
# Treat a feature dimension std below this value as numerically constant.
# This is used to flag collapsed representations in Q6/Q12.
CONSTANT_FEATURE_STD_THRESHOLD = 1e-5


class Encoder(nn.Module):
    """Small MLP encoder: maps raw 64-dimensional pixels to a feature vector z."""

    def __init__(self, in_dim=64, feat_dim=32):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 128),
            nn.ReLU(),
            nn.Linear(128, feat_dim),
        )

    def forward(self, x):
        return self.net(x)


class CollapsedModel(nn.Module):
    """Reference collapse case for Q6/Q12: every input maps to the same nonzero vector."""

    def __init__(self, feat_dim=32):
        super().__init__()
        constant = torch.ones(feat_dim, dtype=torch.float32)
        constant = constant / constant.norm()
        self.register_buffer("constant", constant)

    def encode(self, x):
        # The representation is nonzero, so cosine agreement can be excellent.
        # It is still useless because every example receives the same feature vector.
        constant = self.constant.to(dtype=x.dtype)
        return constant.unsqueeze(0).repeat(x.shape[0], 1)
```

Listing 22.3. Week-8 lab cell 3, part 1: the encoder and the deliberately collapsed model.

`Encoder`: $64 \to 128 \to 32$ with a ReLU; its 32 outputs are the representation that is evaluated. `CollapsedModel` is not trained at all: it returns the vector $(1, \ldots, 1)/\sqrt{32}$ for every input. It is the "perfectly collapsed" reference against which SimSiam is compared.

```
36
37 class MLP(nn.Module):
38     """Projection or prediction MLP used in SimSiam-style training."""
39
40     def __init__(self, dim=32, hidden=64):
41         super().__init__()
42         self.net = nn.Sequential(
43             nn.Linear(dim, hidden),
44             nn.BatchNorm1d(hidden),
45             nn.ReLU(),
46             nn.Linear(hidden, dim),
47         )
48
49     def forward(self, x):
50         return self.net(x)
51
52
53 class ToySimSiam(nn.Module):
54     """Tiny SimSiam-style model: encoder + projector + predictor."""
55
56     def __init__(self):
57         super().__init__()
58         self.encoder = Encoder()
59         self.projector = MLP()
60         self.predictor = MLP()
61
62     def encode(self, x):
63         # Downstream evaluation uses the frozen encoder features.
64         return self.encoder(x)
65
66
67 def augment(x, noise_std=0.12, drop_prob=0.15):
68     """Create two cheap views of the same input: pixel dropout plus noise."""
69     keep = (torch.rand_like(x) > drop_prob).float()
70     noisy = x * keep + noise_std * torch.randn_like(x)
71     return noisy.clamp(0.0, 1.0)
72
73
74 def negative_cosine(p, z):
75     """Agreement objective used by SimSiam: lower is better, minimum is near -1."""
76     p = F.normalize(p, dim=1)
77     z = F.normalize(z, dim=1)
78     return -(p * z).sum(dim=1).mean()
```

Listing 22.4. Week-8 lab cell 3, part 2: projector and predictor, the SimSiam wrapper, views and the loss.

`MLP` (lines 37–51) is used twice, as projector and as predictor: $32 \to 64 \to 32$ with batch normalisation. `ToySimSiam.encode` returns only the encoder output: projector and predictor

are training scaffolding and are dropped for evaluation, just like SimCLR's head (Tutorium 6, Question 15). `augment` (lines 67–71) makes a view: keep each pixel with probability 0.85, add noise of standard deviation 0.12, clip to $[0, 1]$. `negative_cosine` (lines 74–78) normalises both vectors to length 1 and returns minus the mean dot product: -1 for identical directions, 0 for perpendicular ones.

```
81 def extract_features(model, X_tensor):
82     """Freeze a model and return numpy features for kNN, probing, and PCA."""
83     model.eval()
84     with torch.no_grad():
85         feats = model.encode(X_tensor).detach().cpu().numpy()
86     return feats
87
88
89 def feature_stats(feats):
90     """Simple collapse diagnostics: per-dimension spread and feature norm."""
91     per_dim_std = feats.std(axis=0)
92     norms = np.linalg.norm(feats, axis=1)
93     return {
94         "mean_dim_std": float(per_dim_std.mean()),
95         "min_dim_std": float(per_dim_std.min()),
96         "max_dim_std": float(per_dim_std.max()),
97         "mean_norm": float(norms.mean()),
98     }
99
100
101 def majority_baseline_accuracy():
102     """Accuracy when features contain no information and the best guess is class
    ↪ prior."""
103     majority_label = np.bincount(y_train).argmax()
104     pred = np.full_like(y_test, majority_label)
105     return accuracy_score(y_test, pred)
106
107
108 def evaluate_features(train_feats, test_feats, label="features"):
109     """Evaluate frozen features for Q2, Q12, Q13, and Q15.
110
111     kNN checks local neighborhood quality. A linear probe checks whether labels are
112     linearly accessible from the frozen features. Feature std helps detect collapse.
113     """
114     stats = feature_stats(train_feats)
115
116     if stats["max_dim_std"] < CONSTANT_FEATURE_STD_THRESHOLD:
117         # kNN has arbitrary tie behavior on identical vectors, so report the class-prior
    ↪ baseline.
118         knn_acc = majority_baseline_accuracy()
119         probe_acc = knn_acc
120         note = " | constant features"
121     else:
122         knn = KNeighborsClassifier(n_neighbors=5)
123         knn.fit(train_feats, y_train)
124         knn_acc = accuracy_score(y_test, knn.predict(test_feats))
125
126         probe = LogisticRegression(max_iter=1000, random_state=SEED, solver="liblinear")
```

```
        probe.fit(train_feats, y_train)
        probe_acc = accuracy_score(y_test, probe.predict(test_feats))
        note = ""

    print(
        f"{label:18s} | kNN: {knn_acc:.3f} | linear probe: {probe_acc:.3f} | "
        f"mean dim std: {stats['mean_dim_std']:.4f} | mean norm:
    ↪ {stats['mean_norm']:.4f}{note}"
    )
    return {"label": label, "knn": knn_acc, "linear_probe": probe_acc, **stats}
```

Listing 22.5. Week-8 lab cell 3, part 3: the evaluation helpers.

- `extract_features`: run a frozen model over all images.
- `feature_stats`: the collapse diagnostics — the standard deviation of each of the 32 feature dimensions across the data set (averaged, and its minimum and maximum), and the average vector length. If every dimension has spread ≈ 0, all inputs map to the same point.
- `majority_baseline_accuracy`: the accuracy of always guessing the most common digit, used when features are exactly constant (then kNN would only break ties arbitrarily).
- `evaluate_features`: kNN with $k = 5$ and a logistic regression probe, both fitted on the training features and scored on the test features, plus the statistics above in one printed line.

```
def train_simsiam(epochs=35, batch_size=256, lr=1e-3):
    """Train the toy SimSiam model for Q7.

    The key idea is asymmetry: one branch predicts the other branch, while the
    target branch is detached with stop-gradient. This is the anti-collapse
    mechanism tested conceptually in Q7.
    """
    model = ToySimSiam().to(device)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    losses = []

    for epoch in range(epochs):
        model.train()
        perm = torch.randperm(X_train_t.shape[0], device=device)
        epoch_loss = 0.0
        batches = 0

        for start in range(0, len(perm), batch_size):
            idx = perm[start:start + batch_size]
            x = X_train_t[idx]

            # Two augmented views stand in for the multi-view SSL setup.
            x1 = augment(x)
            x2 = augment(x)

            # Shared encoder/projector processes both views.
            z1 = model.projector(model.encoder(x1))
```

```
            z2 = model.projector(model.encoder(x2))

            # Predictor creates the asymmetric student branch.
            p1 = model.predictor(z1)
            p2 = model.predictor(z2)

            # stop-gradient is the .detach(): the target branch does not receive
    ↪ gradients.
            loss = 0.5 * negative_cosine(p1, z2.detach()) + 0.5 * negative_cosine(p2,
    ↪ z1.detach())

            opt.zero_grad()
            loss.backward()
            opt.step()

            epoch_loss += float(loss.item())
            batches += 1

        losses.append(epoch_loss / max(1, batches))

    return model, losses


def collapsed_reference_loss(model, X_tensor):
    """Show the Q6/Q12 trap: a constant nonzero representation gets loss near -1."""
    with torch.no_grad():
        z = model.encode(X_tensor[:256])
        return float(negative_cosine(z, z).item())


print("Creating collapsed nonzero constant-representation reference...")
collapse_model = CollapsedModel().to(device)
collapse_loss_value = collapsed_reference_loss(collapse_model, X_train_t)
collapse_losses = [collapse_loss_value] * 35
print(f"Collapsed reference negative-cosine objective: {collapse_loss_value:.3f}")

print("Training SimSiam-style model...")
simsiam_model, simsiam_losses = train_simsiam(epochs=35)

plt.figure(figsize=(7, 3))
plt.plot(collapse_losses, label="collapsed constant reference")
plt.plot(simsiam_losses, label="SimSiam-style training")
plt.xlabel("epoch")
plt.ylabel("training objective")
plt.title("Low agreement objectives can still hide collapsed features")
plt.legend()
plt.show()
```

Listing 22.6. Week-8 lab cell 4: SimSiam training and the collapsed reference.

`train_simsiam` is Questions 3 and 7 in code. Each step: shuffle the training images into batches of 256; make two views; run both through encoder and projector, then the predictor; the loss is the symmetric negative cosine with `.detach()` on the targets — that `.detach()` *is* the stop-gradient. `collapsed_reference_loss` computes the same loss for the constant

model: since both "views" map to the same vector, it is exactly −1. The cell then plots both curves (Figure 22.1).

```
# Q2/Q13/Q15: compare several feature spaces before expensive full fine-tuning.
# Raw pixels are included as a sanity baseline, not as an SSL method.
raw_result = evaluate_features(X_train, X_test, "raw pixels")

# Q6/Q12: this representation has an excellent agreement objective but no class
    ↪ information.
collapse_train = extract_features(collapse_model, X_train_t)
collapse_test = extract_features(collapse_model, X_test_t)
collapse_result = evaluate_features(collapse_train, collapse_test, "collapsed constant")

# Q7/Q13/Q15: evaluate the frozen encoder learned by the SimSiam-style objective.
simsiam_train = extract_features(simsiam_model, X_train_t)
simsiam_test = extract_features(simsiam_model, X_test_t)
simsiam_result = evaluate_features(simsiam_train, simsiam_test, "SimSiam-style")

results = [raw_result, collapse_result, simsiam_result]
results
```

Listing 22.7. Week-8 lab cell 5: evaluate three feature spaces with the same tests.

```
def pca_2d(feats):
    """Compute a 2D PCA projection with numpy for visualization in Q2/Q12."""
    centered = feats - feats.mean(axis=0, keepdims=True)
    with warnings.catch_warnings():
        warnings.simplefilter("ignore", RuntimeWarning)
        _, _, vt = np.linalg.svd(centered, full_matrices=False)
        return centered @ vt[:2].T


def plot_pca(ax, feats, labels, title):
    """Plot feature geometry. Collapsed features become a single point."""
    if feats.std(axis=0).max() < CONSTANT_FEATURE_STD_THRESHOLD:
        xy = np.zeros((feats.shape[0], 2), dtype=np.float32)
    else:
        xy = pca_2d(feats)
    scatter = ax.scatter(xy[:, 0], xy[:, 1], c=labels, s=12, cmap="tab10", alpha=0.75)
    ax.set_title(title)
    ax.set_xticks([])
    ax.set_yticks([])
    return scatter

fig, axes = plt.subplots(1, 3, figsize=(14, 4))
plot_pca(axes[0], X_train, y_train, "Raw pixels")
plot_pca(axes[1], collapse_train, y_train, "Collapsed constant features")
plot_pca(axes[2], simsiam_train, y_train, "SimSiam-style features")
plt.suptitle("PCA views of feature spaces")
plt.tight_layout()
plt.show()
```

Listing 22.8. Week-8 lab cell 6: PCA pictures of the three feature spaces.

`pca_2d` is PCA in three lines: centre the features, take the SVD, project onto the first two right-singular vectors (the two directions of largest spread). A constant feature set has nothing to decompose, so `plot_pca` draws it as a single point.

```
rng = np.random.default_rng(SEED)

# Q14 data-engine setup: create a deliberately imbalanced pool.
# Digits 0 and 1 are overrepresented; the other digits are scarce.
imbalanced_idx = []
for cls in range(10):
    cls_idx = np.where(y_train == cls)[0]
    keep_frac = 1.0 if cls in (0, 1) else 0.22
    keep_n = max(8, int(len(cls_idx) * keep_frac))
    imbalanced_idx.extend(rng.choice(cls_idx, size=keep_n, replace=False).tolist())
imbalanced_idx = np.array(imbalanced_idx)

# Use the learned representation, not raw pixels, to organize the pool.
imb_feats = simsiam_train[imbalanced_idx]
imb_labels = y_train[imbalanced_idx]

# Q10/Q14: cluster in feature space, then cap examples per cluster.
# This is a small flat version of the balancing idea behind hierarchical k-means.
n_clusters = 20
clusters = KMeans(n_clusters=n_clusters, random_state=SEED,
    ↪ n_init=10).fit_predict(imb_feats)
selected_local = []
cap_per_cluster = 8
for cluster_id in np.unique(clusters):
    members = np.where(clusters == cluster_id)[0]
    keep_n = min(cap_per_cluster, len(members))
    selected_local.extend(rng.choice(members, size=keep_n, replace=False).tolist())
selected_local = np.array(selected_local)

selected_clusters = clusters[selected_local]
selected_labels = imb_labels[selected_local]

# Cluster counts show what the sampler actually controls.
# Digit counts are printed only as a diagnostic because a real web crawl would not have
    ↪ labels.
cluster_before = np.bincount(clusters, minlength=n_clusters)
cluster_after = np.bincount(selected_clusters, minlength=n_clusters)
label_before = np.bincount(imb_labels, minlength=10)
label_after = np.bincount(selected_labels, minlength=10)

fig, axes = plt.subplots(1, 2, figsize=(10, 3), sharey=True)
axes[0].bar(np.arange(n_clusters), cluster_before)
axes[0].set_title("Cluster sizes before cap")
axes[0].set_xlabel("cluster id")
axes[0].set_ylabel("count")
axes[1].bar(np.arange(n_clusters), cluster_after)
axes[1].set_title("Cluster sizes after cap")
axes[1].set_xlabel("cluster id")
plt.suptitle("Embedding clusters can support mode-capped sampling")
plt.tight_layout()
plt.show()
```

```
50
51 print("Total before:", len(imb_labels), "Total after:", len(selected_labels))
52 print("Largest cluster before:", int(cluster_before.max()), "after:",
       ↪ int(cluster_after.max()))
53 print("Hidden digit counts before:", label_before.tolist())
54 print("Hidden digit counts after: ", label_after.tolist())
```

Listing 22.9. Week-8 lab cell 7: cluster-capped sampling on an unbalanced pool.

The first loop builds the unbalanced pool: keep all 0s and 1s and only 22% of each other digit (at least 8). Then it takes the *SimSiam features* of that pool — not labels, not pixels — runs k-means with 20 clusters, and keeps a random 8 from each cluster (all of them if it has fewer). The digit labels are counted at the end only to check the effect.

22.2.2 What the lab shows

Result 1: the collapsed model wins on loss. Figure 22.1. The constant model scores -1.000 — every pair of views points in exactly the same direction because every output is the same vector. SimSiam improves from about -0.1 to about -0.91 and never reaches the collapsed model's value.

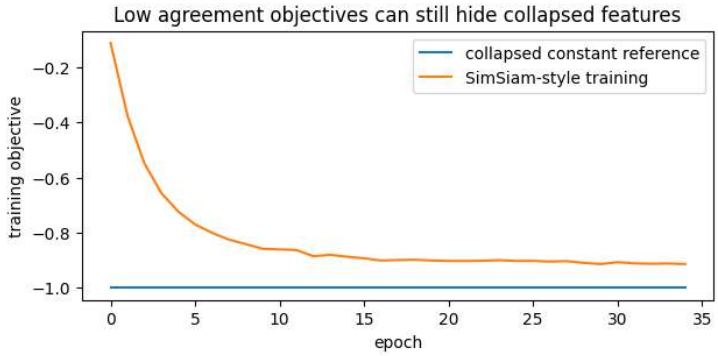


Figure 22.1. The lab's training curves (real output of the week-8 notebook). Blue: the collapsed constant model, flat at the best possible value -1. Orange: SimSiam training, which settles near -0.91.

How to read it: The lower line belongs to the useless model. Loss alone would pick it.

Result 2: evaluations expose it. The notebook's evaluation function reports kNN accuracy ($k = 5$), a logistic-regression linear probe, and two collapse diagnostics: the average per-dimension standard deviation across the data set and the average feature length.

Read the table row by row. The collapsed model is at chance on both tests, and its per-dimension standard deviation is exactly zero — the fastest possible check (Question 12). SimSiam's features are genuinely informative (94% kNN) and spread out.

Table 22.1. The lab's frozen-feature evaluation (real outputs, test set of 450 digits; chance is about 0.10).

features	kNN	linear probe	mean dim. std	mean norm
raw pixels	0.989	0.964	0.230	3.87
collapsed constant	0.102	0.102	0.000	1.00
SimSiam-style	0.944	0.882	0.164	1.17

Common Confusion

"But raw pixels beat SimSiam — so SSL failed?" No. These are 8×8 digits, a tiny and very clean data set on which raw pixels are already almost linearly separable. The lab is not a competition; its point is the *contrast* between a collapsed and a non-collapsed model under the same tests. On real photos, where pixels vary with lighting and position, raw-pixel kNN is poor and learned features win by a wide margin. The notebook's final prompt says the same: do not memorise the numbers, reason about the comparison.

Result 3: the PCA picture. Figure 22.2. PCA projects each 32-dimensional feature onto the two directions of greatest spread. Raw pixels and SimSiam both show digit groups (colours are the hidden labels, used only for colouring). The collapsed model is a single dot.

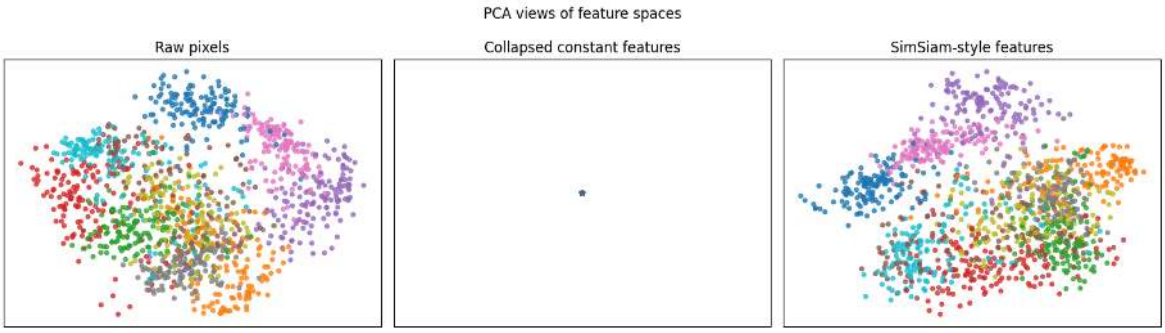


Figure 22.2. PCA of the three feature spaces (real output of the week-8 notebook). Colour: the hidden digit label. Middle: all 1,347 training digits land on one point.

How to read it: Look for groups of one colour: two panels have them, the middle one has nothing to look at.

Result 4: cluster-capped sampling (Questions 10 and 14). The last cell imitates a web crawl in miniature. It builds a deliberately unbalanced pool: all 0s and 1s are kept, but only 22% of every other digit — 501 images, of which 269 are 0s or 1s. Then, *without using any label*, it clusters the SimSiam features into 20 groups with k-means and keeps at most 8 images per cluster.

The hidden digit counts, printed only to check the result, show what happened:

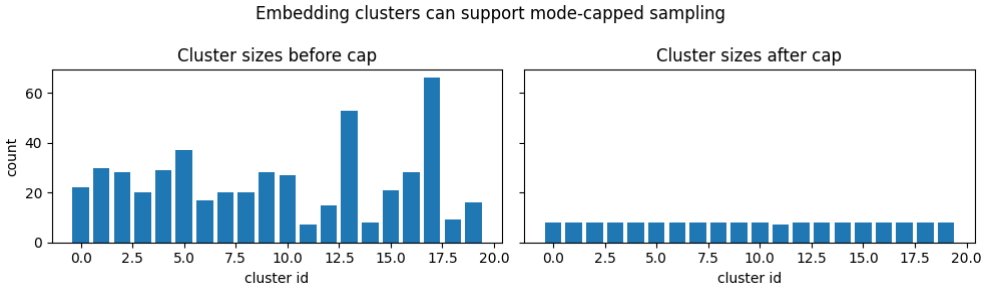


Figure 22.3. Cluster sizes before and after capping at 8 per cluster (real output). Before: the largest cluster has 66 images, the smallest 7. After: every cluster contributes at most 8.

How to read it: The tall bars on the left are the over-represented modes; the cap cuts them down without looking at any label.

digit	0	1	2	3	4	5	6	7	8	9	total
before	133	136	29	30	29	29	29	29	28	29	501
after	29	38	12	11	15	13	16	8	9	8	159

Before, 0s and 1s were 54% of the pool; after, 42%. The balance improves but is not perfect, because clusters are not digits: 0s and 1s occupy several clusters each, and some rarer digits share a cluster. That is the honest lesson of the data-engine questions: clustering the embedding *reduces* dominance of common modes; how well depends on how well the features separate concepts.

22.2.3 The reflection prompts, answered

1 (Q11). If these digits were plant images, where would you use SSL features before asking for labels? Everywhere before labelling: pretrain on all unlabelled images, cluster the features to see how many groups there are and how big, then choose which images to label so that every cluster is covered (the cluster-capped cell is exactly such a label-free selection step). After labelling, fit a probe or kNN on the frozen features first.

2 (Q12). Which signals show the collapsed representation is useless despite its low objective? Three, all in the lab: kNN and probe at chance (0.102); per-dimension standard deviation exactly 0; and a PCA plot that is a single point. Its objective (-1.000) is *better* than SimSiam's, which is why the objective cannot be the test.

3 (Q13). Why kNN and a linear probe before full fine-tuning? They need no backbone training, run in seconds, and with few labels they cannot overfit much (the probe here has only 32×10 weights plus 10 biases). They answer "is the information there and easy to read out?"; fine-tuning a large network on a few labels mostly measures how well it can memorise them.

4 (Q14). Which parts of the cluster-capped cell resemble a DINOv2/v3 data engine? Embedding the pool with a self-supervised model; clustering in that embedding space; sampling with a per-cluster cap to balance frequent and rare modes; never using labels. What the toy leaves out: deduplication, safety filtering, test-leakage removal, retrieval around curated seed sets, and the *hierarchical* clustering.

5 (Q15). What does PCA answer that kNN or a probe does not? It shows the *shape* of the whole feature cloud: one point (collapse), a few dominant directions, separated groups, or a continuum. kNN and probes give single accuracy numbers for one chosen labelling. PCA needs no labels, but a 2D projection can hide structure, so it is a picture for diagnosis, not a score.

Key takeaway — week 8

> Never trust a self-supervised loss on its own. Measure the features: the per-dimension spread (collapse check), kNN and a linear probe (content check), a PCA plot (sanity picture). And when you curate data, the same features let you balance a pool you cannot label.

CHAPTER 23

Tutorial Quiz and Case Study: Self-Training, Cycle Consistency and World Models

In this chapter

23.1 The quiz: questions 1–12 855
23.2 The quiz: case studies 13–18 860
23.3 The case-study notebook, cell by cell ..... 863 • Setup: data, the small CNN and helpers • Section 1: self-training with a confidence filter (Q13) • Section 2: soft targets and heterogeneous teachers (Q14, Q15) • Section 3: unpaired translation and cycle consistency (Q16, Q17) • Section 4: an action-conditioned world model and planning (Q18)
23.4 What the notebook shows 882
23.5 The final reflection questions, answered 885

Chapter roadmap

The material. An eighteen-question quiz and a case-study notebook that builds a small working version of every idea in it. The notebook's own first line says what unites them: three ways to get supervision without labelling every target — *a model labels for another model* (pseudo-labels, distillation), *an input labels its own round trip* (cycle consistency), and *the next observation labels the present* (world models).

The link between the two. Quiz Questions 1–12 are knowledge questions; Questions 13–18 are case studies, and each case study has a section of the notebook that produces evidence for its answer:

quiz	notebook section	evidence produced
Q13	1	confidence filtering, kept true labels, a perturbed student
Q14	2A	soft teacher probabilities versus a hard argmax target
Q15	2B	one encoder with a semantic head and an edge head
Q16	3	two generators, two discriminators, unpaired batches
Q17	3	target-looking translations versus round-trip error
Q18	4	action-conditioned latent prediction, then sequence search

What you need. Chapters 12 and 13.

23.1 The quiz: questions 1–12

Q1 What a hard pseudo-label is

A classifier produces probabilities for an unlabelled input. What is a pseudo-label in the standard hard-label approach?

A. The average of all hidden ground-truth labels in the data set.
B. A random class sampled before the model sees the input.
C. The class selected by taking the argmax of the model's predicted probabilities.
D. The complete probability distribution produced by an ensemble.

Answer: C — the most probable class, treated as if it were the label.

The idea. The model says $[0.7, 0.2, 0.1]$; the pseudo-label is class 1, written as the one-hot target $[1, 0, 0]$, even though the model was only 70% sure. Keeping the whole vector instead is *soft* targets (distillation, Question 4).

In the book. §12.1.

Q2 Clustering as the teacher

In a deep-clustering method, what supplies the targets used to train the feature network?

A. Cluster assignments computed from the current feature representations.
B. Human class labels added after every clustering iteration.
C. Pixel reconstruction targets produced by a decoder.
D. Random identifiers that stay fixed for every example.

Answer: A — the clusters of the model's own features.

The idea. DeepCluster alternates: extract features for all images; run k-means; use each image's cluster number as its "class"; train the network to predict it; repeat with the improved features. The clustering algorithm plays teacher.

Why not the others. B: no human labels. C is an autoencoder. D: fixed random IDs teach memorisation.

Q3 The Noisy Student loop

Which sequence best describes iterative Noisy Student training?

A. Train only on labelled data, remove all augmentation, keep the first model forever.

B. Assign random labels to unlabelled data, train a smaller model, discard the labelled set.

C. Train a student on clean inputs and a teacher on strongly perturbed inputs, without updating either.

D. Train a teacher on labelled data, pseudo-label unlabelled data, train a perturbed student on both sets, then use the student as the next teacher.

Answer: D — teacher → pseudo-labels → noisy student → new teacher.

The idea. Four points make it work. The teacher labels *clean* images (its best guesses). The student trains on *noisy* inputs (augmentation, dropout, stochastic depth), so it must be right in harder conditions than the teacher was. The student is *equal or larger*, so it has room to exceed the teacher. It trains on real *and* pseudo-labels, and then becomes the next teacher. Notebook section 1 does one round of this.

Q4 Soft targets versus hard pseudo-labels

What distinguishes soft-target knowledge distillation from ordinary hard pseudo-labelling?

A. Distillation removes the teacher and trains only from random augmentations.

B. Distillation matches the teacher's full output distribution instead of only its most likely class.

C. Distillation requires a human label for every unlabelled example.

D. Distillation always uses pixel reconstruction.

Answer: B — copy the whole distribution.

The idea. The student minimises the cross-entropy (equivalently, the KL divergence) between the teacher's full probability vector and its own. The "wrong" classes carry information: a teacher that says "shirt 0.6, T-shirt 0.3, sneaker 0.0" teaches that shirts resemble T-shirts and not sneakers — Hinton called this "dark knowledge".

In the book. §12.2.

Q5 Heterogeneous teachers in DUNE

What is the main purpose of heterogeneous teachers in a DUNE-style system?

A. Combine complementary signals such as 2D semantics, human pose and 3D geometry in one shared student representation.

B. Force every teacher to use the same architecture, data, outputs and loss.

C. Replace all teacher outputs by one hard class label.

D. Train separate students that never share information.

Answer: A — one student that knows what several specialists know.

The idea. DUNE distils DINOv2 (semantics), Multi-HMR (human pose and shape) and a 3D-geometry model (MASt3R) into one encoder. Each teacher gets its own small projector head on the student, so outputs of different shapes can each be matched with their own loss, while the shared encoder learns from all of them. Notebook section 2B builds a two-teacher toy version.

Q6 CycleGAN's and DINO-WM's parts

Which statement correctly matches the main components of CycleGAN and DINO-WM?

A. CycleGAN needs paired images and one reconstruction decoder; DINO-WM predicts human action labels from text.

B. CycleGAN uses one generator with no discriminator; DINO-WM plans by comparing raw class labels.

C. CycleGAN clusters both domains and predicts cluster IDs; DINO-WM reconstructs every future pixel with a GAN.

D. CycleGAN uses two generators and two domain discriminators; DINO-WM predicts future visual features and searches for actions that approach an encoded goal.

Answer: D — two translators and two judges; predict features and search actions.

The idea. CycleGAN: $G : X \to Y$ and $F : Y \to X$; D_Y judges "does $G(x)$ look like domain Y?", D_X the reverse. DINO-WM: frozen DINOv2 features z, a learned predictor $z_{t+1} = f(z_t, a_t)$, and a planner that tries action sequences and keeps the one whose predicted final z is closest to the goal image's z (§13.3).

Q7 Confirmation bias

Why can indiscriminately accepting every pseudo-label degrade self-training?

A. Unlabelled inputs cannot be processed by neural networks.

B. Incorrect teacher predictions can be treated as truth and reinforced by the student.

C. Cross-entropy ignores pseudo-labelled examples by definition.

D. A student trained on more data must always have higher test accuracy.

Answer: B — the model learns its own mistakes as facts.

The idea. A wrong pseudo-label goes into the training set as if it were true. Training on it makes the model *more* confident in the same mistake, which makes the next round's pseudo-label even more certain. The notebook measures the size of the problem: on the whole pool only 68% of the teacher's labels are right (§23.4). Confidence filters reduce it; they cannot remove it.

Q8 What argmax throws away

A teacher predicts $[0.52, 0.45, 0.03]$. What is lost when this becomes a hard pseudo-label?

A. The number of parameters in the teacher.
B. The identity of the input and its augmentation history.
C. Whether the model was trained with gradient descent.
D. The teacher's uncertainty and the near-equal probabilities it assigns to the two leading classes.

Answer: D — "almost a tie" becomes "certain".

The idea. After argmax, $[0.52, 0.45, 0.03]$ and $[0.99, 0.005, 0.005]$ both become $[1, 0, 0]$. The student can no longer know that the first case was a coin toss between classes 1 and 2 and that class 3 was never in question. The notebook plots exactly this (Figure 23.3).

Q9 A student that beats its teacher

How can a self-trained or distilled student outperform its teacher?

A. It can learn from more data, stronger perturbations, iterative refinement, or complementary information from several teachers.
B. It is mathematically guaranteed after one gradient step.
C. It ignores the teacher and memorises the test labels.
D. It converts every uncertain teacher prediction into a correct label.

Answer: A — the teacher is a source of signal, not a ceiling.

The idea. The student sees more images than the teacher was trained on, under harder conditions (noise), sometimes with more capacity, and sometimes combining several teachers whose errors differ. Each of these can add information that no single teacher had. Nothing guarantees it; in the notebook the student gains only one point ($0.664 \to 0.673$).

Q10 Data the teacher's own loss could not use

Some images do not satisfy the assumptions of a teacher's original self-supervised loss. Why can they still help during distillation?

A. They become paired with manual labels automatically.

B. The student can skip them while still counting them as data.

C. The trained teacher can turn them into feature or distribution targets that the student matches with a compatible loss.

D. Distillation guarantees every teacher prediction on out-of-domain data is correct.

Answer: C — distillation only needs the teacher's *output*.

The idea. Example: a 3D teacher was trained on image *pairs* with known geometry; a single internet photo cannot be used by that loss. But the trained teacher can still be run on the single photo, and its output (a depth map, a feature map) is a target the student can match. The common interface is "teacher output in, student matches it", whatever the teacher's original training required.

Q11 Two losses in CycleGAN

Why does CycleGAN combine adversarial losses with a cycle-consistency loss?

A. Adversarial losses preserve each source instance exactly, while cycle consistency only selects the target domain.

B. Adversarial losses encourage target-domain realism, while cycle consistency encourages a translation to keep the information needed to return to the source.

C. Both require paired examples and serve the same purpose.

D. Cycle consistency trains only the discriminators, adversarial loss only an encoder.

Answer: B — realism from one, faithfulness from the other.

The idea. The adversarial loss judges a *set*: "does this look like a zebra photo?". It does not care *which* zebra photo; turning every horse into the same zebra would satisfy it. The cycle loss judges one *instance*: "turn it back, is it my horse again?" ($F(G(x)) \approx x$). Together: look like the target domain, keep the content. The notebook shows each loss alone (§23.4).

In the book. §12.4.

Q12 When a world model is self-supervised

When can next-state world-model training be considered self-supervised?

A. Only if every future state is manually assigned a semantic class.

B. Never, because sequential data contains hidden labels.

C. Always, even when a human provides action labels and desired outcomes for every transition.

D. Future observations provide automatic targets, while externally supplied action–state pairs add supervision.

Answer: D — the next frame is free; who supplied the actions matters.

The idea. A recording supplies its own targets: the frame at $t + 1$ is the "label" for the prediction made at t. Action-conditioning needs the action a_t; if the robot logged its own motor commands, that is still free; if a person annotated which action happened, it is extra supervision. Name the source of each signal (§13.2.3).

23.2 The quiz: case studies 13–18

Q13 80 labelled defects, 20,000 unlabelled images

You have 80 labelled defect images and 20,000 unlabelled factory images. An initial classifier is useful but makes confident mistakes on rare defects. Which self-training plan is most defensible?

A. Keep high-confidence pseudo-labels, monitor accepted class balance and errors, train the student with perturbations, and retain the true labelled loss.

B. Treat every pseudo-label as equally reliable and remove the original labelled examples.

C. Assign all unlabelled images to the majority class.

D. Evaluate only the pseudo-label training loss.

Answer: A — filter, monitor per class, perturb, keep the real labels.

The idea. Each part of A answers one danger. *Confidence filter*: drop the teacher's least certain guesses. *Monitor class balance*: a global threshold can silently reject whole classes — in the notebook it accepted *no* pullovers, coats or shirts at all (Figure 23.2, right). For rare defects that would mean the student never learns them. *Perturbations*: Noisy-Student-style robustness. *Keep the true labelled loss*: the 80 real labels anchor training and are the only honest test.

Q14 Compressing a teacher for a robot

A large perception model is accurate but too slow for an embedded robot. You want a smaller student that preserves as much of its behaviour as possible. Which training target is most appropriate?

A. Random one-hot labels, so the student does not copy teacher mistakes.

B. Only the teacher's final argmax.

C. The teacher's soft output distribution, optionally combined with available ground-truth labels.

D. Pixel reconstruction from the robot camera.

Answer: C — soft targets, plus real labels where you have them.

The idea. The goal is to copy the teacher's *behaviour*, including how it ranks the alternatives, so match its full distribution (Question 4). The notebook's section 2A does exactly this with a student of half the width: loss $= 0.75\times$ soft KL $+ 0.25\times$ real-label cross-entropy.

Q15 Three specialised teachers, one encoder

You have a 2D semantic teacher, a human-pose teacher and a 3D-geometry teacher trained with different objectives. You want one broadly useful encoder. Which strategy matches multi-teacher distillation?

A. Average all raw teacher tensors directly, even though their shapes and meanings differ.

B. Use a shared student with teacher-specific projections or heads and optimise the compatible losses across the broad data set.

C. Train one student per teacher, with no shared representation.

D. Convert every output to a single hard object class.

Answer: B — shared body, one head per teacher.

The idea. A 10-number class vector and a 7×7 edge map cannot be averaged — they are not even the same shape. Give the student one head per teacher that produces that teacher's kind of output, match each head with the loss that suits it, and let all gradients flow into the shared encoder. Notebook section 2B: a class head (KL loss) and an edge head (L1 loss) on one encoder.

Q16 Synthetic to real, with no paired scenes

A team has synthetic and real warehouse images, but no scene appears in both. They want to translate synthetic images into a realistic style while preserving content. Which design is most appropriate?

A. A supervised pixel-wise regressor trained on invented pairs.
B. One discriminator for both domains, with no generator or reconstruction.
C. A classifier trained only to tell synthetic from real.
D. Two domain translators with domain-specific discriminators and a cycle-consistency loss.

Answer: D — CycleGAN.

The idea. No pairs rules out pixel losses against a target (A). A discriminator or classifier alone (B, C) can tell domains apart but cannot translate anything. The unpaired recipe is two generators, a judge per domain, and the round trip (Question 11). Notebook section 3 builds it on Fashion-MNIST with "real" images and inverted images as the two domains.

Q17 Realistic outputs, failed round trips

A CycleGAN's outputs look realistic in the target domain, but translating them back does not recover the original layout. What is the most direct diagnosis?

A. The adversarial objective is satisfied, but the cycle-consistency constraint is too weak, missing, or wrongly implemented.
B. The discriminators are too weak, because realistic outputs prove they cannot learn.
C. The model has solved unpaired translation, since only realism matters.
D. Remove the reverse generator so round trips are no longer evaluated.

Answer: A — look at the cycle term.

The idea. Each loss is responsible for one symptom. Realism OK means the adversarial part works. Content lost means the part responsible for content — the cycle loss — is too weak (its weight λ too small), missing, or buggy (e.g. comparing with the wrong image). The notebook reproduces this failure by setting $\lambda = 0$ (Figure 23.5).

Q18 Planning with a latent world model

A robot has a frozen visual encoder, recorded transitions (observation, action, next observation), and an image of a goal. Which pipeline describes latent world-model planning?

A. Fine-tune the encoder on the goal image, then choose a random action.

B. Predict the next action from the current class label without modelling transitions.

C. Train action-conditioned latent dynamics to predict the next encoded observation, then search action sequences whose predicted final latent is closest to the encoded goal.

D. Translate the image into another domain and use discriminator confidence as the action.

Answer: C — learn $z_{t+1} = f(z_t, a_t)$, then search.

The idea. Encode, predict, search, execute. The notebook's section 4 does it on CartPole: learn the dynamics from 5,000 random transitions, try all $2^{10} = 1024$ ten-step plans inside the model, pick the one ending closest to "upright and centred", and run it in the real simulator.

23.3 The case-study notebook, cell by cell

The notebook uses *Fashion-MNIST* (28×28 grey images of ten kinds of clothing; Figure 23.1) and *CartPole* (a cart that must keep a pole upright). Every code cell follows in full; long cells are split at the notebook's own section comments, with the original line numbers kept.

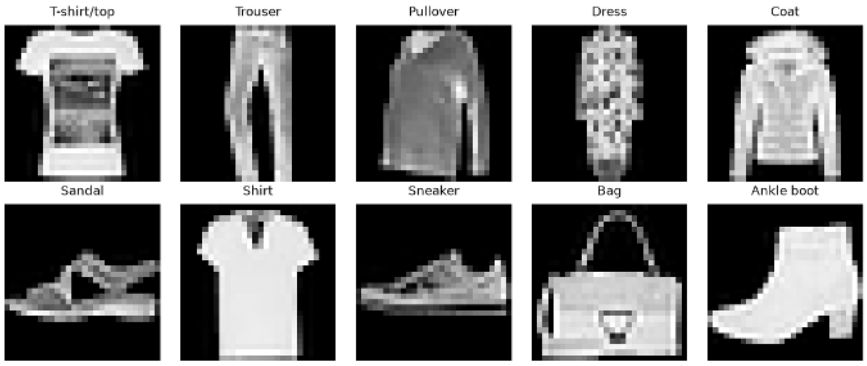


Figure 23.1. One labelled example of each Fashion-MNIST class (output of the notebook's first cell).

How to read it: Note the look-alikes: T-shirt, pullover, coat and shirt differ in small details. They will cause most of the trouble below.

23.3.1 Setup: data, the small CNN and helpers

```python
# ------------------------------------------------------------------------------
# Setup: reproducibility, device selection, data utilities, and a small CNN
# ------------------------------------------------------------------------------
# The first run downloads Fashion-MNIST. Later Colab runs reuse the cached files.
# Gymnasium is installed only when the runtime does not already provide it.
import itertools
import os
import random
import subprocess
import sys
import time

os.environ.setdefault("SDL_VIDEODRIVER", "dummy")
os.environ.setdefault("PYGAME_HIDE_SUPPORT_PROMPT", "1")
try:
    import gymnasium as gym
except ImportError:
    subprocess.check_call(
        [sys.executable, "-m", "pip", "install", "-q",
    ↪ "gymnasium[classic-control]==1.1.1"]
    )
    import gymnasium as gym

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as T
from torch.utils.data import DataLoader, Dataset, TensorDataset

# Fix every random-number generator used below. CUDA kernels can still introduce
# tiny numerical differences, so students should compare trends, not last digits.
SEED = 9
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
gpu_name = torch.cuda.get_device_name(0) if DEVICE.type == "cuda" else ""
print("Device:", DEVICE, gpu_name)
```

Listing 23.1. Week-9 cell 1, part 1: imports, seeds, and three image transforms.

Three transforms, from gentle to harsh: `clean_tf` only converts to a tensor (used whenever a teacher makes targets or a model is evaluated); `weak_tf` adds a random crop with 2 pixels of padding (small shifts); `strong_tf` adds a horizontal flip and random erasing (a grey rectangle over up to 12% of the image). "Weak" and "strong" are the Noisy-Student idea: the teacher sees clean images, the student noisy ones.

```

def loader(dataset, batch=128, shuffle=True):
    """Create the same seeded DataLoader in every experiment."""
    return DataLoader(
        dataset,
        batch_size=batch,
        shuffle=shuffle,
        num_workers=2,
        pin_memory=DEVICE.type == "cuda",
        generator=torch.Generator().manual_seed(SEED),
    )


# Clean images are used when a teacher creates targets or a model is evaluated.
# Weak/strong transforms let students see perturbed versions of the same content.
clean_tf = T.Compose([T.ToPILImage(), T.ToTensor()])
weak_tf = T.Compose(
    [T.ToPILImage(), T.RandomCrop(28, padding=2), T.ToTensor()]
)
strong_tf = T.Compose(
    [
        T.ToPILImage(),
        T.RandomCrop(28, padding=2),
        T.RandomHorizontalFlip(),
        T.ToTensor(),
        T.RandomErasing(p=0.2, scale=(0.02, 0.12)),
    ]
)


class Images(Dataset):
    """Wrap raw uint8 images with targets, transforms, and per-example weights."""

    def __init__(self, data, targets, transform=clean_tf, weights=None):
        self.data = data
        self.targets = torch.as_tensor(targets)
        self.transform = transform
        self.weights = (
            torch.ones(len(data))
            if weights is None
            else torch.as_tensor(weights).float()
        )

    def __len__(self):
        return len(self.data)

    def __getitem__(self, i):
        return self.transform(self.data[i]), self.targets[i], self.weights[i]


class FashionCNN(nn.Module):
    """A compact classifier whose width can be changed for teacher/student roles."""

    def __init__(self, width=16, classes=10):
        super().__init__()
```

```
        # Two convolution/pooling stages map 28x28 pixels to a 7x7 feature grid.
        self.encoder = nn.Sequential(
            nn.Conv2d(1, width, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(width, 2 * width, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.head = nn.Linear(2 * width * 7 * 7, classes)

    def features(self, x):
        return self.encoder(x).flatten(1)

    def forward(self, x):
        return self.head(self.features(x))


def train_classifier(model, dataset, epochs, lr=2e-3):
    """Train a classifier; w controls how much each target contributes."""
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=2e-4)
    for _ in range(epochs):
        model.train()
        for x, y, w in loader(dataset):
            x, y, w = x.to(DEVICE), y.to(DEVICE), w.to(DEVICE)
            # Compute one loss per image before applying genuine/pseudo-label weights.
            per_image_loss = F.cross_entropy(model(x), y, reduction="none")
            loss = (per_image_loss * w).sum() / w.sum()
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return model


@torch.no_grad()
def logits(model, data, batch=256):
    """Run clean-image inference and return CPU logits for later target creation."""
    model.eval()
    outputs = []
    dummy_targets = torch.zeros(len(data))
    for x, _, _ in loader(Images(data, dummy_targets), batch, shuffle=False):
        outputs.append(model(x.to(DEVICE)).cpu())
    return torch.cat(outputs)
```

Listing 23.2. Week-9 cell 1, part 2: loader, dataset wrapper, the CNN, training and inference helpers.

- `Images` returns image, target *and a weight* per example. The weight lets real labels count 1.0 and pseudo-labels 0.5 later.
- `FashionCNN`: two conv–ReLU–maxpool blocks (28 → 14 → 7), channels `width` and 2×`width`, then one linear layer to 10 classes. Changing `width` makes teachers and students of different size (16, 20, 8, 10 below).

- `train_classifier`: cross-entropy per image (`reduction="none"`), multiplied by the weights, divided by the weight sum — a weighted average.
- `logits`: run a model over images (clean transform, no gradients) and return raw scores for later use as targets.

```
# ------------------------------------------------------------------------------
# Make a balanced labeled/unlabeled split from real Fashion-MNIST images.
# ------------------------------------------------------------------------------
DATA_ROOT = os.environ.get(
    "WEEK09_DATA_ROOT",
    "/content/week09_data" if os.path.exists("/content") else "./week09_data",
)
train_set = torchvision.datasets.FashionMNIST(DATA_ROOT, train=True, download=True)
test_set = torchvision.datasets.FashionMNIST(DATA_ROOT, train=False, download=True)
names = train_set.classes
targets = train_set.targets.numpy()
rng = np.random.default_rng(SEED)

# Take 100 labeled and 300 nominally unlabeled examples from every class.
# Balancing the source pool makes any later pseudo-label imbalance easy to see.
labeled_ids, pool_ids = [], []
for class_id in range(10):
    class_ids = np.where(targets == class_id)[0]
    rng.shuffle(class_ids)
    labeled_ids.extend(class_ids[:100])
    pool_ids.extend(class_ids[100:400])

labeled_ids = np.asarray(labeled_ids)
pool_ids = np.asarray(pool_ids)
x_lab = train_set.data[labeled_ids]
y_lab = train_set.targets[labeled_ids]
x_pool = train_set.data[pool_ids]
y_pool = train_set.targets[pool_ids]  # Hidden from training; used only for audit plots.
x_test = test_set.data[:3000]
y_test = test_set.targets[:3000]
print(
    f"Fashion-MNIST: {len(x_lab)} labeled, {len(x_pool)} unlabeled, "
    f"{len(x_test)} test"
)

# Show one labeled example from every class. Selecting by label avoids relying
# on the storage order of the balanced split.
sample_ids = [int(torch.where(y_lab == class_id)[0][0]) for class_id in range(10)]
sample = torch.stack([clean_tf(x_lab[i]) for i in sample_ids])
fig, axes = plt.subplots(2, 5, figsize=(8, 3.4))
for i, axis in enumerate(axes.ravel()):
    axis.imshow(sample[i, 0], cmap="gray")
    axis.set_title(names[int(y_lab[sample_ids[i]])], fontsize=8)
    axis.axis("off")
plt.tight_layout()
plt.show()
```

Listing 23.3. Week-9 cell 1, part 3: a balanced labelled / unlabelled split.

From each class: 100 images with labels (1,000 labelled in total) and the next 300 as the "unlabelled" pool (3,000). The pool's true labels `y_pool` are kept only to *audit* pseudo-labels afterwards; the comment says so explicitly. 3,000 test images measure accuracy.

23.3.2 Section 1: self-training with a confidence filter (Q13)

```
# ------------------------------------------------------------------------
# Stage 1: train the initial teacher using genuine labels only.
# ------------------------------------------------------------------------
start_time = time.time()
teacher = FashionCNN(width=16).to(DEVICE)
train_classifier(teacher, Images(x_lab, y_lab, weak_tf), epochs=5)

# ------------------------------------------------------------------------
# Stage 2: convert teacher predictions on the unlabeled pool into candidate labels.
# ------------------------------------------------------------------------
# softmax turns ten logits into probabilities. The largest probability is used as
# confidence; its class index becomes the hard pseudo-label.
pool_logits = logits(teacher, x_pool)
probs = pool_logits.softmax(dim=1)
confidence, pseudo = probs.max(dim=1)

# Keep the highest-confidence 40%. This is a ranking rule: it does not assume that
# a value such as 0.90 is perfectly calibrated across classes.
threshold = float(torch.quantile(confidence, 0.60))
keep = confidence >= threshold

# y_pool is hidden during training. We consult it only to audit how reliable the
# teacher targets were; a real unlabeled deployment would need a labeled audit set.
correct = pseudo.eq(y_pool)

# ------------------------------------------------------------------------
# Stage 3: build the student's mixed training set.
# ------------------------------------------------------------------------
# Genuine labels are retained. Accepted pseudo-labels are added with half weight,
# acknowledging that teacher-generated targets are less trustworthy.
combined_x = torch.cat([x_lab, x_pool[keep]])
combined_y = torch.cat([y_lab, pseudo[keep]])
combined_w = torch.cat(
    [torch.ones(len(y_lab)), torch.full((int(keep.sum()),), 0.5)]
)

# ------------------------------------------------------------------------
# Stage 4: train a fresh student under stronger image perturbations.
# ------------------------------------------------------------------------
# The wider network and stronger augmentation make this a Noisy-Student-style
# single pass. Full Noisy Student would repeat teacher/student promotion in rounds.
torch.manual_seed(SEED + 1)
student = FashionCNN(width=20).to(DEVICE)
train_classifier(
    student,
    Images(combined_x, combined_y, strong_tf, combined_w),
    epochs=3,
```

```
)
```

Listing 23.4. Week-9 cell 2, part 1: teacher, pseudo-labels, filter, student.

Step by step — one round of Noisy-Student-style self-training

Step 1. **Teacher** (lines 5–6): width 16, 5 epochs on the 1,000 real labels with weak augmentation.

Step 2. **Pseudo-label** (lines 13–15): softmax over the teacher's scores on the clean pool; `confidence` is the top probability, `pseudo` its class.

Step 3. **Filter** (lines 19–20): the 60th percentile of confidence as threshold, i.e. keep the most confident 40% — a *ranking* rule, which does not assume the probabilities are calibrated.

Step 4. **Mixed set** (lines 31–35): real images with weight 1, accepted pseudo-labelled images with weight 0.5.

Step 5. **Student** (lines 42–48): a fresh, slightly wider network (width 20) trained 3 epochs with the *strong* transform.

```
# ------------------------------------------------------------------------------
# Stage 5: evaluate target quality and downstream behavior.
# ------------------------------------------------------------------------------
all_reliability = correct.float().mean().item()
kept_reliability = correct[keep].float().mean().item()
teacher_accuracy = (logits(teacher, x_test).argmax(1) == y_test).float().mean().item()
student_accuracy = (logits(student, x_test).argmax(1) == y_test).float().mean().item()

# Count accepted *predicted* labels. Unlike reliability, this diagnostic does not
# require opening the pool's hidden ground truth and is therefore deployable.
accepted_counts = torch.bincount(pseudo[keep], minlength=10)
accepted_share = accepted_counts.float() / accepted_counts.sum()

print(
    f"Pseudo-label reliability: all {all_reliability:.3f}, "
    f"accepted {kept_reliability:.3f}"
)
print(
    f"Test accuracy: teacher {teacher_accuracy:.3f}, "
    f"self-trained student {student_accuracy:.3f}"
)
print(f"Accepted pseudo-labels: {int(keep.sum())} / {len(x_pool)}")
print(f"Self-training time: {time.time() - start_time:.1f}s")

# The four panels answer different questions: confidence separation, target
# reliability, downstream accuracy, and class balance among accepted targets.
fig, axes = plt.subplots(1, 4, figsize=(15.5, 3.3))
axes[0].hist(confidence[correct], bins=14, alpha=0.7, label="correct")
axes[0].hist(confidence[~correct], bins=14, alpha=0.7, label="wrong")
axes[0].axvline(threshold, color="k", linestyle="--", label="threshold")
axes[0].legend(fontsize=8)
```

```
axes[0].set_title("Teacher confidence")

axes[1].bar(
    ["all", "accepted"],
    [all_reliability, kept_reliability],
)
axes[1].set_ylim(0, 1)
axes[1].set_title("Pseudo-label reliability")

axes[2].bar(
    ["teacher", "student"],
    [teacher_accuracy, student_accuracy],
)
axes[2].set_ylim(0, 1)
axes[2].set_title("Test accuracy")

axes[3].bar(range(10), accepted_share.numpy())
axes[3].axhline(0.1, color="k", linestyle="--", linewidth=1, label="balanced share")
axes[3].set_xticks(range(10), names, rotation=60, fontsize=7)
axes[3].set_ylabel("fraction accepted")
axes[3].set_title("Accepted predicted classes")
axes[3].legend(fontsize=7)

plt.tight_layout()
plt.show()

# This assertion guards the pedagogical comparison against accidental regressions.
assert kept_reliability >= all_reliability
```

Listing 23.5. Week-9 cell 2, part 2: measure and plot.

Four measurements, each answering a different question: reliability of all pseudo-labels versus accepted ones (how good is the filter?), teacher versus student test accuracy (did it help?), and the share of each *predicted* class among the accepted labels. The last one needs no ground truth, so it can be monitored in a real deployment. The final `assert` guards the lesson: the filter must not make things worse.

23.3.3 Section 2: soft targets and heterogeneous teachers (Q14, Q15)

```
# ---------------------------------------------------------------------------
# 2A. Soft-target distillation: compress a teacher into a narrower student.
# ---------------------------------------------------------------------------
# Cache teacher logits once. Logits are kept rather than argmax labels so the
# student can see the teacher's full ranking of all ten clothing classes.
all_x = torch.cat([x_lab, x_pool])
teacher_targets = torch.cat([logits(teacher, x_lab), pool_logits])


class Distill(Dataset):
    """Return an image, teacher logits, optional real label, and label mask."""

    def __len__(self):
```

```
        return len(all_x)

    def __getitem__(self, i):
        has_real_label = i < len(y_lab)
        # A placeholder label is needed for unlabeled examples, but the mask below
        # prevents that placeholder from entering the supervised loss.
        real_label = y_lab[i] if has_real_label else torch.tensor(0)
        return weak_tf(all_x[i]), teacher_targets[i], real_label, has_real_label


# The student has half the channel width of the teacher. Temperature T=2 softens
# both distributions so non-winning classes contribute useful gradients.
compact = FashionCNN(width=8).to(DEVICE)
optimizer = torch.optim.AdamW(compact.parameters(), lr=2e-3)
temperature = 2.0

for _ in range(3):
    compact.train()
    for x, target_logits, y, has_real_label in loader(Distill()):
        x = x.to(DEVICE)
        target_logits = target_logits.to(DEVICE)
        y = y.to(DEVICE)
        has_real_label = has_real_label.to(DEVICE).bool()
        student_logits = compact(x)

        # KL(student || teacher target in PyTorch's input/target convention):
        # input is log student probability and target is teacher probability.
        soft_loss = F.kl_div(
            F.log_softmax(student_logits / temperature, dim=1),
            F.softmax(target_logits / temperature, dim=1),
            reduction="batchmean",
        ) * temperature**2
        # T^2 compensates for the smaller gradients created by softened logits.

        # Only genuinely labeled examples contribute to cross-entropy.
        hard_loss = (
            F.cross_entropy(student_logits[has_real_label], y[has_real_label])
            if has_real_label.any()
            else student_logits.sum() * 0.0
        )
        loss = 0.75 * soft_loss + 0.25 * hard_loss

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Test accuracy measures truth; teacher agreement measures behavioral transfer.
compact_test_logits = logits(compact, x_test)
teacher_test_logits = logits(teacher, x_test)
compact_accuracy = (
    (compact_test_logits.argmax(1) == y_test).float().mean().item()
)
teacher_agreement = (
    (compact_test_logits.argmax(1) == teacher_test_logits.argmax(1))
    .float()
    .mean()
```

```
    .item()
)
print(f"Compact-student test accuracy: {compact_accuracy:.3f}")
print(f"Compact-student agreement with teacher: {teacher_agreement:.3f}")

# Select the least-confident pool prediction to make the discarded alternatives
# visible. The hard target has one nonzero bar; the soft target retains all ten.
ambiguous = int(torch.argmin(confidence))
soft_distribution = probs[ambiguous]
hard_distribution = F.one_hot(pseudo[ambiguous], num_classes=10)
plt.figure(figsize=(8, 2.6))
plt.bar(
    np.arange(10) - 0.18,
    soft_distribution,
    width=0.36,
    label="soft distribution",
)
plt.bar(
    np.arange(10) + 0.18,
    hard_distribution,
    width=0.36,
    label="hard argmax",
)
plt.xticks(range(10), names, rotation=35, fontsize=8)
plt.ylabel("probability")
plt.legend()
plt.title("Argmax discards alternatives")
plt.show()
```

Listing 23.6. Week-9 cell 3, part 1 (2A): soft-target distillation into a half-width student.

The student (width 8, half the teacher) trains on all 4,000 images. Its loss (lines 40–54) has two parts:

$$\mathcal{L} = 0.75 \cdot T^2 \,\mathrm{KL}\big(\mathrm{softmax}(t/T) \,\|\, \mathrm{softmax}(s/T)\big) + 0.25 \cdot \mathrm{CE}(s, y) \quad \text{(real labels only)},$$

with teacher scores t, student scores s and temperature $T = 2$. Dividing by T *softens* both distributions, so small probabilities become visible and give gradient. *Numbers:* scores $(4, 2, 0)$ give probabilities $(0.87, 0.12, 0.02)$ at $T = 1$, and $(0.67, 0.24, 0.09)$ at $T = 2$. Softening shrinks the gradients by about $1/T^2$, hence the T^2 factor. The mask `has_real_label` makes sure the placeholder label of unlabelled images never enters the cross-entropy. The last part plots the pool image the teacher was least sure about.

```
# ------------------------------------------------------------------------------
# 2B. Heterogeneous-teacher analogy: one encoder, target-specific output heads.
# ------------------------------------------------------------------------------
# The semantic teacher outputs a 10-vector. The frozen Sobel operator below acts
# as a second teacher and outputs a spatial 1x7x7 edge map without class labels.
sobel_x = torch.tensor(
    [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]],
    dtype=torch.float32,
    device=DEVICE,
```

```
).view(1, 1, 3, 3)


def edge_teacher(x):
    """Return a downsampled edge-magnitude target for each input image."""
    grad_x = F.conv2d(x, sobel_x, padding=1)
    grad_y = F.conv2d(x, sobel_x.transpose(-1, -2), padding=1)
    magnitude = torch.sqrt(grad_x**2 + grad_y**2 + 1e-8) / 4
    return F.adaptive_avg_pool2d(magnitude, 7).clamp(0, 1)


class MultiHead(nn.Module):
    """A shared visual encoder with one compatible head per teacher target."""

    def __init__(self):
        super().__init__()
        self.encoder = FashionCNN(width=10).encoder
        self.semantic_head = nn.Linear(20 * 7 * 7, 10)
        self.edge_head = nn.Conv2d(20, 1, 1)

    def forward(self, x):
        features = self.encoder(x)
        semantic = self.semantic_head(features.flatten(1))
        edges = torch.sigmoid(self.edge_head(features))
        return semantic, edges


multi = MultiHead().to(DEVICE)
optimizer = torch.optim.AdamW(multi.parameters(), lr=2e-3)
teacher.eval()

# Use the first 2,000 pool images for training and reserve the final 1,000 for
# evaluation. Keeping them disjoint prevents an in-sample reconstruction report.
train_subset = Images(x_pool[:2000], torch.zeros(2000))
eval_subset = Images(x_pool[2000:3000], torch.zeros(1000))

for _ in range(2):
    multi.train()
    for x, _, _ in loader(train_subset):
        x = x.to(DEVICE)
        # Frozen teachers create targets; no gradient should enter either teacher.
        with torch.no_grad():
            semantic_target = F.softmax(teacher(x) / 2.0, dim=1)
            edge_target = edge_teacher(x)

        semantic_prediction, edge_prediction = multi(x)
        semantic_loss = F.kl_div(
            F.log_softmax(semantic_prediction / 2.0, dim=1),
            semantic_target,
            reduction="batchmean",
        ) * 2.0**2
        edge_loss = F.l1_loss(edge_prediction, edge_target)

        # Both losses backpropagate through their head into the same encoder.
        loss = semantic_loss + 2.0 * edge_loss
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

# Evaluate each head with its own metric on the disjoint clean-image slice.
multi.eval()
with torch.no_grad():
    eval_x = next(iter(loader(eval_subset, batch=256, shuffle=False)))[0].to(DEVICE)
    semantic_prediction, edge_prediction = multi(eval_x)
    semantic_target_class = teacher(eval_x).argmax(1)
    edge_target = edge_teacher(eval_x)
    semantic_agreement = (
        (semantic_prediction.argmax(1) == semantic_target_class)
        .float()
        .mean()
        .item()
    )
    edge_mae = F.l1_loss(edge_prediction, edge_target).item()
    show_images = eval_x[:6]
    show_edge_target = edge_target[:6]
    show_edge_prediction = edge_prediction[:6]

print("Semantic teacher target shape:", (3, 10))
print("Edge teacher target shape:", tuple(show_edge_target[:3].shape))
print(f"Held-out semantic-head agreement with teacher: {semantic_agreement:.3f}")
print(f"Held-out edge-head mean absolute error: {edge_mae:.3f}")

# Rows show the input, the frozen edge teacher's target, and the student's output.
fig, axes = plt.subplots(3, 6, figsize=(9, 4.3))
for i in range(6):
    axes[0, i].imshow(show_images[i, 0].cpu(), cmap="gray")
    axes[1, i].imshow(show_edge_target[i, 0].cpu(), cmap="magma")
    axes[2, i].imshow(show_edge_prediction[i, 0].cpu(), cmap="magma")
    for row in range(3):
        axes[row, i].axis("off")
axes[0, 0].set_title(
    "photo / teacher edge / student edge", loc="left", fontsize=8
)
plt.tight_layout()
plt.show()
```

Listing 23.7. Week-9 cell 3, part 2 (2B): one encoder, two teachers, two heads.

The second teacher is not learned at all: a *Sobel filter*, the classic edge detector, whose output is shrunk to a 7×7 edge map. The student `MultiHead` has one encoder (the two conv blocks, width 10) and two heads: a linear layer to 10 classes (matched to the soft semantic teacher by KL) and a 1×1 convolution to a 7×7 map with a sigmoid (matched to the Sobel teacher by L1). Both losses flow into the same encoder (line 163; the edge loss is weighted by 2). Evaluation uses pool images 2,000–2,999, which the student never trained on, and scores each head with its own metric.

23.3.4 Section 3: unpaired translation and cycle consistency (Q16, Q17)

```python
# ---------------------------------------------------------------------------
# Build two unpaired image domains.
# ---------------------------------------------------------------------------
# X contains ordinary Fashion-MNIST images. Y contains inverted images from a
# different shuffled index. The model therefore never sees an (x, matching y) pair.
cycle_data = x_pool[:2000]
permutation = np.random.default_rng(SEED + 5).permutation(len(cycle_data))


class Unpaired(Dataset):
    def __len__(self):
        return len(cycle_data)

    def __getitem__(self, i):
        source_x = clean_tf(cycle_data[i])
        unrelated_y = 1 - clean_tf(cycle_data[permutation[i]])
        return source_x, unrelated_y


class Generator(nn.Module):
    """A tiny fully convolutional image-to-image translator."""

    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 16, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 1, 3, padding=1),
            nn.Tanh(),
        )

    def forward(self, x):
        return self.network(x)


class Discriminator(nn.Module):
    """Return a spatial real/fake score map for one image domain."""

    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, 16, 4, 2, 1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(16, 32, 4, 2, 1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(32, 1, 3, padding=1),
        )

    def forward(self, x):
        return self.network(x)
```

Listing 23.8. Week-9 cell 4, part 1: two unpaired domains, a generator and a discriminator.

Domain X: ordinary Fashion-MNIST (light garment on black). Domain Y: *inverted* images (dark garment on white), taken from a shuffled index, so the x and y of a pair are different garments — genuinely unpaired. The generator is three 3×3 convolutions ending in tanh (outputs in $[-1, 1]$). The discriminator outputs a small *map* of real/fake scores, one per image region (a PatchGAN-style judge).

```
def train_cycle(cycle_weight):
    """Train one miniature CycleGAN condition with a chosen cycle weight."""
    torch.manual_seed(SEED + 6)

    # G maps X->Y, F_reverse maps Y->X. Each domain has its own discriminator.
    G = Generator().to(DEVICE)
    F_reverse = Generator().to(DEVICE)
    D_X = Discriminator().to(DEVICE)
    D_Y = Discriminator().to(DEVICE)

    generator_optimizer = torch.optim.Adam(
        itertools.chain(G.parameters(), F_reverse.parameters()),
        lr=3e-4,
        betas=(0.5, 0.999),
    )
    discriminator_optimizer = torch.optim.Adam(
        itertools.chain(D_X.parameters(), D_Y.parameters()),
        lr=3e-4,
        betas=(0.5, 0.999),
    )

    for _ in range(5):
        for x, y in loader(Unpaired(), batch=128):
            # Tanh generators operate naturally on the [-1, 1] image range.
            x = (2 * x - 1).to(DEVICE)
            y = (2 * y - 1).to(DEVICE)

            # ----- Discriminator step -----
            # Detached translations are treated as fake samples. Least-squares GAN
            # targets use 1 for real inputs and 0 for generated inputs.
            with torch.no_grad():
                fake_y = G(x)
                fake_x = F_reverse(y)
            d_x_real, d_x_fake = D_X(x), D_X(fake_x)
            d_y_real, d_y_fake = D_Y(y), D_Y(fake_y)
            discriminator_loss = 0.5 * (
                F.mse_loss(d_x_real, torch.ones_like(d_x_real))
                + F.mse_loss(d_x_fake, torch.zeros_like(d_x_fake))
                + F.mse_loss(d_y_real, torch.ones_like(d_y_real))
                + F.mse_loss(d_y_fake, torch.zeros_like(d_y_fake))
            )
            discriminator_optimizer.zero_grad()
            discriminator_loss.backward()
            discriminator_optimizer.step()

            # ----- Generator step -----
            # Adversarial terms ask each translation to look real in its destination.
            fake_y = G(x)
```

```
            fake_x = F_reverse(y)
            score_fake_y = D_Y(fake_y)
            score_fake_x = D_X(fake_x)
            adversarial_loss = F.mse_loss(
                score_fake_y, torch.ones_like(score_fake_y)
            ) + F.mse_loss(score_fake_x, torch.ones_like(score_fake_x))

            # Cycle terms reconstruct each particular input after a round trip.
            reconstructed_x = F_reverse(fake_y)
            reconstructed_y = G(fake_x)
            cycle_loss = F.l1_loss(reconstructed_x, x) + F.l1_loss(
                reconstructed_y, y
            )
            generator_loss = adversarial_loss + cycle_weight * cycle_loss

            generator_optimizer.zero_grad()
            generator_loss.backward()
            generator_optimizer.step()

    return G.eval(), F_reverse.eval()
```

Listing 23.9. Week-9 cell 4, part 2: one CycleGAN training run.

Step by step — one CycleGAN step

Step 1. Discriminators (lines 82–98): real images should score 1, translated ones 0 (least-squares GAN loss). The translations are made under `no_grad`, so only the judges learn in this half-step.

Step 2. Generators, adversarial part (lines 100–108): translations should be scored 1 by the judge of their *destination* domain.

Step 3. Generators, cycle part (lines 110–115): $F(G(x))$ should equal x and $G(F(y))$ should equal y, measured by L1.

Step 4. Total (line 116): adversarial + $\lambda \times$ cycle. The function is run twice, with $\lambda = 0$ and $\lambda = 3$.

```

# Train an adversarial-only baseline and a cycle-constrained model from the same seed.
start_time = time.time()
G_no_cycle, F_no_cycle = train_cycle(cycle_weight=0)
G_cycle, F_cycle = train_cycle(cycle_weight=3)
print(f"CycleGAN time: {time.time() - start_time:.1f}s")


@torch.no_grad()
def check_round_trip(G, F_reverse, n=128):
    """Translate X->Y->X and return reconstruction error plus images."""
    source = torch.stack([clean_tf(cycle_data[i]) for i in range(n)]).to(DEVICE)
    translated = (G(2 * source - 1) + 1) / 2
    reconstructed = (F_reverse(2 * translated - 1) + 1) / 2
    error = F.l1_loss(reconstructed, source).item()
```

```
    return error, source.cpu(), translated.cpu(), reconstructed.cpu()


no_cycle_result = check_round_trip(G_no_cycle, F_no_cycle)
cycle_result = check_round_trip(G_cycle, F_cycle)
print(
    f"Round-trip L1: adversarial only {no_cycle_result[0]:.3f}, "
    f"with cycle {cycle_result[0]:.3f}"
)

# Guard the key intended result: explicitly training reconstruction should reduce
# the measured round-trip error for this fixed demonstration.
assert cycle_result[0] < no_cycle_result[0]

# The rows deliberately separate target-domain appearance from instance recovery.
real_y = torch.stack(
    [1 - clean_tf(cycle_data[permutation[i]]) for i in range(6)]
)
rows = [
    (no_cycle_result[1][:6], "source"),
    (real_y, "unpaired target"),
    (no_cycle_result[2][:6], "G(x), no cycle"),
    (no_cycle_result[3][:6], "round trip, no cycle"),
    (cycle_result[2][:6], "G(x), cycle"),
    (cycle_result[3][:6], "round trip, cycle"),
]

fig, axes = plt.subplots(6, 6, figsize=(8, 8))
for row_id, (images, label) in enumerate(rows):
    for column_id in range(6):
        axes[row_id, column_id].imshow(
            images[column_id, 0], cmap="gray", vmin=0, vmax=1
        )
        axes[row_id, column_id].axis("off")
    axes[row_id, 0].text(
        -0.12,
        0.5,
        label,
        transform=axes[row_id, 0].transAxes,
        ha="right",
        va="center",
        fontsize=8,
    )
plt.tight_layout()
plt.show()
```

Listing 23.10. Week-9 cell 4, part 3: compare the two runs.

`check_round_trip` translates 128 images $X \to Y \to X$ and reports the mean absolute pixel error. The six picture rows separate the two questions: rows 3 and 5 ("does it look like Y?") and rows 4 and 6 ("is it still the same garment?").

23.3.5 Section 4: an action-conditioned world model and planning (Q18)

```
# ---------------------------------------------------------------------------
# Stage 1: collect self-supervised transition targets from CartPole.
# ---------------------------------------------------------------------------
# A transition is (current observation, action, next observation). The environment
# supplies the next observation automatically; no person labels the system dynamics.
environment = gym.make("CartPole-v1")
rng = np.random.default_rng(SEED)
states, actions, next_states = [], [], []
state, _ = environment.reset(seed=SEED)

while len(states) < 5000:
    action = int(rng.integers(2))  # 0 pushes left; 1 pushes right.
    next_state, _, terminated, truncated, _ = environment.step(action)
    states.append(state.copy())
    actions.append(action)
    next_states.append(next_state.copy())
    state = next_state
    if terminated or truncated:
        state, _ = environment.reset()

environment.close()
states = torch.tensor(np.asarray(states)).float()
actions = torch.tensor(actions)
next_states = torch.tensor(np.asarray(next_states)).float()

# Randomly hold out 500 transitions. The model never optimizes their targets.
ids = torch.randperm(len(states), generator=torch.Generator().manual_seed(SEED))
train_ids, test_ids = ids[:4500], ids[4500:]

# ---------------------------------------------------------------------------
# Stage 2: define a frozen, transparent representation E(o).
# ---------------------------------------------------------------------------
# Fit normalization on training states only, then freeze it. This coordinate-style
# encoder is an explicit lightweight stand-in for a meaningful frozen DINO feature.
mean = states[train_ids].mean(dim=0)
std = states[train_ids].std(dim=0).clamp_min(1e-5)


def encode(observation):
    return (observation - mean) / std


class Dynamics(nn.Module):
    """Predict a residual change in four-dimensional latent state."""

    def __init__(self):
        super().__init__()
        # Four latent coordinates + two one-hot action entries = six inputs.
        self.network = nn.Sequential(
            nn.Linear(6, 48),
            nn.SiLU(),
            nn.Linear(48, 4),
        )
```

```
54
55      def forward(self, z, one_hot_action):
56          predicted_change = self.network(torch.cat([z, one_hot_action], dim=1))
57          return z + predicted_change
```

Listing 23.11. Week-9 cell 5, part 1: record transitions, a frozen encoder, the dynamics model.

5,000 transitions (o_t, a_t, o_{t+1}) from CartPole with *random* actions; the simulator supplies o_{t+1}, so no one labels anything. An observation has four numbers: cart position, cart velocity, pole angle, pole angular velocity. The "encoder" E is deliberately simple: standardise each number with the training mean and standard deviation, then freeze — a transparent stand-in for DINO-WM's frozen DINO features. `Dynamics` takes z (4 numbers) and the one-hot action (2 numbers) and predicts the *change* of z; returning $z + \Delta z$ (a *residual* model) makes "nothing much changes in one step" the easy default.

```
60  # ------------------------------------------------------------------------
61  # Stage 3: learn z_t, a_t -> z_{t+1} from recorded transitions.
62  # ------------------------------------------------------------------------
63  model = Dynamics().to(DEVICE)
64  optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3)
65  training_history = []
66  training_data = TensorDataset(
67      encode(states[train_ids]),
68      actions[train_ids],
69      encode(next_states[train_ids]),
70  )
71
72  for _ in range(15):
73      model.train()
74      epoch_losses = []
75      for z, action, next_z in loader(training_data, batch=512):
76          z = z.to(DEVICE)
77          action = action.to(DEVICE)
78          next_z = next_z.to(DEVICE)
79          one_hot_action = F.one_hot(action, num_classes=2).float()
80          predicted_next_z = model(z, one_hot_action)
81          loss = F.mse_loss(predicted_next_z, next_z)
82
83          optimizer.zero_grad()
84          loss.backward()
85          optimizer.step()
86          epoch_losses.append(loss.item())
87      training_history.append(np.mean(epoch_losses))
88
89  # Measure one-step prediction on transitions excluded from optimization.
90  model.eval()
91  with torch.no_grad():
92      test_z = encode(states[test_ids]).to(DEVICE)
93      test_action = F.one_hot(actions[test_ids].to(DEVICE), num_classes=2).float()
94      test_next_z = encode(next_states[test_ids]).to(DEVICE)
95      one_step_mse = F.mse_loss(model(test_z, test_action), test_next_z).item()
```

Listing 23.12. Week-9 cell 5, part 2: train the dynamics.

Loss: mean squared error between predicted and actual next latent; 15 epochs, batches of 512. The 500 held-out transitions give the one-step error.

```
# ------------------------------------------------------------------------------
# Stage 4: plan by rolling candidate action sequences through the learned model.
# ------------------------------------------------------------------------------
start = torch.tensor([0.0, 0.0, 0.06, 0.0])
goal = torch.zeros(4)
horizon = 10

# With two actions and horizon ten, exhaustive search contains only 2^10=1024 plans.
candidates = torch.tensor(
    list(itertools.product([0, 1], repeat=horizon)), device=DEVICE
)
predicted_z = encode(start).to(DEVICE).repeat(len(candidates), 1)

with torch.no_grad():
    for step in range(horizon):
        candidate_action = F.one_hot(candidates[:, step], num_classes=2).float()
        predicted_z = model(predicted_z, candidate_action)

    # Decode standardized coordinates before applying task-specific physical scales.
    predicted_state = predicted_z * std.to(DEVICE) + mean.to(DEVICE)

# These scales make different physical units comparable. The score is therefore a
# weighted physical-state distance, not an unqualified Euclidean latent distance.
physical_scale = torch.tensor([2.4, 2.0, 0.2095, 2.0], device=DEVICE)
candidate_cost = torch.linalg.norm(
    (predicted_state - goal.to(DEVICE)) / physical_scale,
    dim=1,
)
best_plan = candidates[candidate_cost.argmin()].cpu().tolist()
```

Listing 23.13. Week-9 cell 5, part 3: plan by trying every 10-step action sequence in the model.

Start state: pole tilted by 0.06 radians (3.4°), everything else zero. `itertools.product` lists all $2^{10} = 1024$ left/right sequences; all are rolled through the model *in parallel* (one batch of 1024). The final predicted states are decoded to physical units and scored by distance to the goal (all zeros: centred, still, upright), with each quantity divided by a meaningful scale (2.4 is the track half-length, 0.2095 radians the angle at which the episode fails).

```
# ------------------------------------------------------------------------------
# Stage 5: execute the chosen plan in the real simulator.
# ------------------------------------------------------------------------------
# This final check matters because a plan can exploit errors in the learned model.
real_environment = gym.make("CartPole-v1", render_mode="rgb_array")
real_environment.reset(seed=SEED)
real_environment.unwrapped.state = start.numpy().copy()
```

```
executed_states = [start.numpy().copy()]

for action in best_plan:
    new_state, _, terminated, truncated, _ = real_environment.step(int(action))
    executed_states.append(new_state.copy())
    if terminated or truncated:
        break

final_frame = real_environment.render()
real_environment.close()
executed_states = np.asarray(executed_states)


def goal_cost(state):
    """Use the same normalized physical-state distance for real execution."""
    return float(np.linalg.norm(state / np.array([2.4, 2.0, 0.2095, 2.0])))


start_cost = goal_cost(executed_states[0])
final_cost = goal_cost(executed_states[-1])
print(f"One-step latent MSE: {one_step_mse:.6f}")
print("Plan:", " ".join("L" if action == 0 else "R" for action in best_plan))
print(f"Executed goal cost: {start_cost:.3f} -> {final_cost:.3f}")

# These plots connect model fit, real pole behavior, and the final rendered state.
fig, axes = plt.subplots(1, 3, figsize=(11, 3))
axes[0].plot(training_history)
axes[0].set_title("World-model loss")
axes[0].set_xlabel("epoch")
axes[1].plot(executed_states[:, 2])
axes[1].axhline(0, color="k")
axes[1].set_title("Executed pole angle")
axes[1].set_xlabel("plan step")
axes[2].imshow(final_frame)
axes[2].set_title("Final real state")
axes[2].axis("off")
plt.tight_layout()
plt.show()

# Fixed seeds make the selected plan improve this demonstration's real goal cost.
assert final_cost < start_cost
```

Listing 23.14. Week-9 cell 5, part 4: run the chosen plan in the real simulator.

The plan is executed open-loop in the true CartPole, from the same start state, and scored with the same cost. The final `assert` checks that the real cost went down.

23.4 What the notebook shows

The notebook was shipped without outputs; these are from running it for this book on a laptop CPU with its fixed seeds (total runtime about two minutes, most of it the two CycleGAN runs). A GPU gives slightly different numbers and the same picture.

Section 1: self-training. Printed: *pseudo-label reliability: all 0.680, accepted 0.951; test accuracy: teacher 0.664, student 0.673; accepted 1200/3000.*

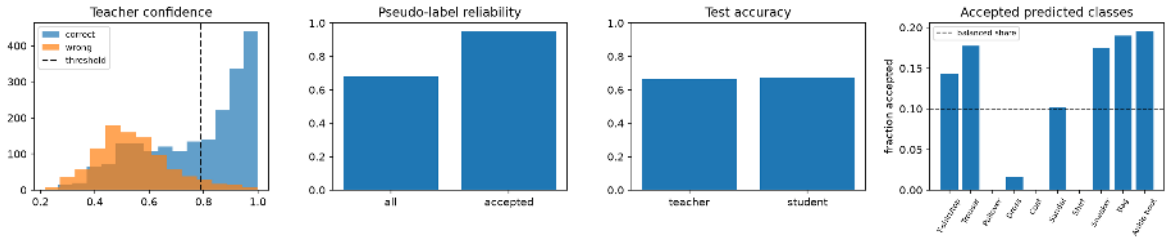


Figure 23.2. Section 1 outputs. From left: confidence of correct (blue) and wrong (orange) pseudo-labels with the threshold; reliability of all versus accepted pseudo-labels; teacher versus student test accuracy; share of each predicted class among accepted labels (dashed: the balanced 10%).

How to read it: Panel 1: orange bars to the right of the dashed line are confident mistakes that get through. Panel 4: three classes have no bar at all.

- *The filter works on average.* Only 68% of all pseudo-labels are right, but 95% of the accepted 40% are. Quantity was traded for quality.
- *It does not remove confident mistakes.* Panel 1 still shows orange (wrong) bars above the threshold: about 60 wrong labels enter the student's training set with full confidence (Question 7).
- *It silently drops whole classes.* Panel 4: *no* pullover, coat or shirt was accepted, and almost no dress. Those are the look-alike classes on which the teacher is never very sure. A global threshold accepted mostly easy classes (trousers, sneakers, bags, boots). This is exactly the risk in Question 13 — and it was visible *without* any ground truth.
- *The student barely improves* (0.664 → 0.673). With 1,000 labels, one round, and pseudo-labels concentrated on classes the teacher already handled, there is little new to learn. Improvement is possible, not guaranteed (Question 9).

Section 2A: soft targets. Printed: *compact-student test accuracy 0.668; agreement with teacher 0.842.* The half-width student is as accurate as its teacher and agrees with it on 84% of test images.

Figure 23.3 is Question 8 with a real example: the soft vector says "any of four quite different things"; the hard label claims certainty about one. A student trained on the hard label would be taught a confident answer the teacher never had.

Section 2B: two teachers. Printed: target shapes $(3, 10)$ and $(3, 1, 7, 7)$ for three images — the incompatible shapes of Question 15; held-out semantic agreement 0.457; edge error 0.222.

The edge head has learned the rough outline of each garment (Figure 23.4). The semantic head agrees with the teacher on only 46% of held-out images after 2 epochs: the shared encoder is small (width 10) and has to serve two masters. The notebook's point stands — two

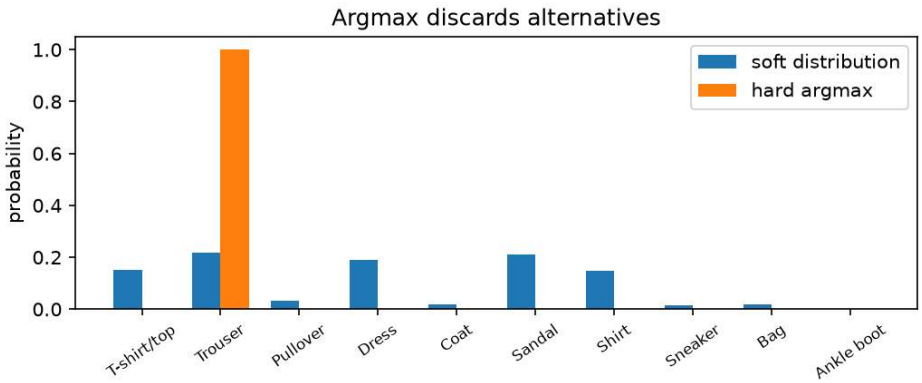


Figure 23.3. The pool image the teacher was least sure about: its soft distribution (blue) and the hard argmax target (orange).

How to read it: Four classes (trouser, sandal, dress, T-shirt) each get about 0.15–0.22; argmax turns "I have no idea" into "trouser, 100%".

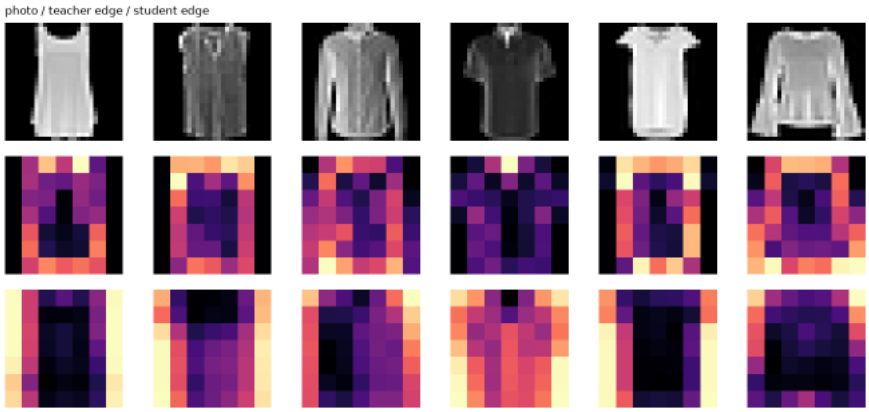


Figure 23.4. Section 2B on six held-out images. Rows: the photo; the Sobel teacher's 7×7 edge map; the student's edge head.

How to read it: Compare rows 2 and 3 column by column: the student's map has the garment's outline in the right place, but blurrier and with bright borders.

differently shaped targets *can* train one encoder once each has its own head — and so does its warning: the two heads must be evaluated separately, because neither number summarises the representation.

Section 3: CycleGAN with and without the cycle. Printed: *round-trip L1: adversarial only 0.532, with cycle 0.131.*

Figure 23.5 is Question 17 reproduced. Without the cycle loss, G produces what the discriminator accepts as "domain Y" — mostly white images — and the garment is largely lost: the round trip cannot recover it (L1 0.53). With the cycle loss, $G(x)$ is a recognisable dark-on-white version of the *same* garment and the round trip brings it back (L1 0.13). Two

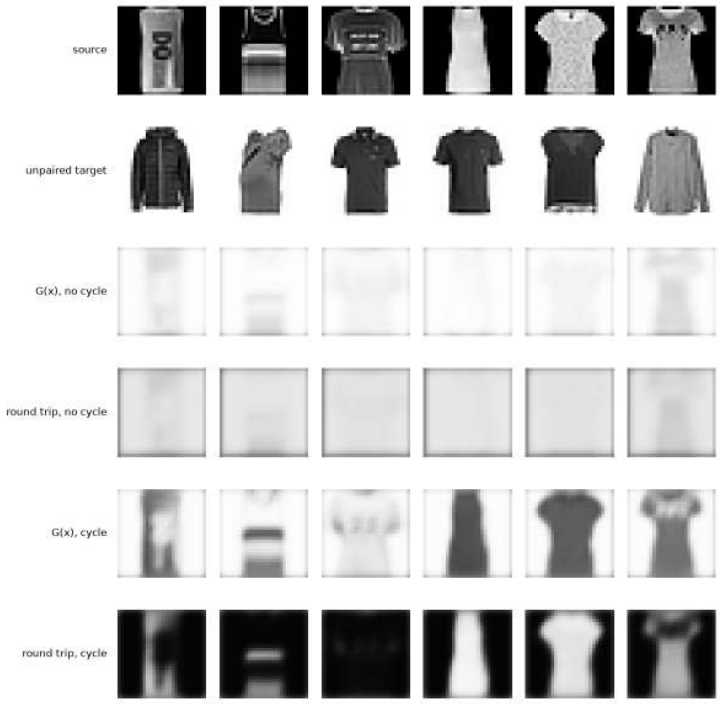


Figure 23.5. Rows from top: source image; an unpaired target-domain image (for reference); $G(x)$ without cycle loss; its round trip; $G(x)$ with cycle loss ($\lambda = 3$); its round trip.

How to read it: Row 3: white backgrounds, like domain Y — but the garments are nearly gone. Row 6: the garments come back.

further observations: the third column's translation still loses most of its print, so recoverability is not perfection; and, as the notebook warns, the cycle model is trained on the very L1 quantity being reported, so the number shows what the objective enforces, while the pictures show whether content is really kept.

Section 4: world model and planning. Printed: *one-step latent MSE 0.000126; plan R R R L R L L L L R; executed goal cost 0.286 → 0.047.*

The plan starts with three pushes to the right — moving the cart under the pole, which leans right — and then alternates to brake. In the real simulator the pole goes from 0.06 to about −0.004 radians and the goal cost falls from 0.286 to 0.047. The model was never shown a good action: it learned only what actions *do*, from random play, and the search found what to do.

23.5 The final reflection questions, answered

1. What if teacher confidence is high but badly calibrated for a rare class? Then the filter accepts that class's pseudo-labels exactly when they are most likely wrong, and the

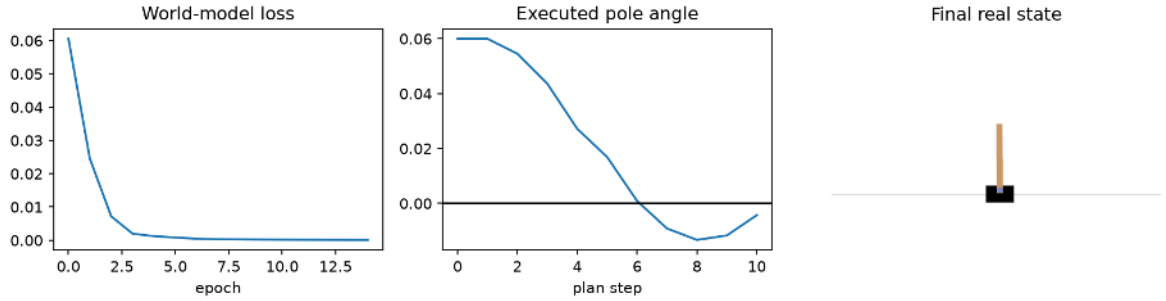


Figure 23.6. Left: world-model training loss. Middle: the pole angle while the chosen plan runs in the real simulator. Right: the final real state.

How to read it: Middle: the tilt of 0.06 *is brought to about* 0 *within ten steps, overshooting slightly to the other side.*

student learns the errors with full weight (confirmation bias, Question 7). The fix is to audit calibration per class on a small labelled set, use per-class thresholds or quotas, and watch the accepted-class histogram (Figure 23.2).

2. What is in a soft distribution but gone after argmax? The teacher's uncertainty (how close the runner-up was) and the similarity structure between classes (which wrong answers were plausible and which impossible). Figure 23.3: four plausible classes become one certain one.

3. Why can semantic vectors and edge maps train one encoder only after adding compatible heads? Losses compare a prediction with a target of the same shape and meaning. A 10-class probability vector and a 7×7 spatial map have neither in common, so there is no sensible loss between them or between their average and anything. A head per target produces each target's own kind of output; each loss is then well defined, and their gradients meet in the shared encoder.

4. Why can a target-realistic translation still fail to preserve its source? Because the adversarial loss scores the *set* of outputs, not the link between one input and its output. Any realistic output passes, including one unrelated to the input (row 3 of Figure 23.5). Only an instance-level constraint, such as the cycle loss, ties the output to its input.

5. Why can excellent one-step error coexist with poor long-horizon planning? Errors compound: a small error at every step feeds into the next prediction, and after many steps the predicted state can be far off. The planner also *searches* for sequences that score well in the model, so it actively seeks out places where the model is wrong in a flattering way. And a distance in feature space may not match task success. Hence the notebook's last step: execute in the real simulator and measure there.

Key takeaway — the notebook's closing line

Self-supervision defines a *training signal*, not a guarantee of correctness. Each section had a signal that could be satisfied in a bad way — confident wrong pseudo-labels, hard targets without uncertainty, realistic but unfaithful translations, a model the planner can fool — and each needed an evaluation aimed at the behaviour that was actually wanted.

CHAPTER 24

Tutorial Quiz: System-Level Self-Supervision

In this chapter

24.1 Where the supervision comes from 889
24.2 The ten questions and the bonus 889
24.3 The quiz on one page 895

Chapter roadmap

The quiz. Ten questions and a bonus on Lecture 09: paired versus unpaired data, pseudo-labels, fusing several sensors, consistency between models and over time, DF-Net's depth–pose–flow triangle, balancing a loss with several terms, multi-teacher distillation, and resets.

The one question behind all eleven. *Where does the supervision come from, and when can it be wrong?* The answer sheet ends with a table answering the first half; Section 24.1 extends it with the second half. Read it first.

What you need. Chapter 14, especially the geometry triangle (§14.4) and cross-modal distillation (§14.6).

24.1 Where the supervision comes from

method	supervision comes from	it misleads when ...
paired data	ground-truth correspondences	pairs are unavailable or wrong
unpaired data	cycle / consistency constraints	many mappings satisfy the constraint
pseudo-labels	the model's own predictions	the model is confidently wrong
cross-model consistency	agreement between models	the models share the same mistake, or are both still bad
forward–backward consistency	agreement over time	occlusion or fast motion breaks the round trip
multi-modal learning	agreement between sensors	one sensor fails or disagrees
multi-teacher distillation	agreement between teachers	the teachers' errors are the same
DF-Net	geometry + motion constraints	things move by themselves; blank surfaces
simple reset	escaping self-reinforcing errors	(see Question 10 for two meanings)

In plain words — the pattern

System-level self-supervision replaces a human labeller by a *check*: another model, another moment, another sensor, a law of geometry. Every such check has a blind spot where it passes a wrong answer. Most questions below are "which blind spot is this?".

24.2 The ten questions and the bonus

Q1 Day to night without pairs

You want to convert daytime driving images into nighttime images. You have 5,000 day and 5,000 night images with no correspondence. A student proposes minimising the pixel-wise difference between a day image and a randomly selected night image. What is the main flaw?

A. Pixel-wise supervision assumes paired correspondences that do not exist.
B. Nighttime images cannot be used for SSL.
C. Pixel-wise losses only work for segmentation.
D. The data set is too small.

Answer: A — a pixel loss needs the *same* scene as its target.

The idea. A pixel loss says "pixel (i, j) of my output should equal pixel (i, j) of the target". That only makes sense if both images show the same street from the same place. Against a

random night image of another street, the best the model can do is output the *average* night image for every input — blurry darkness that ignores the day image. Unpaired data needs a set-level loss (a discriminator) plus an instance-level constraint (the cycle), as in CycleGAN (§12.4).

Why not the others. B: night images are fine data. C: pixel losses are used for reconstruction, super-resolution and more. D: a million unpaired images would not fix it.

Q2 A detector reinforcing its own mistakes

An object detector generates pseudo-labels for unlabelled images and begins to reinforce several incorrect predictions. Which change most directly reduces this?

A. Use every pseudo-label immediately.
B. Accept only high-confidence predictions, or require agreement from another estimator.
C. Increase the learning rate.
D. Remove all labelled data.

Answer: B — filter by confidence, or by a second opinion.

The idea. Confirmation bias again (week-9 quiz, Question 7). Two filters: the model's own confidence, and an independent check — another model, another sensor, or another frame agreeing. The second is stronger, because it can catch confident mistakes. The week-9 notebook measured what a confidence filter does: reliability rose from 68% to 95% (§23.4).

Why not the others. A is the problem. C does not change label quality. D removes the only reliable anchor.

Q3 RGB, depth and IMU; the depth camera sometimes fails

A robot has RGB, depth and IMU sensors. Occasionally the depth camera fails. Which system is expected to be most robust?

A. Always trust the depth sensor.
B. Ignore missing depth measurements.
C. Dynamically fuse whichever modalities remain reliable.
D. Train separate models and average predictions only at test time.

Answer: C — fuse what is working, adapt when something breaks.

The idea. The sensors are complementary: depth gives distance, RGB gives appearance, the IMU gives the robot's own motion. A fusion system that notices when depth is missing or implausible (and relies more on RGB and IMU then) keeps working; one that always trusts depth has a single point of failure. Training with sensors randomly dropped is a common way to teach this.

Why not the others. A: single point of failure. B is ambiguous; if it means "carry on as if nothing were missing", the model receives garbage. D: separately trained models never learn how the sensors relate.

Q4 Forcing two untrained networks to agree

Two independently initialised segmentation networks consistently disagree on unlabelled data. Immediately forcing them to match each other could be harmful because ...

A. They may reinforce each other's mistakes before either has learned reliable representations.

B. Consistency losses only work for classification.

C. Two models cannot be trained simultaneously.

D. Different architectures should never agree.

Answer: A — agreement is easy; agreement on the truth is not.

The idea. Two networks can agree trivially — both predict "background" everywhere — and a consistency loss rewards that as much as agreeing on the right answer. Early in training neither is reliable, so the consistency term pulls both towards *some* common answer, not the right one. Practical rule: warm up each model on labelled data (or a sound self-supervised task) first, and ramp the consistency weight up gradually.

Why not the others. B, C, D are false.

Q5 Tracking a point there and back

A point is tracked from frame A to frame B and back to A. It returns far from where it started. Which explanation is most plausible?

A. The scene contains occlusions or incorrect motion estimates.

B. The camera resolution is too high.

C. The optimiser has converged.

D. Forward–backward consistency guarantees perfect reconstruction.

Answer: A — something hid the point, or a motion estimate is wrong.

The idea. If the forward flow sends pixel p to q and the backward flow sends q back to p, both are probably right. If it comes back elsewhere, either one estimate is wrong or the point is not visible in frame B (hidden behind a moving car, or out of the image) — then there is no correct answer to find. Systems use this test to build a *mask*: pixels that fail are excluded from the loss. DF-Net does exactly this (§14.4). *Numbers:* $p = (100, 50)$, forward flow $(+8, 0)$ gives $q = (108, 50)$; backward flow at q is $(-8, +1)$: back at $(100, 51)$, error 1 pixel — keep. If the backward flow is $(+2, -6)$: back at $(110, 44)$, error about 12 pixels — mask out.

Why not the others. B, C are unrelated. D: it is a test, not a guarantee.

Q6 DF-Net and moving objects

A self-driving car sees pedestrians and moving bicycles. A monocular depth model trained only with photometric reconstruction produces distorted depth around the moving objects. Why?

A. Photometric reconstruction assumes scene changes are explained only by camera motion.
B. Moving objects violate image brightness.
C. Depth cannot be estimated from videos.
D. Dynamic objects contain too few pixels.

Answer: A — the rule "only the camera moves" is false on them.

The idea. Photometric reconstruction rebuilds frame B from frame A using depth and camera motion. That works if the world is rigid. A cyclist riding along with the car barely moves in the image; the only way the depth–pose explanation can account for that is to put the cyclist *infinitely far away* (far things do not move in the image). So the network learns wrong depth exactly on moving objects — the classic "black hole" artefact.

A precision note on the answer sheet. The sheet says DF-Net "explicitly models dynamic objects". More precisely, DF-Net *detects* where the rigid rule fails — by comparing the rigid prediction with the separately learned optical flow and with forward–backward checks — and *masks those pixels out* of the consistency losses; the optical-flow network is free to follow objects that move by themselves (§14.4, Figure 14.39).

Why not the others. B: the brightness of a moving object usually stays constant; its *motion* is the problem. C: it can. D: size is not the issue.

Q7 Depth maps without edges

Your total loss has a reconstruction, a smoothness and a consistency term. Depth maps come out extremely smooth and object boundaries disappear. Which term is most likely overweighted?

A. Reconstruction loss.
B. Smoothness loss.
C. Consistency loss.
D. Data augmentation.

Answer: B — smoothness.

The idea. The smoothness term charges for depth differences between neighbouring pixels. It is needed on blank walls and sky, where the photometric term cannot tell depths apart. But real depth *jumps* at object boundaries; with too much smoothness weight, the cheapest solution is to blur every jump. The usual remedy is edge-aware smoothness: weight the penalty by $e^{-|\nabla I|}$, so it relaxes where the image itself has an edge.

Why not the others. A pulls towards the real image, which has sharp edges. C asks for agreement, not flatness. D is not a loss term.

Q8 Low reconstruction error, wrong depth

A depth network and a pose network jointly reconstruct consecutive frames. Reconstruction error is low, but the depth maps are clearly unrealistic. Why?

A. Multiple incorrect depth–pose combinations can produce similar reconstructions.
B. Low reconstruction error always guarantees correct depth.
C. Pose estimation is unnecessary.
D. Depth should always be supervised.

Answer: A — the problem has many answers that fit the pixels.

The idea. Only the combination of depth and motion is checked. The simplest example is scale: double every depth and double the camera's step, and every pixel lands in the same place (§14.4). Other ambiguities: on a textureless wall any depth reconstructs equally well; when the camera only rotates, depth has no effect on the image at all. Low reconstruction error means "consistent with the images", not "correct".

Why not the others. B is the misconception. C: pose is essential. D: supervision helps but is not required.

Q9 Two confident teachers that disagree

Two teacher models disagree on many unlabelled samples, and both are highly confident. What is the strongest indication that using both teachers may still improve the student?

A. Both teachers have exactly the same architecture.
B. Both teachers predict every sample with high confidence.
C. The student has more parameters than either teacher.
D. Their errors are complementary rather than identical.

Answer: D — they are wrong in *different* places.

The idea. A second teacher adds value only where it knows something the first does not. If teacher 1 fails at night and teacher 2 (thermal) fails in bright sun, together they cover both. If both fail on the same images, the second adds nothing but confidence. MM-DistillNet's RGB, depth and thermal teachers are chosen for exactly this reason (§14.6).

Why not the others. A: identical architectures tend to make identical mistakes. B: confidence is not correctness. C: student size does not explain why two teachers help.

Q10 Resetting part of the network

A self-supervised model has converged to poor pseudo-labels and no longer improves. Why can periodically resetting part of the network help?

A. It helps escape self-reinforcing errors while preserving useful learned representations.
B. Resetting increases the number of training samples.
C. Resetting removes the need for pseudo-labels.
D. Resetting guarantees convergence to the global optimum.

Answer: A — forget the bad habit, keep the useful features.

The idea. A model trained on its own pseudo-labels can lock into a loop: its outputs define its targets, which confirm its outputs. Reinitialising the *last* layers (the head that produces the labels) while keeping the earlier feature layers breaks the loop: the new head must re-learn the mapping from good features, and often finds a better one. In reinforcement learning the same trick is used against the "primacy bias" — a network over-committed to what it saw first.

Two meanings of "simple reset". Be careful with the course's wording. In Lecture 09, "Reinforcement Learning: self-supervision by simple reset" is the title of the ReLMM slides and means resetting the *world*, not the network: the robot puts the room back into a new starting state itself by driving a random path and dropping the object (the pseudo-reset, §14.7.6). What the two share is the idea behind the answer: a cheap reset creates new starting points, so learning does not get stuck on the same situation.

Why not the others. B: no new data. C: the labels are still needed. D: no guarantee.

Bonus Only two modules fit on the GPU

You have four modules: pseudo-label generator, cross-model consistency, forward–backward consistency, multi-teacher distillation. You can keep only two. Which pair is most likely to produce reliable pseudo-labels on an unlabelled *video*?

A. Pseudo-labels + cross-model consistency.
B. Cross-model consistency + forward–backward consistency.
C. Forward–backward consistency + multi-teacher distillation.
D. Pseudo-label generation + multi-teacher distillation.

Answer: — — the answer sheet: no single correct answer; it depends on the application.

How to reason it out. Split the four modules into two roles. *Producers* create labels: the pseudo-label generator, and multi-teacher distillation (the teachers' outputs are labels). *Checkers* filter labels: cross-model consistency, and forward–backward consistency. Reliable labels need at least one of each. That rules out D (two producers, no check) and weakens B (two checkers; the labels are just the two models' own predictions, with nothing producing a strong starting label).

- **C** is the strongest choice *when good pretrained teachers exist*: several teachers produce labels whose errors hopefully differ (Question 9), and the forward–backward check is free on video and catches what they get wrong at occlusions and fast motion (Question 5).
- **A** is the fallback *when no teachers exist*: one model produces pseudo-labels and a second, differently initialised model vetoes the ones it disagrees with — after both have been warmed up (Question 4).

The skill being tested is not the letter but the argument: name who produces the labels, who checks them, and what the check cannot see.

24.3 The quiz on one page

Q	Ans.	The one-line lesson (the answer sheet's key takeaways)
1	A	Pixel losses need paired data; unpaired data needs other constraints.
2	B	Filter or cross-check pseudo-labels against confirmation bias.
3	C	Fusing complementary sensors makes the system robust.
4	A	Cross-model consistency helps only once the models are decent.
5	A	Forward–backward checks flag occlusions and bad motion.
6	A	Moving objects break the "only the camera moves" rule.
7	B	Overweighted smoothness erases boundaries.
8	A	Low reconstruction error does not pin down depth.
9	D	Multiple teachers help when their errors differ.
10	A	Resets break self-reinforcing loops.
Bonus	—	Pair a label producer with a label checker.

Key takeaway — end of the tutorials

Across all eight tutorials one habit kept giving the right answer: ask *where the target comes from* (data, a model, a sensor, time, a law of physics, or a person) and *what could make that target wrong without anyone noticing.* The first question classifies the method; the second tells you what to measure.

CHAPTER 25

The First-Cycle Exam, Reconstructed and Solved

In this chapter

25.1 The format and how to spend 120 minutes 897
25.2 The questions, solved 898 • Coding: a VAE with the reparameterisation trick • Scenario: SSL for robot object handling • Scenario: an image from a segmentation map • Scenario: clustering images with about six categories • Coding: a batch k-means with augmentations and labelling • Coding: DINO
25.3 What the paper tells you about the next one 906

Chapter roadmap

What this chapter is. The first exam of the course, rebuilt from what students reported right after it, with a full-credit model answer to every question we know. The next three chapters turn it into a *question bank* in exactly the same pattern: multiple-choice questions with theory and calculations (Chapter 26), coding on paper (Chapter 27), and scenario case studies with a mock exam (Chapter 28).

How reliable the reconstruction is. The exam format and the five long questions quoted in §25.2 come from students' messages; wording is approximate. Two coding questions were only described ("a custom k-means batch with data augmentations and labelling", "DINO"); for those, §25.2.5 and §25.2.6 give the most likely form and solve it, and the question bank covers the variations. No MCQ was reported verbatim; the MCQ bank covers every lecture.

What students said. "Lengthy and tough", "too many cases to read", MCQs with "both theoretical and calculations". The exam rewards reading fast, recognising the pattern, and writing structured answers — which is what the next section trains.

25.1 The format and how to spend 120 minutes

Part	Points	What it was
Multiple choice	$25 \times 2 = 50$	theory and small calculations, all lectures
Coding on paper	part of 70	VAE with the reparameterisation trick; a batch k-means with augmentations and labelling; DINO
Scenarios / case studies	part of 70	robot object handling; image generation from segmentation maps; clustering images with about 6 categories
Total	120	120 minutes (Lecture 07 logistics); two handwritten A4 sheets allowed

The time budget. 120 points in 120 minutes is *one point per minute*. A 2-point MCQ is worth 2 minutes; a 15-point scenario is worth 15 minutes, including reading. Students ran out of time because the case texts are long. A plan that works:

1. **Minutes 0–5:** skim the whole paper; mark each long question with its points.
2. **Minutes 5–45:** all 25 MCQs, at most 90 seconds each. Calculation MCQs: write the formula first, then the numbers. Skip and mark anything that takes longer.
3. **Minutes 45–110:** long questions in order of points per minute *for you*: coding first if you know the template by heart (they are the fastest points), then scenarios.
4. **Minutes 110–120:** return to marked MCQs; never leave an MCQ blank.

Key takeaway — the answer skeletons that earn the points

Scenario ("select a method, architecture, loss, assumptions, failure modes"). One line per heading, in that order: *Method* (name it and say why it fits the data you have). *Architecture* (the blocks and what flows between them; a small sketch). *Loss* (the formula, every symbol named, what it rewards). *Assumptions* (what must be true of the data). *Failure modes* (two concrete ways it breaks, each with a fix).

"Discuss the suitability of A, B, C, D". For each: *can it do the job at all?* (yes/no and the one reason), *its main strength*, *its main weakness here*. End with a one-sentence choice.

Coding on paper. Define the blocks once, keep shapes in comments, and write the loss explicitly. Examiners grade the structure, not syntax.

25.2 The questions, solved

25.2.1 Coding: a VAE with the reparameterisation trick

Check Your Understanding

Pseudocode exercise: VAE with the reparameterisation trick.

a) Complete the code below. You can use: `net(d_in, d_out)` creates a feed-forward neural-network block with the given input and output dimension; `linear(d_in, d_out)` creates a linear projection block; `rand(dim)` samples a random vector of the given dimension. Given: `d_latent`, `d_input`.

```
encode(x):

decode(z):

forward(x):
    ...
    z =
    return decode(z)
```

b) Why do we use reparameterisation? What problem does it solve?

```
# blocks, created once (d_hidden is a free choice, e.g. 256)
enc_body   = net(d_input, d_hidden)       # x -> h
enc_mu     = linear(d_hidden, d_latent)   # h -> mean of q(z|x)
enc_logvar = linear(d_hidden, d_latent)   # h -> log-variance of q(z|x)
dec        = net(d_latent, d_input)       # z -> reconstruction x_hat

encode(x):                      # x: (d_input,)
    h = enc_body(x)
    mu = enc_mu(h)              # (d_latent,)
    logvar = enc_logvar(h)      # (d_latent,)  log sigma^2, any real number
    return mu, logvar

decode(z):                      # z: (d_latent,)
    return dec(z)               # x_hat: (d_input,)

forward(x):
    mu, logvar = encode(x)
    eps = rand(d_latent)                # eps ~ N(0, I): the ONLY random part
    sigma = exp(0.5 * logvar)
    z = mu + sigma * eps                # reparameterisation trick
    return decode(z)

# training loss for one x (negative ELBO), beta = 1 for the true ELBO
loss(x):
    mu, logvar = encode(x)
    z = mu + exp(0.5 * logvar) * rand(d_latent)
    x_hat = decode(z)
    recon = sum((x - x_hat)**2)                                  # -log p(x|z) up to consts
    kl = -0.5 * sum(1 + logvar - mu**2 - exp(logvar))        # KL(q(z|x) || N(0,I))
    return recon + beta * kl
```

Listing 25.1. Model answer to (a).

What earns the points in (a). (i) The encoder has *two* outputs, a mean and a (log-)variance — a VAE encoder outputs a distribution, not a code. (ii) `rand` is used only for ϵ, and $z = \mu + \sigma \odot \epsilon$. (iii) The variance is kept positive, e.g. by predicting $\log \sigma^2$ and exponentiating. (iv) The decoder maps `d_latent` back to `d_input`. Writing the loss is not asked for but costs one minute and shows you know why the pieces exist. State the assumption that `rand` draws from a standard normal; if it draws uniformly, transform it (e.g. Box–Muller) or say so.

Model answer to (b). Training needs the gradient of the loss with respect to the encoder parameters ϕ. But z is *sampled* from $q_\phi(z \mid x) = \mathcal{N}(\mu_\phi(x), \sigma^2_\phi(x))$, and sampling is not a differentiable operation: backpropagation cannot say how a random draw changes when μ changes. The trick rewrites the sample as a deterministic, differentiable function of the parameters plus parameter-free noise, $z = \mu + \sigma \odot \epsilon$ with $\epsilon \sim \mathcal{N}(0, I)$. Now $\partial z/\partial \mu = 1$ and $\partial z/\partial \sigma = \epsilon$, so gradients flow from the reconstruction loss through z into the encoder, and the expectation in the ELBO can be estimated by Monte Carlo with low variance (one sample

per image is usually enough). Without it, one would need the high-variance score-function (REINFORCE) estimator. (§4.6.2; week-3 quiz Question 8.)

25.2.2 Scenario: SSL for robot object handling

Check Your Understanding

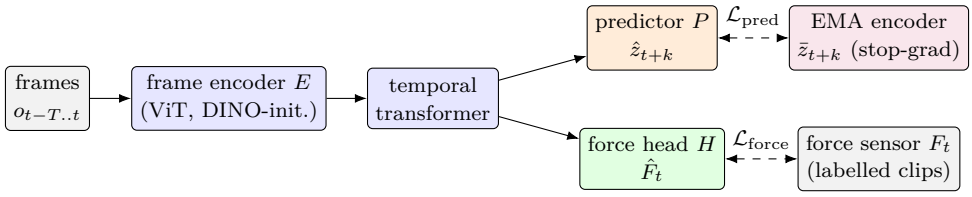
Exercise: SSL for robot object handling (about 15 points). We want a robot to learn to handle objects. Available data: (1) unlabelled RGB videos of humans handling objects; (2) labelled RGB videos in the same set-up that additionally contain measurements of the interaction forces. Tasks: select an adequate SSL method; briefly explain the architecture; explain the objective (loss); what are the key assumptions; what are likely failure modes?

Method. A two-stage recipe that uses each data source for what it is good at.

1. *Self-supervised pretraining on the unlabelled videos*: learn a visual representation of hands, objects and their motion by **predicting the future in feature space** — a latent world model in the style of DINO-WM / JEPA: encode frames, and train a predictor to predict the features of frame $t + k$ from frames up to t (the next frame is a free target, §13.2). A masked video autoencoder is an acceptable alternative.
2. *Cross-modal supervision from the force sensor*: in the labelled videos the forces are recorded automatically by the sensor — they are "free labels" from another modality, exactly like WVN's footstep measurements or MM-DistillNet's camera teachers (§14.5, §14.6). Train a force head on the pretrained features to predict the interaction force from video. Optionally pseudo-label the unlabelled videos with this head and retrain (self-training with a confidence filter).

The robot then uses the video-to-force model as a sensor it does not have (predict the force a grasp will produce) and the world model to plan manipulation towards a goal image.

Architecture.



A frame encoder E (a ViT, ideally initialised from DINO) turns each frame into features; a small temporal transformer mixes the last T frames; two heads sit on top: a predictor P for future features and a regression head H for the force (3D force, possibly torque) at time t. The future target comes from an EMA copy of the encoder with stop-gradient.

Objective.

$$\begin{aligned}\mathcal{L} \;=\; & \underbrace{\big\|P(z_{\le t}) - \mathrm{sg}(\bar{E}(o_{t+k}))\big\|_2^2}_{\text{all videos: predict the future}} \\ & +\; \lambda \underbrace{\mathbb{1}[\text{labelled}]\,\big\|H(z_{\le t}) - F_t\big\|_2^2}_{\text{labelled videos: predict the force}} \\ & +\; \mu \quad \underbrace{w_i\,\big\|H(z_{\le t}) - \tilde{F}_i\big\|_2^2}_{\text{optional: confident pseudo-labels } \tilde{F}_i} \quad .\end{aligned}$$

The first term needs no labels and teaches how hands and objects move; the stop-gradient and EMA target prevent collapse (the trivial solution of predicting a constant). The second term ties the representation to physical interaction; λ balances them.

Key assumptions.

- Both video sets come from the *same set-up* (camera, objects, lighting), so features learned on one transfer to the other.
- Forces are *inferable from vision*: contact, deformation, acceleration and hand posture carry information about the force.
- Video and force streams are *time-synchronised*.
- Human demonstrations are relevant to the robot: what matters about objects transfers across the human–robot embodiment gap.

Likely failure modes (and fixes).

- **Force ambiguity**: a heavy and a light object held the same way look the same; vision alone cannot tell the force. Fix: predict a distribution or add object identity/mass cues; accept a ceiling.
- **Occlusion**: the hand hides the contact region exactly when the force matters. Fix: several cameras; temporal context.
- **Embodiment gap**: human hands versus a robot gripper look and act differently, so features and forces do not transfer one to one. Fix: fine-tune on a few robot clips; mask out the hand.
- **Shortcuts**: the model predicts force from which object or which background is visible, not from the interaction. Fix: vary objects and scenes; check per-object errors.
- **Collapse** of the prediction loss without stop-gradient/EMA; **overfitting** the small labelled set; **confirmation bias** if pseudo-labels are used unfiltered; sensor noise and drift.

25.2.3 Scenario: an image from a segmentation map

Check Your Understanding

Exercise. We are given images and their corresponding segmentation maps. We want to generate an image conditioned on a given segmentation map. Discuss the suitability of: an unconditional GAN; a conditional GAN; kernel density estimation (KDE); normalising flows.

First, read the data. We have *pairs* (map, image). The task is conditional and one-to-many: one map allows many realistic images (colours, textures, lighting). A good answer uses both facts.

Approach	Suitable?	Why
Unconditional GAN	no	Generates from noise only; there is no input through which the map could enter, so the output cannot follow the given layout. Samples would be realistic street scenes, but of *any* layout.
Conditional GAN (Pix2Pix, SPADE)	**yes, best**	Generator takes the map (and noise); discriminator judges *(map, image) pairs*, so it penalises images that do not fit their map. Because we have pairs, add an L1 loss to the true image for faithfulness. Sharp, fast samples. Weaknesses: training instability; mode collapse — Pix2Pix tends to ignore the noise, giving one image per map.
KDE	no	Non-parametric density in pixel space: with tens of thousands of dimensions it suffers the curse of dimensionality; a conditional KDE would return (blurred mixtures of) *training images with similar maps*, not a new image that follows a new map. Fine for low-dimensional data only.
Normalising flow	possible, weaker	A conditional flow (condition the coupling layers on the map) gives exact likelihoods and naturally diverse samples. But invertibility forces the latent to be as large as the image and restricts the layers, so at image resolution flows are memory-heavy and their samples less sharp than GANs'.

Conclusion. A conditional GAN, trained on the pairs with an adversarial loss on (map, image) pairs plus an L1 reconstruction loss:

$$\min_G \max_D \; \mathbb{E}\big[\log D(s,x)\big] + \mathbb{E}\big[\log(1 - D(s, G(s,z)))\big] + \lambda\, \mathbb{E}\big\|x - G(s,z)\big\|_1.$$

Mention that today a conditional (latent) diffusion model with classifier-free guidance would be the strongest choice, though it was not among the options (§7.7). (Generative models quiz Questions 7 and 10, Chapter 19.)

25.2.4 Scenario: clustering images with about six categories

Check Your Understanding

Exercise. We have unlabelled images with large variations in lighting, background, etc. We know the data contains approximately 6 semantic categories. We extract features from the data.

a) What are two preprocessing operations you would consider before clustering?
b) Discuss the suitability of k-means, DBSCAN, and overclustering with k-means. Which approach would you initially choose?

(a) Two preprocessing operations.

1. **Normalise the features**: L2-normalise each feature vector (so Euclidean distance behaves like cosine distance, which is how SSL features such as DINO's are compared, §3.1), or standardise each dimension. Otherwise brightness and feature magnitude — exactly the lighting nuisance — dominate the distances.
2. **Reduce the dimension with PCA** (optionally with whitening) to about 64–256 dimensions: removes noisy, correlated directions, makes distances more meaningful, and makes clustering faster — the step DeepCluster uses before k-means (§3.3).

Other acceptable answers, operating on the images: photometric normalisation (per-image standardisation, histogram equalisation, colour constancy) against the lighting variation; cropping to the object or masking the background against the background variation; using features from a network trained to be invariant to colour jitter and crops (an SSL encoder) rather than raw pixels.

(b) Suitability.

- **k-means** ($k = 6$). Natural fit because k is known; fast; scales with mini-batches. But it assumes compact, roughly spherical, similar-sized clusters and assigns every point, so outliers pull the centroids; it depends on initialisation (use k-means++, several restarts). Danger here: with strong lighting/background variation, the 6 clusters may split by *lighting* instead of by category, or one category with two looks gets split while two similar categories merge.
- **DBSCAN**. Needs no k (we have one anyway), finds arbitrary shapes, marks outliers as noise. But in high dimensions a single density threshold ε is hard to choose, categories with different densities break it (one merged blob or many fragments), it cannot be told "6", and memory grows as $O(n^2)$. Least suitable here.
- **Overclustering with k-means** ($k \gg 6$, e.g. 30–60). Each category may consist of several visual modes (the same object in daylight and at night, on grass and on a table); many small clusters can each be pure, and are then *merged* into 6 (agglomerative merging

of centroids, a few labels, or inspection). It keeps k-means' speed and fixes its spherical-cluster assumption by covering each category with several spheres. DeepCluster found that using many more clusters than classes (10,000 for ImageNet's 1,000) gives better features. Cost: a merging step, and the number of sub-clusters is a new hyperparameter.

Initial choice. Start with **k-means with $k = 6$** on L2-normalised, PCA-reduced features, initialised with k-means++ and several restarts: it uses the known k, is cheap, and is a baseline you can inspect (look at images nearest each centroid). If clusters turn out mixed or split by lighting — likely given the stated variation — move to **overclustering and merging**. Either choice earns full credit when it is justified with the data's properties; a choice without a reason does not.

25.2.5 Coding: a batch k-means with augmentations and labelling

Students described a coding question on "custom k-means batch with data augmentations and labelling". The course material it rests on is mini-batch k-means (Sculley 2010, Lecture 02 slide 17, §3.2.4) and DeepCluster's pseudo-labels. The most likely form:

Check Your Understanding

Exercise (reconstructed). Complete a mini-batch k-means that clusters image features. Each image is augmented before its features are computed; cluster assignments are then used to label the data. Available: `encoder(x)`, `augment(x)`, `minibatches(X, b)`, `argmin`, `norm`.

```
# init: k centroids from k random images (or k-means++ on a sample)
C = [normalize(encoder(augment(x))) for x in random_sample(X, k)]   # (k, d)
counts = zeros(k)                                                   # points seen per
    ↪ centroid

for it in range(num_iters):
    for x_batch in minibatches(X, b):
        # 1) features of an augmented view, L2-normalised (cosine geometry)
        f = normalize(encoder(augment(x_batch)))       # (b, d)
        # 2) assignment step: nearest centroid for each point
        a = argmin(dist(f, C), axis=1)                 # (b,)  cluster ids
        # 3) update step, one point at a time, per-centroid learning rate
        for i in range(b):
            j = a[i]
            counts[j] += 1
            eta = 1.0 / counts[j]                      # running-mean step size
            C[j] = (1 - eta) * C[j] + eta * f[i]       # move centroid towards point
        C = normalize(C)                               # keep centroids on the sphere

# labelling: assign every image to a cluster using the CLEAN image
labels = [argmin(dist(normalize(encoder(x)), C)) for x in X]
# optional: name clusters with a few human labels -> majority label per cluster
```

Listing 25.2. Model answer: mini-batch k-means with augmented views and final labelling.

What earns the points. (i) The two alternating steps inside each batch: *assign* to the nearest centroid, then *update* only the assigned centroids. (ii) The per-centroid learning rate $\eta = 1/n_j$, which makes each centroid the running mean of the points it has received (Sculley's rule); a fixed small η is acceptable if stated. (iii) Augmentation is applied *before* the encoder, so every pass sees a new view and the clusters must hold across views — clusters become invariant to the augmentations (a crop or colour change of a cat stays with the cats). (iv) Final labels come from the *un-augmented* image. (v) Normalising features (and centroids) if distances are cosine.

Variations to be ready for. *Consistency labelling*: assign both of two augmented views and keep a pseudo-label only if both views land in the same cluster (a confidence filter). *Empty clusters*: re-seed a centroid that receives no points from a random point of the largest cluster (DeepCluster's fix). *Using the labels*: train a classifier on the cluster ids with cross-entropy (DeepCluster, Chapter 29, exam-style question 3). All three are written out in the coding bank (§27.1).

25.2.6 Coding: DINO

The DINO coding question was not reported in detail. The standard exam form is "complete the training step"; the model answer below follows the DINO paper's own pseudocode (§11.3).

```
# gs, gt: student and teacher networks (backbone + projection head), same architecture
# C: center (K,), tps, tpt: student / teacher temperatures (e.g. 0.1 / 0.04)
# l: EMA momentum for the teacher (e.g. 0.996 -> 1), m: center momentum (e.g. 0.9)
gt.params = gs.params                          # teacher starts as a copy
for x in loader:                               # minibatch of images
    x1, x2 = augment(x), augment(x)            # two global views (+ local crops in full
    ↪ DINO)
    s1, s2 = gs(x1), gs(x2)                    # student outputs  (n, K)
    t1, t2 = gt(x1), gt(x2)                    # teacher outputs  (n, K)

    loss = H(t1, s2)/2 + H(t2, s1)/2           # each view's student predicts the OTHER
    ↪ view's teacher
    loss.backward()                            # gradients reach the STUDENT only
    update(gs)                                 # SGD / AdamW step on student params

    gt.params = l*gt.params + (1-l)*gs.params          # teacher = EMA of student
    C = m*C + (1-m)*cat([t1, t2]).mean(dim=0)          # update center (running mean of
    ↪ teacher outputs)

def H(t, s):
    t = t.detach()                                     # stop-gradient on the teacher
    s = softmax(s / tps, dim=1)                        # student: normal temperature
    t = softmax((t - C) / tpt, dim=1)                  # teacher: center, then SHARPEN
    ↪ (low temp)
    return -(t * log(s)).sum(dim=1).mean()             # cross-entropy teacher -> student
```

Listing 25.3. Model answer: one DINO training step.

What earns the points. Five mechanisms, each with its reason: *two views, cross-prediction* (the student must match the teacher on a different view: invariance); *stop-gradient* on the teacher; *EMA* teacher (stable targets, no negatives needed); *centering* (subtract a running mean before the teacher softmax: stops one output dimension from winning for all images); *sharpening* (teacher temperature lower than the student's: stops the flat, uniform output). Centering and sharpening push in opposite directions; together they prevent both kinds of collapse (week-8 quiz Question 8). In full DINO the teacher sees only the global crops while the student also sees several small local crops (multi-crop), which teaches "local-to-global" correspondence.

25.3 What the paper tells you about the next one

1. **Every lecture appears.** Clustering (Lecture 02), autoencoders/VAE (03), generative models (04), representation learning and DINO (07), systems/robotics (09) all had long questions. Nothing is safe to skip.
2. **Coding is three templates.** VAE forward pass, k-means (batch, with augmentation, with labels), DINO step. Learn the bank's templates until you can write them in five minutes each; they are the cheapest points on the paper.

3. **Scenarios reward structure.** The robot question asked for five named things; each is a heading in the answer. The "discuss suitability" questions reward one reason per option and a final choice.
4. **MCQs include calculations.** Expect to compute a KDE value, a k-means update, a KL term, an InfoNCE loss, an IoU, a patch count. Chapter 26 has a calculation block with the formulas you should have on your A4 sheet.

CHAPTER 26

Exam Question Bank I: Multiple Choice, Theory and Calculations

In this chapter

26.1 Part A: theory, lecture by lecture ..... 909 • Lecture 02: nearest neighbours and clustering • Lecture 03: autoencoders and VAEs • Lecture 04–05: generative models, diffusion • Lecture 06: pretext tasks and augmentations • Lecture 07: representation learning • Lecture 08: self-training, cycle consistency, world models • Lecture 09: systems
26.2 Part B: calculations 919
26.3 Answer key 924
26.4 The formula sheet for the calculations 925

Chapter roadmap

What this chapter is. Sixty multiple-choice questions in the exam's pattern: one correct answer, 2 points each, about 90 seconds each. Part A (35 questions) tests understanding, lecture by lecture. Part B (25 questions) is the part students found hardest: *small calculations* — every one can be done by hand in under two minutes.

How to use it. Do a block of 25 in 40 minutes, as in the exam, then check. Every answer comes with the reason, and every calculation with the worked numbers. Questions that repeat a quiz from Chapters 17 to 24 are not repeated here: do those quizzes too (another 107 questions).

The formula sheet. Section 26.4 lists every formula Part B needs; copy it onto your A4 sheet.

26.1 Part A: theory, lecture by lecture

26.1.1 Lecture 02: nearest neighbours and clustering

A1 kd-trees in high dimensions

Why do kd-trees lose their advantage for nearest-neighbour search on 768-dimensional features?

A. They require labels to split the space.
B. They only work with cosine distance.
C. In high dimensions almost every cell must be visited, so search degrades towards brute force.
D. They cannot store more than 2^{10} points.

Answer: C — the curse of dimensionality defeats the pruning.

A kd-tree prunes cells that cannot contain a closer point. In high dimensions distances concentrate (all points are about equally far), so few cells can be ruled out. Rule of thumb: kd-trees help when $n \gg 2^d$. Approximate methods (LSH, graph indices) take over (§3.1.2).

A2 What LSH trades

Locality-sensitive hashing speeds up nearest-neighbour search by …

A. hashing so that close points collide with high probability, accepting occasional misses.
B. sorting all points by their norm.
C. training a classifier on the neighbours.
D. guaranteeing the exact nearest neighbour in $O(1)$.

Answer: A — approximate search: speed for occasional misses.

Random hyperplanes (for cosine) put nearby vectors in the same bucket with high probability; only the bucket is searched (§3.1.3).

A3 Cosine versus Euclidean

For L2-normalised feature vectors, ranking neighbours by Euclidean distance and by cosine similarity gives …

A. the same ranking, since $\|a-b\|^2 = 2 - 2\cos(a,b)$.
B. the same ranking only in 2D.
C. opposite rankings.
D. different rankings in general.

Answer: A — on the unit sphere the two are monotonically related.

A4 k-means++

What does k-means++ change compared with random initialisation?

A. It removes outliers before clustering.
B. It picks each new initial centre with probability proportional to its squared distance to the nearest centre already chosen.
C. It chooses k automatically.
D. It replaces the update step with gradient descent.

Answer: B — spread-out seeds, $O(\log k)$-competitive in expectation.

(§3.2.3.)

A5 What k-means minimises

Lloyd's algorithm is guaranteed to …

A. find clusters of arbitrary shape.
B. find the global minimum of the within-cluster sum of squares.
C. never increase the within-cluster sum of squares, converging to a local minimum.
D. detect outliers.

Answer: C — monotone decrease, local optimum.

Both the assignment step and the mean update can only lower the objective, and there are finitely many assignments; the global problem is NP-hard.

A6 DBSCAN's parameters

In DBSCAN, a *core point* is a point that …

A. was labelled by a human.
B. is closest to a cluster mean.
C. lies on the convex hull of a cluster.
D. has at least `min_samples` points within distance ε.

Answer: D — density is counted in an ε-ball.

Border points are within ε of a core point but are not core themselves; everything else is noise (§3.2.5).

A7 Why DeepCluster works at all

DeepCluster starts from a randomly initialised network. Why can its first pseudo-labels teach anything?

A. k-means uses the true labels internally.
B. Features of even a random CNN cluster images better than chance, and that weak signal is amplified over iterations.
C. PCA removes all noise.
D. Random networks already output class labels.

Answer: B — "better than chance" — the slide's emphasised sentence.

A8 DeepCluster's trivial solutions

Which failure must DeepCluster guard against?

A. Too-fast convergence of PCA.
B. Too many labels.
C. Empty or giant clusters, which let the classifier reach low loss by predicting almost one class.
D. Negative learning rates.

Answer: C — degenerate clusters.

Fixes: re-seed empty clusters, sample images uniformly over clusters.

26.1.2 Lecture 03: autoencoders and VAEs

A9 Denoising autoencoder

A denoising autoencoder receives $\tilde{x} = x + \text{noise}$. Its loss compares the output with ...

A. the clean x.
B. a class label.
C. $\tilde{x}$.
D. the noise.

Answer: A — reconstruct the clean input.

Copying $\tilde{x}$ is no longer the solution, so the code must capture structure that tells signal from noise.

A10 Beta above one

Raising β in a β-VAE (recon $+ \beta$ KL) mainly ...

A. sharpens reconstructions.

B. removes the need for the reparameterisation trick.

C. pushes the posterior closer to the prior: more regular latent space, blurrier reconstructions, risk of posterior collapse.

D. makes the model a GAN.

Answer: C — more KL pressure.

A11 The ELBO gap

$\log p(x) -$ ELBO equals ...

A. zero for every VAE.

B. the reconstruction error.

C. $\mathrm{KL}(q(z \mid x) \,\|\, p(z \mid x))$.

D. $\mathrm{KL}(q(z \mid x) \,\|\, p(z))$.

Answer: C — the gap is the KL to the *true posterior*.

Option D is the KL *term inside* the ELBO — the classic confusion.

26.1.3 Lecture 04–05: generative models, diffusion

A12 GAN training objective at the optimum

With the optimal discriminator, the original GAN generator minimises (up to constants) ...

A. the squared error to the nearest training image.

B. the KL divergence $\mathrm{KL}(p_g \| p_{\text{data}})$.

C. the Jensen–Shannon divergence between p_{data} and p_g.

D. the ELBO.

Answer: C — Jensen–Shannon.

(§6.4.)

A13 Why flows need a tractable Jacobian

In the change-of-variables formula, the term $\log|\det J|$ ...

A. is optional.
B. is the reconstruction loss.
C. is only needed at sampling time.
D. accounts for how the map stretches or compresses volume; it must be cheap, hence triangular Jacobians (coupling layers).

Answer: D — volume change; triangular ⇒ product of diagonal.

A14 Mode collapse versus posterior collapse

Which pair is matched correctly?

A. Mode collapse: VAE decoder ignores z. Posterior collapse: GAN outputs one kind of sample.
B. Both are the same thing.
C. Both happen only in flows.
D. Mode collapse: GAN covers few modes. Posterior collapse: VAE posterior equals the prior and z is ignored.

Answer: D — GAN ↔ modes; VAE ↔ posterior.

A15 DDPM versus DDIM

DDIM samples in 50 steps from a model trained with 1000 DDPM steps because ...

A. it drops the noise predictor.
B. it uses a deterministic (non-Markovian) sampler with the same trained noise predictor, skipping timesteps.
C. it samples in pixel space instead of latent space.
D. it retrains the network for 50 steps.

Answer: B — same network, different sampler.

A16 Latent diffusion's risk

What is lost in latent diffusion that pixel diffusion would keep?

A. Conditioning on text.
B. The ability to sample.
C. The noise schedule.
D. Any detail the autoencoder cannot reconstruct.

Answer: D — the autoencoder's reconstruction ceiling.

26.1.4 Lecture 06: pretext tasks and augmentations

A17 Colourisation as a pretext

Why can colourising grey images teach semantics?

A. Colours are class labels.

B. Predicting plausible colours requires recognising what objects are (grass green, sky blue).

C. It is a contrastive task.

D. It uses a GAN discriminator.

Answer: B — the task needs object knowledge.

A18 Context prediction shortcut

In relative-patch-position prediction, networks exploited chromatic aberration. What is this an example of?

A. Collapse.

B. Overfitting to labels.

C. Mode collapse.

D. A shortcut: a low-level cue that solves the pretext task without semantics.

Answer: D — shortcut.

Fixes: gaps and jitter between patches, colour dropping.

A19 Augmentations as inductive bias

Choosing colour jitter as an augmentation tells the model ...

A. to predict colours.

B. representations should be invariant to colour changes — harmful if colour matters downstream (e.g. flower species, ripe fruit).

C. nothing; augmentations only enlarge the data.

D. colour is important for the downstream task.

Answer: B — augmentations encode what should *not* matter.

(§8.3.)

26.1.5 Lecture 07: representation learning

A20 Why negatives in contrastive learning

In SimCLR, the negatives are needed to ...

A. prevent collapse: without them, mapping every image to one point minimises the positive-pair loss.

B. provide labels.

C. increase the batch size.

D. speed up augmentation.

Answer: A — repulsion prevents collapse.

A21 BYOL without negatives

Which pair of ingredients lets BYOL avoid collapse without negatives?

A. Labels and a classifier head.

B. A predictor on the online branch and an EMA target network with stop-gradient.

C. Large batches and a memory bank.

D. Pixel reconstruction and a KL term.

Answer: B — asymmetry + slow target.

A22 Barlow Twins

Barlow Twins avoids collapse by ...

A. pushing the cross-correlation matrix of the two views' embeddings towards the identity.

B. using a momentum queue.

C. a discriminator.

D. predicting rotations.

Answer: A — diagonal $\to 1$ (invariance), off-diagonal $\to 0$ (redundancy reduction).

(§10.5.)

A23 VICReg's three terms

VICReg's loss consists of ...

A. reconstruction, KL and adversarial terms.

B. variance, invariance and covariance terms.

C. InfoNCE, triplet and margin terms.

D. cross-entropy, centering and sharpening.

Answer: B — keep each dimension's std above a threshold, match views, decorrelate dimensions.

A24 Dimensional collapse

Embeddings that are not constant but occupy only a low-dimensional subspace show ...

A. complete collapse.
B. overfitting.
C. dimensional collapse — visible as many near-zero singular values of the embedding matrix.
D. mode collapse.

Answer: C — dimensional collapse.

A25 DINO's multi-crop

In DINO's multi-crop, which views go through the teacher?

A. Only the local crops.
B. All crops, global and local.
C. Only the global crops; the student sees all crops.
D. None; the teacher is not used during training.

Answer: C — local-to-global correspondence.

A26 DINO's attention maps

The self-attention of DINO's CLS token on ViT features is notable because ...

A. it was trained with segmentation masks.
B. it segments the main object without any mask supervision.
C. it only attends to the image border.
D. it equals the supervised ViT's attention.

Answer: B — emergent object segmentation.

A27 iBOT's contribution

iBOT adds to DINO ...

A. pixel reconstruction with a decoder.
B. a GAN loss.
C. a masked-image-modelling loss in feature space: the student predicts the teacher's features for masked patches.
D. negatives from a queue.

Answer: C — patch-level self-distillation.

26.1.6 Lecture 08: self-training, cycle consistency, world models

A28 Knowledge distillation temperature

Distilling with temperature $T > 1$ ...

A. removes the teacher.

B. softens both distributions so that small probabilities ("dark knowledge") carry gradient; the loss is scaled by T^2.

C. turns it into hard pseudo-labelling.

D. makes the teacher's distribution sharper.

Answer: B — softening.

A29 Cycle consistency without adversarial loss

If CycleGAN is trained with only the cycle loss, a likely solution is ...

A. mode collapse to one image.

B. G and F both close to the identity map: round trips are perfect, but nothing is translated.

C. a normalising flow.

D. perfect domain translation.

Answer: B — identity satisfies $F(G(x)) = x$.

The adversarial loss is what forces $G(x)$ to look like the target domain.

A30 Planning in DINO-WM

At inference, DINO-WM chooses actions by ...

A. copying expert actions.

B. optimising an action sequence (e.g. CEM or MPC) so that the predicted final latent is close to the goal image's latent.

C. a policy network trained with rewards.

D. reconstructing future pixels and comparing them with the goal.

Answer: B — search in latent space, no reward, no decoder needed.

(§13.4.)

A31 Why freeze the encoder in DINO-WM

DINO-WM keeps DINOv2 frozen. The main reason is ...

A. training the encoder with the prediction loss alone invites collapse (predict a constant latent); a frozen pretrained encoder gives a fixed, meaningful target space.

B. frozen encoders need no GPU.

C. DINOv2 cannot be fine-tuned.

D. it makes the actions discrete.

Answer: A — fixed target space, no collapse.

26.1.7 Lecture 09: systems

A32 Photometric loss and scale

A monocular depth-and-pose system trained with a photometric loss recovers depth ...

A. in metres.

B. only with LiDAR supervision at training time.

C. only up to an unknown global scale.

D. only for moving objects.

Answer: C — scaling depth and translation together leaves every pixel unchanged.

A33 WVN's label source

Wild Visual Navigation learns traversability from ...

A. human-annotated terrain maps.

B. the robot's own experience: how well it tracked its commanded velocity where it walked, projected back onto the images of those places.

C. satellite images.

D. simulation only.

Answer: B — proprioception as the label.

A34 MM-DistillNet's student

In MM-DistillNet, the student at test time uses ...

A. RGB, depth and thermal cameras.

B. only the microphone array's audio.

C. human labels.

D. LiDAR.

Answer: B — sound only; the cameras were teachers at training time.

A35 ReLMM's grasp choice

ReLMM scores grasps by ensemble mean plus a multiple of the ensemble's standard deviation. The standard deviation term …

A. measures the gripper force.
B. is only used after training.
C. penalises uncertain grasps.
D. rewards grasps the ensemble disagrees about, because trying them is informative (exploration).

Answer: D — optimism under uncertainty.

26.2 Part B: calculations

Write the formula first, then substitute. All numbers are chosen so that the arithmetic is quick; $e^1 \approx 2.718$, $e^2 \approx 7.39$, $\ln 2 \approx 0.693$, $\phi(1) = \frac{1}{\sqrt{2\pi}} e^{-1/2} \approx 0.242$.

B1 A k-means update

A cluster contains $(1, 2)$, $(3, 4)$, $(5, 0)$. The new centroid is …

A. $(9, 6)$ (B) $(3, 2)$ (C) $(2, 3)$ (D) $(3, 3)$

Answer: B — $(\frac{1+3+5}{3}, \frac{2+4+0}{3}) = (3, 2)$.

B2 A mini-batch k-means step

Centroid $c = (0, 0)$ has received 3 points so far. A new point $x = (4, 0)$ is assigned to it. With Sculley's per-centre rate $\eta = 1/n$, the new centroid is …

A. $(2, 0)$ (B) $(1, 0)$ (C) $(1.33, 0)$ (D) $(4, 0)$

Answer: B — $n \to 4$, $\eta = 1/4$: $c = \frac{3}{4}(0, 0) + \frac{1}{4}(4, 0) = (1, 0)$.

This is exactly the running mean of the four points.

B3 The k-means objective

Points $0, 2, 10, 12$ on a line, centroids 1 and 11. The within-cluster sum of squares is …

A. 8 (B) 2 (C) 4 (D) 0

Answer: C — $1 + 1 + 1 + 1 = 4$.

B4 DBSCAN by hand

Points on a line: $0, 1, 2, 5, 6, 10$; $\varepsilon = 1.5$; `min_samples` $= 3$ (counting the point itself). How many clusters and how many noise points?

A. 3 clusters, 0 noise (B) 2 clusters, 1 noise (C) 1 cluster, 3 noise (D) 1 cluster, 1 noise

Answer: C — only point 1 is core.

Neighbourhoods: 0: $\{0,1\}$, 1: $\{0,1,2\}$ (core), 2: $\{1,2\}$, 5: $\{5,6\}$, 6: $\{5,6\}$, 10: $\{10\}$. Cluster $\{0,1,2\}$ (0 and 2 are border points); 5, 6, 10 are noise — 5 and 6 are close, but neither has enough neighbours to be core.

B5 A KDE value

Gaussian KDE with bandwidth $h = 1$ on samples $\{0, 2\}$. The density at $x = 1$ is …

A. 0.484 (B) 0.121 (C) 0.242 (D) 0.399

Answer: C — $\hat{p}(1) = \frac{1}{2}[\phi(1-0) + \phi(1-2)] = \frac{1}{2}(0.242 + 0.242) = 0.242$.

General form: $\hat{p}(x) = \frac{1}{nh}\sum_i \phi(\frac{x-x_i}{h})$.

B6 KL of a Gaussian to the prior

$\mathrm{KL}(\mathcal{N}(\mu, \sigma^2) \,\|\, \mathcal{N}(0,1))$ for $\mu = 1$, $\sigma = 1$ is …

A. 0.5 (B) 1 (C) 0 (D) 2

Answer: A — $\frac{1}{2}(\sigma^2 + \mu^2 - 1 - \ln\sigma^2) = \frac{1}{2}(1 + 1 - 1 - 0) = 0.5$.

B7 KL with a wide posterior

Same formula, $\mu = 0$, $\sigma = 2$:

A. 2.0 (B) 1.5 (C) 0 (D) 0.81

Answer: D — $\frac{1}{2}(4 + 0 - 1 - \ln 4) = \frac{1}{2}(3 - 1.386) = 0.807$.

Being *wider* than the prior costs too, not only being off-centre.

B8 Reparameterisation

The encoder outputs $\mu = 1$, $\log\sigma^2 = \ln 4$; the sampled $\epsilon = -0.5$. Then $z =$

A. 0.5 (B) -1 (C) 3 (D) 0

Answer: D — $\sigma = e^{0.5\ln 4} = 2$, $z = 1 + 2(-0.5) = 0$.

B9 A β-VAE loss

Reconstruction term 20, KL term 100, $\beta = 0.1$. The loss is …

A. 12 (B) 120 (C) 30 (D) 2

Answer: C — $20 + 0.1 \times 100 = 30$.

B10 MAE token count and cost

A 224×224 image, 16×16 patches, 75% masking. How many tokens does the encoder see, and by roughly what factor is self-attention cheaper than on all tokens?

A. 49 tokens, 16× (B) 147 tokens, 16× (C) 196 tokens, 1× (D) 49 tokens, 4×

Answer: A — $(224/16)^2 = 196$ patches, 25% visible = 49; attention is quadratic: $(196/49)^2 = 16$.

(The MLP layers are linear in the tokens, so the whole encoder is about 3–4× cheaper, not 16×.)

B11 Noise schedule

Constant $\beta_t = 0.1$. After 3 steps, $x_3 = a\,x_0 + b\,\epsilon$ with ...

A. $a = 0.9$, $b = 0.1$

B. $a \approx 0.854$, $b \approx 0.521$

C. $a = 0.7$, $b = 0.3$

D. $a = 0.729$, $b = 0.271$

Answer: B — $\bar{\alpha}_3 = 0.9^3 = 0.729$; $a = \sqrt{0.729} \approx 0.854$, $b = \sqrt{0.271} \approx 0.521$.

Option D is the trap: forgetting the square roots.

B12 Classifier-free guidance

$\epsilon_u = 0.2$, $\epsilon_c = 0.5$, $w = 3$. The guided prediction is ...

A. 0.5 (B) 1.5 (C) 1.1 (D) 0.9

Answer: C — $0.2 + 3(0.5 - 0.2) = 1.1$.

B13 An Euler step of rectified flow

Trained velocity $v = z - x$ (data to noise). At $x_t = 5$ the model predicts $v = 3$; step $dt = 0.5$ towards the data. The next point is ...

A. 6.5 (B) 3.5 (C) 5.0 (D) 2.0

Answer: B — $x - v\,dt = 5 - 1.5 = 3.5$.

B14 Tweedie's score

Noisy value $x_t = 3$, optimal denoiser output $D(x_t) = 1$, $\sigma^2 = 4$. The score $\nabla \log p(x_t)$ is ...

A. −0.5 (B) −2 (C) 0.5 (D) −8

Answer: A — $(D - x_t)/\sigma^2 = (1 - 3)/4 = -0.5$.

B15 Optimal discriminator

At some image, $p_{\text{data}} = 0.3$ and $p_g = 0.1$. The optimal GAN discriminator outputs ...

A. 0.75 (B) 3 (C) 0.25 (D) 0.5

Answer: A — $D^* = p_{\text{data}}/(p_{\text{data}} + p_g) = 0.3/0.4 = 0.75$.

B16 A coupling layer's log-determinant

An affine coupling layer leaves half the coordinates unchanged and scales the other three by 2, 0.5 and 4. $\log|\det J| =$

A. $\ln 4 \approx 1.386$ (B) 0 (C) $\ln 2$ (D) $\ln 6.5$

Answer: A — triangular Jacobian: $\ln 2 + \ln 0.5 + \ln 4 = \ln 4$ (the unchanged half contributes $\ln 1 = 0$).

B17 Softmax with temperature

Logits $(2, 1, 0)$, temperature $\tau = 0.5$. The largest probability is about ...

A. 0.33 (B) 0.67 (C) 0.87 (D) 0.99

Answer: C — logits$/\tau = (4, 2, 0)$: $54.6/(54.6 + 7.39 + 1) \approx 0.87$.

At $\tau = 1$ it would be $7.39/(7.39 + 2.72 + 1) \approx 0.67$ (option B): lower temperature, sharper distribution.

B18 An InfoNCE loss

Cosine similarity of the query to its positive is 0.8, to two negatives 0.2 each; $\tau = 0.5$. The loss is about ...

A. 0.47 (B) 0.80 (C) 1.10 (D) 0.13

Answer: A — logits $(1.6, 0.4, 0.4)$: $p_+ = 4.953/(4.953 + 2 \times 1.492) = 0.624$, loss $-\ln 0.624 = 0.47$.

Trap: forgetting to divide by τ gives logits $(0.8, 0.2, 0.2)$, $p_+ = 2.23/(2.23 + 2 \times 1.22) = 0.48$ and loss 0.74. With $\tau = 0.1$ the same similarities give logits $(8, 2, 2)$ and loss 0.005: a lower temperature makes the positive win by far more.

B19 Number of negatives in SimCLR

Batch of $N = 256$ images, two views each. For one view, how many negatives are in its InfoNCE denominator?

A. 511 (B) 510 (C) 255 (D) 256

Answer: B — $2N$ views, minus itself, minus its positive: $2N - 2 = 510$.

B20 EMA teacher

Teacher weight 1.0, student weight 2.0, momentum $m = 0.99$. After one EMA update the teacher weight is ...

A. 1.99 (B) 2.0 (C) 1.5 (D) 1.01

Answer: D — $0.99 \times 1.0 + 0.01 \times 2.0 = 1.01$.

B21 DINO's centre update

Centre $c = (0.5, 0.5)$, batch mean of teacher outputs $(1, 0)$, centre momentum 0.9. The new centre is ...

A. $(0.95, 0.05)$ (B) $(1, 0)$ (C) $(0.5, 0.5)$ (D) $(0.55, 0.45)$

Answer: D — $0.9(0.5, 0.5) + 0.1(1, 0) = (0.55, 0.45)$.

B22 SimSiam loss value

Predictor output $p = (1, 0)$, target $z = (1, 1)$. The negative cosine similarity is ...

A. -0.707 (B) 0.707 (C) 0 (D) -1

Answer: A — $-\frac{1 \cdot 1 + 0 \cdot 1}{1 \cdot \sqrt{2}} = -0.707$.

B23 Linear-probe size

A linear probe on 768-dimensional frozen features for 1,000 classes has how many trainable parameters?

A. 769,000 (B) 768 (C) 1,768 (D) 768,000

Answer: A — 768×1000 weights $+1000$ biases.

B24 IoU for merging detections

Boxes $[0, 0, 4, 4]$ and $[2, 2, 6, 6]$ (corners). Their IoU is ...

A. 0.143 (B) 0.25 (C) 0.5 (D) 0.33

Answer: A — overlap $2 \times 2 = 4$; union $16 + 16 - 4 = 28$; $4/28 = 0.143$.

With NMS at threshold 0.5 both boxes are kept (MM-DistillNet's label merge).

B25 Cross-entropy of hard versus soft targets

The teacher predicts $(0.52, 0.45, 0.03)$. A student that outputs exactly this distribution has cross-entropy against the *hard* pseudo-label (class 1) of ...

A. 0.81 (B) 0.65 (C) 0 (D) 0.52

Answer: B — $-\ln 0.52 = 0.654$.

Against the *soft* target the cross-entropy is the entropy $-\sum p \ln p = 0.340 + 0.359 + 0.105 = 0.805$ (option A) — the minimum possible for that target, so the student is already perfect under distillation but still "wrong" under the hard label.

26.3 Answer key

A1 C	A2 A	A3 A	A4 B	A5 C	A6 D	A7 B	A8 C	A9 A	A10 C
A11 C	A12 C	A13 D	A14 D	A15 B	A16 D	A17 B	A18 D	A19 B	A20 A
A21 B	A22 A	A23 B	A24 C	A25 C	A26 B	A27 C	A28 B	A29 B	A30 B
A31 A	A32 C	A33 B	A34 B	A35 D					
B1 B	B2 B	B3 C	B4 C	B5 C	B6 A	B7 D	B8 D	B9 C	B10 A
B11 B	B12 C	B13 B	B14 A	B15 A	B16 A	B17 C	B18 A	B19 B	B20 D
B21 D	B22 A	B23 A	B24 A	B25 B					

26.4 The formula sheet for the calculations

quantity	formula
k-means objective	$\sum_j \sum_{x \in C_j} \|x - \mu_j\|^2$; update $\mu_j = \frac{1}{\|C_j\|} \sum_{x \in C_j} x$
mini-batch k-means	$n_j \leftarrow n_j + 1,\ \ \mu_j \leftarrow (1 - \frac{1}{n_j})\mu_j + \frac{1}{n_j} x$
KDE	$\hat{p}(x) = \frac{1}{nh} \sum_i \phi\big(\frac{x - x_i}{h}\big),\ \ \phi(u) = \frac{1}{\sqrt{2\pi}} e^{-u^2/2}$
Gaussian KL to $\mathcal{N}(0,1)$	$\frac{1}{2} \sum_j (\sigma_j^2 + \mu_j^2 - 1 - \ln \sigma_j^2)$
reparameterisation	$z = \mu + \sigma\epsilon,\ \sigma = e^{\frac{1}{2} \log \sigma^2}$
ELBO	$\mathbb{E}_q[\log p(x \mid z)] - \mathrm{KL}(q(z \mid x) \| p(z)) = \log p(x) - \mathrm{KL}(q \| p(z \mid x))$
flow likelihood	$\log p(x) = \log p_Z(f(x)) + \log \lvert \det \partial f / \partial x \rvert$
optimal discriminator	$D^*(x) = p_{\text{data}} / (p_{\text{data}} + p_g)$
diffusion forward	$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon,\ \bar{\alpha}_t = \prod_s (1 - \beta_s)$
x_0 from ϵ	$\hat{x}_0 = (x_t - \sqrt{1 - \bar{\alpha}_t}\, \epsilon_\theta) / \sqrt{\bar{\alpha}_t}$
CFG	$\tilde{\epsilon} = \epsilon_u + w(\epsilon_c - \epsilon_u)$
Tweedie	$\nabla \log p(x_t) = (D(x_t) - x_t)/\sigma^2$
softmax with temperature	$p_i = e^{s_i/\tau} / \sum_j e^{s_j/\tau}$
InfoNCE	$-\ln \frac{e^{s_+/\tau}}{e^{s_+/\tau} + \sum_i e^{s_i^-/\tau}}$; SimCLR negatives $= 2N - 2$
EMA	$\theta_t \leftarrow m\theta_t + (1 - m)\theta_s$; DINO centre $c \leftarrow mc + (1 - m)\bar{t}$
cosine	$\cos(a, b) = a \cdot b / (\|a\| \|b\|)$; $\|a - b\|^2 = 2 - 2\cos$ for unit vectors
MAE	patches $= (H/P)^2$; visible $= (1 - r) \times$patches; attention cost $\propto$ tokens^2
IoU	overlap area / (area$_1$ + area$_2$ − overlap)
focal loss	$-\alpha(1 - p_t)^\gamma \ln p_t$
ensemble grasp score	α·mean + β·std; sample $\propto e^{\text{score}}$

CHAPTER 27

Exam Question Bank II: Coding on Paper

In this chapter

27.1 Clustering 926
27.2 Autoencoders and generative models 929
27.3 Representation learning 932
27.4 Self-training, cycles, world models, systems 934

Chapter roadmap

What this chapter is. Every algorithm in the course that can be asked as "complete the pseudocode", in the exam's style: a short task with the building blocks you may use, a model answer, and the points an examiner looks for. The first exam asked for three of them (VAE, a batch k-means with augmentations and labelling, DINO, Chapter 25); the course's own example asked for DeepCluster (Chapter 29, exam-style question 3).
The building blocks. Unless a task says otherwise you may use: `net(d_in, d_out)` (an MLP or CNN block), `linear(d_in, d_out)`, `rand(dim)` (standard normal), `randint`, `encoder(x)`, `augment(x)`, `minibatches(X, b)`, `normalize` (L2), `softmax`, `log`, `exp`, `mean`, `argmin`/`argmax`, `cat`, `detach` (stop-gradient), `sgd_step(loss)`.
How to practise. Read only the task box, write your answer on paper in five minutes, then compare. Syntax does not matter; structure, shapes and the loss do.

27.1 Clustering

Check Your Understanding

C1. Mini-batch k-means (Lecture 02, slide 17). Complete the mini-batch version of k-means for data `X` with k clusters and batch size b. Explain why the learning rate should differ per centroid.

```
C = X[random_choice(n, k)]            # (k, d) initial centroids, or k-means++
counts = zeros(k)
for it in range(T):
    B = X[random_choice(n, b)]        # (b, d) random minibatch
    a = argmin(dist(B, C), axis=1)    # cache assignments first (assignment step)
    for i in range(b):                # then update (update step)
        j = a[i]
        counts[j] += 1
        eta = 1 / counts[j]
        C[j] = (1 - eta) * C[j] + eta * B[i]
```

Listing 27.1. C1 model answer (Sculley, 2010).

Why per-centroid rates. With $\eta_j = 1/n_j$ each centroid is exactly the running mean of all points ever assigned to it: early points move it a lot, later points less, so it converges. A centroid that is rarely chosen keeps a large step size and can still move. Memory is $O(kd)$ whatever the data size — the reason it scales where full k-means and DBSCAN do not.

Check Your Understanding

C2. k-means on augmented views, with labelling (the exam's variant). Images must be clustered by content, not by lighting. Use `augment` so that the clusters are robust to it, keep a pseudo-label only when two augmented views of the image agree, and name the clusters with a small labelled set `(X_lab, y_lab)`.

```
f = lambda x: normalize(encoder(x))              # features on the unit sphere
C = normalize(f(augment(X[random_choice(n, k)])))
counts = zeros(k)
for epoch in range(E):
    for xb in minibatches(X, b):
        z = f(augment(xb))                       # a NEW random view each time
        a = argmax(z @ C.T, axis=1)              # nearest centroid = max cosine
        for i in range(len(xb)):
            j = a[i]; counts[j] += 1; eta = 1 / counts[j]
            C[j] = normalize((1 - eta) * C[j] + eta * z[i])
    # empty clusters: re-seed from a random point of the largest cluster
    for j in where(counts == 0):
        C[j] = normalize(C[largest_cluster()] + 0.01 * rand(d))

# pseudo-labels with a two-view consistency check
pseudo = {}
for x in X:
    a1 = argmax(f(augment(x)) @ C.T); a2 = argmax(f(augment(x)) @ C.T)
    if a1 == a2:
        pseudo[x] = a1                           # keep only consistent assignments

# naming clusters with a few labels: majority vote per cluster
for j in range(k):
    members = [y for x, y in zip(X_lab, y_lab) if argmax(f(x) @ C.T) == j]
    name[j] = majority(members) if members else "unknown"
```

Listing 27.2. C2 model answer.

What earns the points. Augment before the encoder, inside the loop; nearest centroid by cosine on normalised features; the running-mean update; the consistency filter (an assignment that flips between two views of the same image is unreliable); naming by majority vote. Mention that the filter reduces but does not eliminate wrong labels.

Check Your Understanding

C3. k-means++ seeding. Write the seeding procedure and state its guarantee.

```
C = [X[randint(n)]]                              # first centre uniformly at random
while len(C) < k:
    D2 = [min(dist(x, c)**2 for c in C) for x in X]   # squared distance to nearest
    ↪ centre
    probs = D2 / sum(D2)
    C.append(X[sample_index(probs)])             # far points are more likely
```

Listing 27.3. C3 model answer.

Guarantee: the expected k-means cost after seeding is within $O(\log k)$ of the optimum (Arthur and Vassilvitskii, 2007) — before any Lloyd iteration.

Check Your Understanding

C4. DBSCAN. Complete the cluster expansion given `neighbours(i)` (indices within ε, including i) and `min_samples`.

```
label = [UNVISITED] * n; c = 0
for i in range(n):
    if label[i] != UNVISITED: continue
    N = neighbours(i)
    if len(N) < min_samples:
        label[i] = NOISE; continue          # may later become a border point
    label[i] = c; queue = list(N)
    while queue:
        j = queue.pop()
        if label[j] == NOISE: label[j] = c   # border point
        if label[j] != UNVISITED: continue
        label[j] = c
        Nj = neighbours(j)
        if len(Nj) >= min_samples:           # j is core: keep expanding
            queue += Nj
    c += 1
```

Listing 27.4. C4 model answer.

Points to state: only *core* points expand a cluster; border points join but do not expand; the number of clusters is an output; naive `neighbours` for all points needs $O(n^2)$ time/memory.

Check Your Understanding

C5. DeepCluster. See Chapter 29, exam-style question 3 (the course's own example): re-extract features every epoch, PCA, k-means, cluster ids as pseudo-labels, cross-entropy.

27.2 Autoencoders and generative models

Check Your Understanding

C6. A denoising autoencoder. Complete `loss(x)` using `net`, a bottleneck of size `d_code` and Gaussian noise of standard deviation `s`.

```
enc = net(d_input, d_code); dec = net(d_code, d_input)
loss(x):
    x_noisy = x + s * rand(d_input)       # corrupt the INPUT
    x_hat = dec(enc(x_noisy))
    return sum((x_hat - x)**2)            # compare with the CLEAN x
```

Listing 27.5. C6 model answer.

Check Your Understanding

C7. Generate new images with the trained VAE of the exam, and interpolate between two images.

```
sample():                                # no encoder needed
    z = rand(d_latent)                   # z ~ N(0, I), the prior
    return decode(z)

interpolate(x0, x1, steps):
    mu0, _ = encode(x0); mu1, _ = encode(x1)   # use the means, no noise
    return [decode((1 - a) * mu0 + a * mu1) for a in linspace(0, 1, steps)]
```

Listing 27.6. C7 model answer.

Why sampling from $\mathcal{N}(0, I)$ works for a VAE and not for a plain autoencoder: the KL term made the codes fill the prior (Chapter 17, Question 16).

Check Your Understanding

C8. KDE. Write `density(x)` and `sample()` for a Gaussian KDE with bandwidth `h` on data `X` (`n` points, dimension `d`).

```
density(x):
    k = [exp(-sum((x - xi)**2) / (2*h**2)) for xi in X]
    return sum(k) / (n * (2*pi*h**2)**(d/2))
sample():
    i = randint(n)                       # pick a training point uniformly
    return X[i] + h * rand(d)            # add kernel noise
```

Listing 27.7. C8 model answer.

Check Your Understanding

C9. One GAN training step with generator `G`, discriminator `D` (outputs a probability) and noise size `d_z`.

```
for x in minibatches(X, b):
    z = rand(b, d_z)
    fake = G(z)
    # 1) discriminator: real -> 1, fake -> 0
    loss_D = -mean(log(D(x))) - mean(log(1 - D(detach(fake))))
    sgd_step(loss_D)                    # updates D only
    # 2) generator: make D call the fakes real
    loss_G = -mean(log(D(G(z))))
    sgd_step(loss_G)                    # updates G only
```

Listing 27.8. C9 model answer (non-saturating generator loss).

Points: `detach` the fake in the D step; alternate the two updates; the non-saturating form $-\log D(G(z))$ gives strong gradients early, when D easily rejects fakes. Conditional GAN: feed the condition s to both, `G(z, s)` and `D(x, s)`.

Check Your Understanding

C10. Diffusion (DDPM): the training step and the sampler. Given `abar[t]`, `alpha[t]`, `beta[t]` for $t = 1..T$ and a noise predictor `eps_theta(x, t)`.

```
train_step(x0):
    t = randint(1, T); eps = rand(shape(x0))
    x_t = sqrt(abar[t]) * x0 + sqrt(1 - abar[t]) * eps
    return mean((eps_theta(x_t, t) - eps)**2)       # target: the NOISE

sample():
    x = rand(shape)                                  # start from pure noise
    for t in range(T, 0, -1):
        z = rand(shape) if t > 1 else 0
        eps_hat = eps_theta(x, t)
        x = (x - beta[t] / sqrt(1 - abar[t]) * eps_hat) / sqrt(alpha[t]) + sqrt(beta[t])
    ↪ * z
    return x
```

Listing 27.9. C10 model answer.

Tutorial 5, Questions 1 and 3 are the two lines most often broken on purpose: the one-line jump to x_t, and the target ϵ (not x_0).

Check Your Understanding

C11. Rectified flow / flow matching: training and a 4-step Euler sampler, with data x at $t = 0$ and noise at $t = 1$.

```
train_step(x):
    z = rand(shape(x)); t = uniform(0, 1)
    x_t = (1 - t) * x + t * z          # straight line from data to noise
    v_target = z - x                   # its constant velocity
    return mean((v_theta(x_t, t) - v_target)**2)

sample(N=4):
    x = rand(shape); dt = 1 / N
    for t in linspace(1, dt, N):       # t = 1, 0.75, 0.5, 0.25
        x = x - v_theta(x, t) * dt     # MINUS: walk from noise to data
    return x
```

Listing 27.10. C11 model answer.

Check Your Understanding

C12. MAE: masking and the loss. Patches `P` (shape `(N, D)`), mask ratio `r`; encoder `enc`, decoder `dec`, learned `mask_token`, positional embeddings `pos`.

```
keep = random_permutation(N)[: int(N * (1 - r))]     # visible patch indices
hidden = [i for i in range(N) if i not in keep]
h = enc(P[keep] + pos[keep])                         # encoder sees VISIBLE patches only
tokens = empty(N)
tokens[keep] = h
tokens[hidden] = mask_token                          # placeholders join in the decoder
pred = dec(tokens + pos)                             # predict every patch
loss = mean((pred[hidden] - P[hidden])**2)           # loss on HIDDEN patches only
```

Listing 27.11. C12 model answer.

Three points: encoder never sees mask tokens (cost, and no train/test mismatch); positions added so the decoder knows where each hole is; loss on hidden patches. (Chapter 18 runs this code.)

27.3 Representation learning

Check Your Understanding

C13. SimCLR's NT-Xent loss for a batch of N images, encoder f, projection head g, temperature τ.

```
x1, x2 = augment(x), augment(x)              # (N, ...) two views
z = normalize(g(f(cat([x1, x2]))))            # (2N, d)
S = z @ z.T / tau                             # (2N, 2N) cosine similarities / tau
S[diag] = -inf                                # a view is not its own negative
pos = cat([arange(N, 2N), arange(0, N)])      # partner of i is i+N (and vice versa)
loss = cross_entropy(S, pos)                  # softmax over the 2N-1 others
```

Listing 27.12. C13 model answer.

Points: the loss is a classification — "which of the other $2N-1$ is my partner" — with $2N-2$ negatives; features for downstream use are $f(x)$, before the head.

Check Your Understanding

C14. MoCo: one step with query encoder `f_q`, key encoder `f_k`, a queue of `K` keys, momentum `m`.

```
q = normalize(f_q(augment(x)))                # (b, d)
k = detach(normalize(f_k(augment(x))))        # (b, d) no gradient into keys
l_pos = sum(q * k, dim=1, keepdim=True)       # (b, 1)
l_neg = q @ queue.T                           # (b, K)  old keys as negatives
logits = cat([l_pos, l_neg], dim=1) / tau
loss = cross_entropy(logits, zeros(b))        # the positive is at index 0
sgd_step(loss)                                # updates f_q only
f_k.params = m * f_k.params + (1 - m) * f_q.params   # momentum update
queue = cat([queue, k])[-K:]                  # enqueue newest, dequeue oldest
```

Listing 27.13. C14 model answer.

Check Your Understanding

C15. SimSiam (and BYOL). Write one training step. What changes for BYOL?

```
z1 = proj(f(augment(x))); z2 = proj(f(augment(x)))     # shared encoder + projector
p1 = pred(z1);           p2 = pred(z2)                 # predictor
D = lambda p, z: -mean(sum(normalize(p) * normalize(detach(z)), dim=1))
loss = 0.5 * D(p1, z2) + 0.5 * D(p2, z1)               # stop-grad on the targets
sgd_step(loss)
```

Listing 27.14. C15 model answer.

BYOL: the targets come from a separate *target network* (encoder + projector) that is an EMA of the online one, instead of the same network with stop-gradient. Removing `detach` in SimSiam collapses immediately.

Check Your Understanding

C16. DINO, three variants. (a) Write the step (see §25.2.6). (b) Add multi-crop: 2 global and 6 local crops. (c) What happens if you delete the centering line? If you set the teacher temperature equal to the student's?

```
views = [global_crop(x), global_crop(x)] + [local_crop(x) for _ in range(6)]
t = [gt(v) for v in views[:2]]                # teacher: global crops only
s = [gs(v) for v in views]                    # student: all 8 crops
loss, n = 0, 0
for i, ti in enumerate(t):
    for j, sj in enumerate(s):
        if j == i: continue                   # never compare a view with itself
        loss += H(ti, sj); n += 1
loss = loss / n                               # 2*8 - 2 = 14 pairs
```

Listing 27.15. C16 (b) model answer: the multi-crop loss.

(c) Without centering, one output dimension comes to dominate for every image (collapse to a one-hot); the loss becomes low and the features useless. With equal temperatures (no sharpening), the outputs drift to the uniform distribution (the other collapse). Each mechanism alone is not enough; the pair balances (Chapter 22, Question 8).

Check Your Understanding

C17. Linear probe and kNN evaluation of a frozen encoder with a small labelled set.

```
F_tr = detach(encoder(X_tr)); F_te = detach(encoder(X_te))   # frozen features
W = linear(d, n_classes)                                     # the ONLY trained part
for epoch in range(E):
    sgd_step(cross_entropy(W(F_tr), y_tr))
acc_probe = mean(argmax(W(F_te), 1) == y_te)
# kNN: nothing is trained
S = normalize(F_te) @ normalize(F_tr).T                     # cosine similarities
nn = topk(S, k=20, dim=1)
acc_knn = mean(majority(y_tr[nn]) == y_te)
```

Listing 27.16. C17 model answer.

27.4 Self-training, cycles, world models, systems

Check Your Understanding

C18. Self-training with a confidence threshold, and the soft-target distillation loss.

```
teacher = train(X_lab, y_lab)
p = softmax(teacher(X_unl))                 # clean inputs
conf, y_hat = max(p, dim=1)
keep = conf >= threshold                    # e.g. 0.9, or top-40% by rank
student = train(cat([X_lab, X_unl[keep]]),
                cat([y_lab, y_hat[keep]]),
                weights=cat([ones, 0.5 * ones]), augment=strong)   # noisy student
# distillation instead of hard labels (T = temperature):
kd = T**2 * KL(softmax(teacher(x) / T) || softmax(student(x) / T))
```

Listing 27.17. C18 model answer.

Check Your Understanding

C19. CycleGAN: the generator loss for $G : X \to Y$, $F : Y \to X$, discriminators D_X, D_Y, cycle weight λ.

```
fake_y = G(x); fake_x = F(y)
adv = mean((D_Y(fake_y) - 1)**2) + mean((D_X(fake_x) - 1)**2)     # look real (LSGAN)
cyc = mean(abs(F(fake_y) - x)) + mean(abs(G(fake_x) - y))          # come back
loss_G = adv + lam * cyc
```

Listing 27.18. C19 model answer.

Check Your Understanding

C20. A latent world model and planning (DINO-WM style). Frozen encoder `E`, transitions `(o, a, o_next)`, predictor `P(z, a)`, goal image `o_goal`, horizon `H`.

```
# training: the next observation is the free target
for o, a, o_next in minibatches(D, b):
    z = detach(E(o)); z_next = detach(E(o_next))       # encoder frozen
    sgd_step(mean((P(z, a) - z_next)**2))

# planning with the cross-entropy method (CEM)
mu, sigma = zeros(H, d_a), ones(H, d_a)
z0, zg = E(o_now), E(o_goal)
for it in range(iters):
    A = mu + sigma * rand(n_samples, H, d_a)           # candidate action sequences
    z = repeat(z0, n_samples)
    for h in range(H): z = P(z, A[:, h])               # imagined rollout
    cost = sum((z - zg)**2, dim=-1)                    # distance to goal latent
    elite = A[argsort(cost)[:n_elite]]
    mu, sigma = mean(elite, 0), std(elite, 0)          # refit the sampling distribution
execute(mu[0])                                         # MPC: first action, then replan
```

Listing 27.19. C20 model answer: training and CEM planning.

Check Your Understanding

C21. Forward–backward consistency mask from flows `F_ab` and `F_ba` (each maps a pixel to its displacement).

```
for p in pixels:
    q = p + F_ab[p]                      # where p goes in frame B
    p_back = q + F_ba[round(q)]          # and back to frame A
    valid[p] = norm(p_back - p) < thr    # large error: occlusion or wrong flow
loss = sum(valid * photometric_error) / sum(valid)
```

Listing 27.20. C21 model answer.

Check Your Understanding

C22. ReLMM's grasp selection with an ensemble of M grasp predictors.

```
P = stack([G_m(obs, candidates) for G_m in ensemble])   # (M, n_candidates)
score = alpha * mean(P, 0) + beta * std(P, 0)           # exploit + explore
probs = softmax(score)                                   # proportional to exp(score)
g = candidates[sample_index(probs)]
r = execute_grasp(g)                                     # 1 if object in gripper
for G_m in ensemble:                                     # each trained on its own
    ↪ resample
    sgd_step(binary_cross_entropy(G_m(obs, g), r))
```

Listing 27.21. C22 model answer.

Key takeaway — the three things every coding answer needs

Shapes in comments (the examiner checks you know what flows where). *Where the gradient stops* (`detach`: VAE has none, SimSiam and DINO do, GAN's D step does). *The loss, written out* with its target named (noise not image; hidden patches not all; the partner view not all views).

CHAPTER 28

Exam Question Bank III: Case Studies and a Mock Exam

In this chapter

28.1 Design questions 937
28.2 Compare questions 941
28.3 Procedure questions 942
28.4 A complete mock exam (120 points, 120 minutes) 943

Chapter roadmap

What this chapter is. Eighteen scenario questions in the three shapes the first exam used, each with a model answer written the way it should be written in the exam: short headings, one or two sentences each, a formula where a loss is asked. At the end, a complete 120-point mock exam assembled from all four chapters of this bank, with a points table.

The three shapes.

- **Design** (like "SSL for robot object handling"): select a method, sketch the architecture, give the loss, state the assumptions, name the failure modes.
- **Compare** (like "image from a segmentation map"): discuss the suitability of four named methods and choose.
- **Procedure** (like "cluster images with about 6 categories"): which steps, which method first, and why.

Time. In the exam each is worth 10–15 points, i.e. 10–15 minutes. Practise with a timer; the model answers are about the length you can write in that time.

28.1 Design questions

Each task below asks for the same five things: *method, architecture, loss, assumptions, failure modes.*

Check Your Understanding

D1. Chest X-rays: flag unusual images. A hospital has 200,000 unlabelled chest X-rays, almost all normal, and wants to flag the rare abnormal ones for a radiologist. No abnormal examples are labelled.

Method. Anomaly detection by density: learn what *normal* looks like and flag images with low probability. Best: a normalising flow on SSL features (exact $\log p$), or a VAE (ELBO as an approximate score); a simpler baseline is kNN distance to the training set in DINO feature space. **Architecture.** Frozen SSL encoder (e.g. DINOv2 fine-tuned self-supervised on X-rays) $\to$ features $\to$ flow (coupling layers) $\to \log p(x)$. **Loss.** Maximum likelihood: $\mathcal{L} = -\big[\log p_Z(f(h)) + \log|\det \partial f/\partial h|\big]$ over normal images; at test time, flag if $\log p$ is below a threshold chosen for a target false-alarm rate. **Assumptions.** Training set is (almost) all normal; abnormalities change the image in ways the features capture; test images come from the same scanners. **Failure modes.** (1) *Likelihood favours simple images*: flows can give high $\log p$ to low-texture or overexposed images that are anomalous — score in feature space, not pixel space, and compare with a kNN score. (2) *Domain shift*: a new scanner makes every image "anomalous" — normalise intensities, recalibrate the threshold per site. (3) A pixel autoencoder with reconstruction error can *reconstruct anomalies too* if it is powerful enough, hiding them.

Check Your Understanding

D2. Satellite images of farmland. You have millions of unlabelled top-down satellite tiles and 500 tiles labelled with crop type. Design the pretraining.

Method. Contrastive or self-distillation pretraining (SimCLR, DINO) with augmentations adapted to aerial images, or MAE. *Not* RotNet: aerial images have no canonical "up", so rotation prediction is ill-posed (Tutorium 6, Question 5). **Architecture.** ViT or ResNet encoder, projection head, two views; for DINO an EMA teacher. **Loss.** InfoNCE $-\log \frac{e^{s(z_1,z_2)/\tau}}{\sum_k e^{s(z_1,z_k)/\tau}}$ or DINO's cross-entropy; positives from augmentations *including rotations and flips* (a field rotated is still the same field), and — a strong extra — the same location at *different dates* as a natural positive pair. Then a linear probe / fine-tune on the 500 labels. **Assumptions.** Crop type is invariant to rotation and to small shifts; different dates of one location share the crop (within a season). **Failure modes.** Colour jitter can destroy the spectral cue that separates crops — keep it weak; temporal positives across seasons teach invariance to growth stage, which may matter; false negatives: neighbouring fields of the same crop are pushed apart.

Check Your Understanding

D3. Camera traps. 1.5 million wildlife camera-trap images, fixed cameras, day and night (infrared), 40 species, 2,000 labelled images.

Method. SSL pretraining (DINO) on all images, then few-label classification with a linear probe or kNN, plus self-training on confident pseudo-labels. **Architecture.** ViT encoder; DINO with multi-crop (local crops focus on the animal); a linear head. **Loss.** DINO cross-entropy between teacher and student views; then cross-entropy on labels plus weighted pseudo-label cross-entropy. **Assumptions.** Species identity is invariant to crop, lighting, and day/night; the labelled set covers all 40 species. **Failure modes.** (1) *Background shortcut*: each camera sees one background, and most images from one camera show the same species, so features learn "which camera" — evaluate on held-out cameras; crop to detected animals. (2) *Empty frames* (triggered by wind) dominate — filter with a detector. (3) *Rare species* are dropped by a global confidence threshold (the week-9 notebook showed three classes getting no pseudo-labels at all) — per-class thresholds.

Check Your Understanding

D4. Industrial defect inspection. 80 labelled defect images, 20,000 unlabelled images from the line; defects are rare and small.

Method. Self-training (Noisy Student) on top of SSL features. **Architecture.** SSL-pretrained encoder (MAE is attractive here: small local defects need patch-level detail), classifier head; teacher and a slightly larger student. **Loss.** $\mathcal{L} = \mathrm{CE}(y_{\text{lab}}) + \lambda \sum_{i:\, c_i > \theta} w_i \,\mathrm{CE}(\hat{y}_i)$, student with strong augmentation. **Assumptions.** The teacher is better than chance on defects; defects look similar across the line. **Failure modes.** Confident mistakes on rare defects become training targets (confirmation bias); class imbalance — the threshold keeps almost only "no defect"; augmentations that hide small defects (big crops, blur) turn defects into "normal". Monitor accepted class counts and keep a labelled validation set (Tutorial week-9 quiz, Question 13).

Check Your Understanding

D5. A drone learns depth from its own flight videos. Monocular camera, no depth sensor, many videos of streets with cars and pedestrians.

Method. Self-supervised depth and ego-motion from video with a photometric loss, plus optical flow and consistency masks (SfMLearner / DF-Net, §14.4). **Architecture.** Depth network $D(I_t)$, pose network $T(I_t, I_{t+1})$, flow network $F(I_t, I_{t+1})$. **Loss.** $\mathcal{L} = \sum_p M(p)\,|I_t(p) - I_{t+1}(\pi(K\,T\,D(p)\,K^{-1}p))| + \lambda_s\,\mathcal{L}_{\text{smooth}} + \lambda_c\,\|F - F_{\text{rigid}}(D,T)\|$ over valid pixels M. **Assumptions.** Static scene except where masked; brightness constancy; textured surfaces; enough translation. **Failure modes.** Scale unknown (only relative depth; fix with GPS/IMU speed); moving objects break the rigid rule ("holes" at infinite depth) — mask via forward–backward and rigid-vs-flow inconsistency; oversmoothing if λ_s is too large (System quiz, Question 7); pure rotation gives no depth signal.

Check Your Understanding

D6. A legged robot learns where it can walk in a new forest, with no labelled terrain.

Method. Learning from own experience (WVN, §14.5): proprioception supplies the label. **Architecture.** Frozen DINO features per image segment; small MLP predicting traversability $\in [0, 1]$; an autoencoder on the features for familiarity. **Loss.** Where the robot walked (footprints projected into past images): $(\hat{t} - s)^2$ with s from how well the robot tracked its commanded velocity; plus reconstruction loss of the familiarity autoencoder; untouched segments get no label. **Assumptions.** Terrain that looks alike behaves alike; the robot's velocity tracking reflects traversability; camera–leg calibration is correct. **Failure modes.** Only walked-on terrain is labelled (positive-only data), so everything unfamiliar must be treated as uncertain; appearance-identical but different-behaving terrain (mud under leaves); dangerous terrain can never be "tried".

Check Your Understanding

D7. A car detects vehicles at night and in fog. The car has RGB, thermal and depth cameras and a microphone array.

Method. Cross-modal distillation (MM-DistillNet, §14.6): camera detectors are teachers, an audio student learns to detect from sound. **Architecture.** Pretrained detectors on RGB, thermal, depth; merge their boxes with NMS into pseudo ground truth; audio spectrograms $\rightarrow$ CNN student $\rightarrow$ boxes. **Loss.** Focal loss on the merged boxes + a multi-teacher attention-alignment term (KL between the student's attention and the product of the teachers'). **Assumptions.** Sensors are synchronised and calibrated; cars are audible; teachers are good in conditions where they are used for training. **Failure modes.** Teachers' shared mistakes are inherited; silent (electric, parked) cars cannot be heard; echoes in tunnels misplace sources; train only on moments when the camera teachers are reliable (daylight), test at night.

Check Your Understanding

D8. A household robot learns to pick up clutter by itself, with nobody resetting the room.

Method. Autonomous RL with self-supervised grasp labels and pseudo-resets (ReLMM, §14.7). **Architecture.** Ensemble of grasp-success predictors on wrist-camera images; navigation policy trained with SAC on a front-camera image. **Loss.** Grasp predictors: binary cross-entropy against $r_g \in \{0, 1\}$ from the gripper sensor; navigation reward $r_n = r_g - 1$ (or the predicted grasp success -1 with relabelling). Grasps chosen by α·mean + β·std. **Assumptions.** Grasp success is detectable (gripper closure/force); dropping an object somewhere random is a valid reset; the room stays safe. **Failure modes.** Resets not random enough (objects pile in one corner); unreachable objects stall learning; exploration of odd grasps breaks things; battery and stuck objects still need people.

Check Your Understanding

D9. A robot arm learns to push objects to a goal image, from a dataset of recorded random interactions without rewards.

Method. Latent world model plus planning (DINO-WM, §13.3). **Architecture.** Frozen DINOv2 patch features z_t; action-conditioned transformer predictor $z_{t+1} = P(z_t, a_t)$; CEM/MPC planner. **Loss.** Training: $\|P(z_t, a_t) - z_{t+1}\|^2$ (teacher forcing, frozen targets). Planning cost: $\|\hat{z}_{t+H} - z_{\text{goal}}\|^2$. **Assumptions.** The random data covers the states the task needs; distance in DINO space reflects task progress; dynamics are learnable one step at a time. **Failure modes.** Compounding error over long horizons; the planner exploits model errors (always check in reality, replan every step); goals that differ in a detail DINO features ignore (a small rotation) are not reached.

Check Your Understanding

D10. Speed up a large perception model (accurate, 300 ms per image) for a robot that needs 20 ms.

Method. Knowledge distillation into a small student on a large unlabelled set. **Architecture.** Teacher frozen; student 10–20× smaller. **Loss.** $T^2\,\mathrm{KL}(\sigma(t/T)\,\|\,\sigma(s/T)) + \alpha\,\mathrm{CE}(s, y)$ where labels exist; for dense tasks, also match intermediate feature maps through a small adapter. **Assumptions.** Unlabelled data from the robot's environment; the student has enough capacity. **Failure modes.** The student copies the teacher's errors; it cannot exceed capacity limits on hard cases; distribution shift if distillation data differ from deployment.

28.2 Compare questions

Check Your Understanding

P1. Anomaly scores for manufacturing images. Discuss the suitability of: a GAN; a VAE; a normalising flow; KDE on SSL features.

GAN: no likelihood and no encoder — no direct score (needs inversion tricks); unsuitable. **VAE**: approximate score (ELBO or reconstruction error), cheap; blurry reconstructions blur small defects; suitable baseline. **Flow**: exact $\log p$; best in principle, but in pixel space it favours low-complexity images; use on features. **KDE on SSL features**: simple, strong for small feature dimensions (after PCA), memory grows with the data; a very good practical choice. *Choice*: KDE/kNN on SSL features first, flow on features if more accuracy is needed.

Check Your Understanding

P2. Pretraining for a small medical data set (5,000 unlabelled CT slices, 300 labelled). Discuss: RotNet; SimCLR; MAE; DINO.

RotNet: CT slices have a canonical orientation, so the task is solvable, but it teaches orientation cues rather than pathology, and the last layers specialise (Tutorium 6, Q7). **SimCLR**: needs large batches (many negatives) and augmentation design; colour jitter meaningless for grey CT; false negatives among similar slices. **MAE**: works with small batches, learns local texture and anatomy, data-efficient; features need fine-tuning (weak linear probe). **DINO**: strong features, but tuned for natural images; smaller data sets need care (collapse, long training). *Choice*: MAE pretraining then fine-tuning on the 300 labels, or start from a DINOv2 backbone and continue self-supervised on the CT data.

Check Your Understanding

P3. Evaluate a new SSL encoder. Discuss: training loss; kNN; linear probe; fine-tuning.

Training loss: tells you the pretext is being solved, not that features are useful; blind to collapse. **kNN**: no training, tests neighbourhood structure, cheap; sensitive to feature scaling. **Linear probe**: tests linear separability; standard for comparing methods; can underrate MAE-like features. **Fine-tuning**: tests the weights as an initialisation; expensive; overfits with few labels. *Choice*: report kNN and linear probe first, fine-tuning last (week-8 quiz, Q15).

Check Your Understanding

P4. Negatives or not. For a team with one GPU and batch size 64, discuss SimCLR, MoCo, BYOL/SimSiam, and DINO.

SimCLR: its negatives come from the batch; 64 is far too small (126 negatives), accuracy drops. **MoCo**: the queue decouples negatives from batch size (65,536 keys) — fits one GPU. **BYOL/SimSiam**: no negatives at all, work with small batches; SimSiam simplest (no EMA). **DINO**: no negatives, small batches workable, multi-crop adds cost. *Choice*: SimSiam or MoCo; DINO if compute allows.

28.3 Procedure questions

Check Your Understanding

R1. Unknown number of categories. Ten thousand product images, the number of product types is unknown (between 10 and 100). (a) Which features? (b) How do you choose the number of clusters? (c) Which algorithm first?

(a) SSL features (DINO), L2-normalised, PCA to ≈128 dimensions. (b) Run k-means for a range of k and inspect the elbow of the objective or the silhouette score; better, label a small random sample and pick the k whose clusters are purest (Lecture 02's plant question). (c) DBSCAN or HDBSCAN is tempting because it needs no k, but on 10,000 points in 128

dimensions ε is hard to set; start with k-means *overclustering* (k=150), then merge clusters whose centroids are close — robust and cheap.

Check Your Understanding

R2. Curating a web-scale pretraining set. 1.2 billion web images and a small curated seed set. List the steps in order and justify each.

(1) Deduplicate (exact and near duplicates by embedding similarity) — saves compute, avoids memorising. (2) Filter unsafe/low-quality content. (3) Remove near-duplicates of evaluation sets — prevents leakage. (4) Embed with an SSL model. (5) Retrieve images near the curated seeds — keeps the pool relevant. (6) Hierarchical k-means and balanced sampling across clusters — rare concepts get their share (week-8 quiz, Q10 and Q14; the week-8 lab's cluster cap).

Check Your Understanding

R3. Your SSL run looks too good. The loss dropped to its minimum in two epochs. List the checks, in order.

(1) Per-dimension standard deviation of the embeddings across a batch (zero ⇒ collapse). (2) Singular values of the embedding matrix (few non-zero ⇒ dimensional collapse). (3) kNN accuracy on a small labelled set (chance ⇒ useless). (4) Check the anti-collapse code: stop-gradient present? EMA updating? centering applied? negatives not all from the same image? (5) Check for shortcuts: do the two views share something trivial (identical crops, same colour histogram, a watermark)?

Check Your Understanding

R4. Choosing augmentations for a new domain (microscope images of cells, where colour stains matter and orientation does not).

Keep: random rotations and flips (orientation irrelevant), crops (but not so small that cells vanish), small blur and noise. Avoid or keep weak: colour jitter and greyscale (they would teach invariance to the stain, which carries the information). Test: probe accuracy with and without each augmentation; augmentations are the inductive bias (Lecture 06).

28.4 A complete mock exam (120 points, 120 minutes)

Assembled from the bank in the proportions of the first exam. Answers are at the referenced places.

Part	Questions	Points	Minutes
1. MCQ	A1, A6, A11, A13, A18, A21, A25, A29, A32, A35; B1, B2, B4, B5, B6, B8, B10, B11, B12, B17, B18, B20, B21, B24, B25 (Chapter 26)	50	45
2. Coding	C2 mini-batch k-means with augmentations and labelling (12); C13 SimCLR loss (8) (Chapter 27)	20	20
3. Coding	C10 diffusion training step and sampler (10)	10	10
4. Design	D9 robot pushing with a world model (15)	15	15
5. Compare	P2 pretraining for a small medical data set (12)	12	12
6. Procedure	R1 unknown number of categories (13)	13	13
		120	**115**

Marking guide for the long questions. Design: method named and justified 3, architecture 3, loss with symbols 3, assumptions 3, failure modes with fixes 3. Compare: 2–3 per option (can it, strength, weakness), 2 for a justified choice. Coding: structure 50%, loss and target 30%, stop-gradients/shapes 20%. Procedure: each correct step with its reason.

Key takeaway — what to write on the two A4 sheets

Sheet 1: the formula sheet of §26.4. Sheet 2: the three coding templates the exam asked for (VAE, batch k-means with augmentation, DINO step), the five-heading design skeleton, and one line per method family on "where does the supervision come from" (§24.1).

Part V

Exercises and Solutions

CHAPTER 29

Exercises and Solutions

In this chapter

Part A: The Exercise Bank 950
29.1 A.0 The course's own example exam questions (Lecture 07, slides 2–5) 950
29.2 A.1 Chapter 1 — Why self-supervised learning? 952
29.3 A.2 Chapter 2 — Technical foundations 953
29.4 A.3 Chapter 3 — Clustering and classical–SSL hybrids 955
29.5 A.4 Chapter 4 — Autoencoders and VAEs 957
29.6 A.5 Chapter 5 — Masked autoencoders 959
29.7 A.6 Chapter 6 — Flows and GANs 960
29.8 A.7 Chapter 7 — Denoising, diffusion, flow matching 961
29.9 A.8 Chapter 8 — Pretext tasks and augmentations 963
29.10 A.9 Chapter 9 — Contrastive learning 965
29.11 A.10 Chapter 10 — Beyond negatives 966
29.12 A.11 Chapter 11 — Self-distillation and the DINO family 968
29.13 A.12 Chapter 12 — Self-training, distillation, cycle consistency 970
29.14 A.13 Chapter 13 — World models 971
29.15 A.14 Chapter 14 — System-level SSL 973
29.16 A.15 Chapter 15 — Why does SSL work? 974
29.17 A.16 Chapter 16 — Comparing the families 975
Part B: How to write a strong exam answer . 978 • B.1 Exemplar answer — question 1 (the DBSCAN multiple choice) • B.2 Exemplar answer — question 2 (the plant case study) • B.3 Exemplar answer — question 3 (deep clustering code completion)
Part C: Capstone problems 983

Chapter roadmap

What this chapter is. Every exercise in the book, with worked answers — not sketches. If you can do these, you can sit the exam.

How to use it, depending on why you are here.

- *Reading a chapter and want to check you got it.* Do that chapter's Level 1 and Level 2 questions in Part A, immediately after finishing the chapter. They take

a few minutes each and they are designed to catch the specific misunderstanding that chapter invites.
- *Revising for an exam.* Go to Part B first. It takes the course's own example exam questions and writes out full-credit answers, with the answer *pattern* made explicit — what an examiner is actually rewarding. Then work backwards through Part A's Level 3 and 4 questions.
- *Want to know whether you really understand the field.* Part C: five integrative problems, each requiring you to combine methods from several chapters. These are the ones worth struggling with.

How to actually do them. Write your answer before reading the solution — including the wrong parts. The worked answers are written to be read *after* an honest attempt, and several of them are more useful for the mistakes they discuss than for the answer they give.

If you get stuck. Every exercise names the section it comes from. Go back and re-read that section rather than reading the answer; the exercises are deliberately placed so that the relevant explanation is one cross-reference away.

Key takeaway — the four-part answer pattern this course rewards

Almost every exam-style question in this field can be answered with the same four moves, in this order. **(1) Intuition:** what is the one-sentence bet the method makes? **(2) Mechanism:** what actually happens to the data, step by step? **(3) Objective:** what exactly is minimised, and what does it reward versus punish? **(4) Evidence:** what experiment would show the method works, and what would show it failing? An answer that does all four scores full marks even when it is brief. An answer that does only (2) — reciting the architecture — rarely does, however detailed.

This chapter collects the book's exercises with *complete worked answers*. It is organized in three parts. Part A is a full exercise bank ordered by chapter: it opens with the course's own example exam questions (Lecture 07, slides 3–5, paraphrased) and then adds new exercises for every chapter of Parts I–III. Part B is a coach's corner: we take the three exam questions from Lecture 07 and write out full-credit exemplar answers, making explicit the answer pattern this course's exam rewards. Part C gives five integrative capstone problems that require combining methods from several chapters.

All exercises are graded by difficulty:

- **Level 1** — recall of definitions and mechanisms.
- **Level 2** — apply a definition or algorithm in a new setting.
- **Level 3** — compute something concrete by hand.
- **Level 4** — reason quantitatively about a method's behavior.
- **Level 5** — design or analyze at research depth.

Numerical answers are worked out in full; if your arithmetic deviates in the last decimal place, your method is probably right and ours merely noisier. Labels of the form `ex:...` allow chapters to point readers here.

Part A: The Exercise Bank

The bank follows the book's chapter order. Each block gives the questions first and the full worked solutions immediately after, mirroring the order of the questions. The three exam-style questions below are the course's own warm-up: they were shown in Lecture 07 (slides 2–5) as illustrations of the exam format — a knowledge multiple-choice question, a case study, and a code-completion question on paper. Their content deliberately reviews Lecture 02's clustering material, so they sit naturally before the per-chapter blocks.

29.1 A.0 The course's own example exam questions (Lecture 07, slides 2–5)

The instructor opened Lecture 07 with exam logistics (a 120-minute written exam; two handwritten A4 sheets allowed; format: knowledge questions, case studies, coding on paper) and then gave three example questions. Exemplar *full-credit answers* to exactly these three questions are developed in Part B; here we state the questions and give direct solutions.

Check Your Understanding

Exam-style question 1 (knowledge, multiple choice, Level 1). Which of the following statements about DBSCAN is wrong?

(A) DBSCAN does not require a given number of clusters.
(B) DBSCAN is a density-based clustering method.
(C) DBSCAN scales efficiently with the dataset size.
(D) DBSCAN can detect outliers.

Worked Example — Direct answer to exam-style question 1

Statement (C) is wrong. (A) is true: DBSCAN discovers the number of clusters from the data, which is its main selling point over k-means. (B) is true by definition: clusters are dense regions separated by sparse regions, built from ε-neighborhoods and the min_samples core-point criterion. (D) is true: points that are neither core points nor within reach of a core point are explicitly labeled as noise/outliers — an outlier mechanism k-means does not have. (C) is the trap: DBSCAN reasons over neighborhoods of *all pairs of points*; the naive algorithm needs the $n \times n$ affinity/neighborhood structure, so memory grows as $O(n^2)$ and naive time is quadratic unless a spatial index exists. In practice it is exactly the method you must scale with tricks (OPTICS as the memory-friendlier variant, PCA or subsampling to exemplars, sparse neighborhood storage), whereas k-means can be approximated in minibatches with constant memory.

Check Your Understanding

Exam-style question 2 (case study, Level 4). You are given a dataset of 2,000,000 images of plants and are asked to categorize them into species. You do not know how

many species occur in the dataset, and you are not an expert on plants. A colleague has agreed to label 2000 images for you. Describe a method to categorize the plants with high accuracy.

Worked Example — Solution to exam-style question 2 (summary)

The intended shape of an answer: (i) learn visual features on *all* 2,000,000 images with a label-free objective (a joint-embedding or pretext method such as SimCLR- or DINO-style training; at minimum, plain deep features), because your label budget of 2000 covers at most 0.1% of the data and never the unknown species count directly; (ii) cluster the resulting feature vectors — the number of species is unknown, which argues for a density method or for clustering over a range of k and validating with the labels (see below); at this scale apply PCA first and use minibatch k-means or exemplar-based pre-clustering, since naive DBSCAN on 2×10^6 points needs an $\sim n^2$ neighborhood structure; (iii) spend the 2000 labels to *calibrate and name* the structure: check which clustering granularity aligns with labeled species, then assign each cluster the majority species of its labeled members; (iv) propagate cluster labels to all images (pseudo-labeling), optionally training a classifier on the pseudo-labels in the DeepCluster loop, and finally (v) evaluate with a held-out portion of the labeled set (e.g. a kNN or linear classifier on frozen features). A complete exemplar in exam-register prose, with definitions, assumptions, and a failure mode, is Part B, answer 2.

Check Your Understanding

Exam-style question 3 (code completion on paper, Level 3). The following template must be completed to implement the deep clustering algorithm. `encoder` is a neural network mapping an image $\boldsymbol{x}$ to features.

```
for epoch in range(num_epochs):
    ...
    for x in minibatches(X):
        # x: input minibatch, shape (batch_size, input_dim)
        ...
        loss = ...
        sgd(loss).backwards()
```

Worked Example — Solution to exam-style question 3

DeepCluster alternates, once per epoch, between (1) computing features for the whole dataset with the *current* encoder, reducing their dimension with PCA, and clustering them with k-means; and (2) treating the resulting cluster indices as pseudo-labels for a full classification training epoch.

```
for epoch in range(num_epochs):
    # --- clustering step: turn current features into pseudo-labels ---
    feats = encoder(all_images)          # (n_images, feature_dim)
    feats = pca(feats, output_dim=256)   # decorrelate + compress
    centroids, pseudo = kmeans(feats, k) # pseudo: per-image cluster id

    # --- network update: classify each image into its cluster ---
    for x, y in minibatches(X, pseudo):
        # x: input minibatch, shape (batch_size, input_dim)
        logits = classifier_head(encoder(x))
        loss = cross_entropy(logits, y)
        sgd(loss).backwards()
```

The points that earn credit: features used for clustering come from the network as it was *before* the inner loop (re-extract them every epoch, because the encoder changes); PCA precedes k-means; the pseudo-label is the cluster index; the loss is plain cross-entropy between the predicted cluster distribution and the pseudo-label. Why this learns anything at all: the initial features, even random or weakly trained, cluster *better than chance*; the better-than-chance assignment is exploited as a learning signal, and each round of classification improves the features that produced the assignment. Failure mode to name: collapse — the network can obtain zero classification loss by predicting constant logits if k-means produces degenerate (e.g. nearly single-cluster) assignments; DeepCluster-style pipelines guard against this with balanced re-sampling and re-clustering.

29.2 A.1 Chapter 1 — Why self-supervised learning?

Check Your Understanding

1.1 (Level 1) Define *pretext task* and *downstream task*, and state the central dogma of self-supervised learning in one sentence. Give one vision example of a pretext task whose downstream value is not the pretext ability itself.

Worked Example — Solution 1.1 — pretext vs. downstream

A *pretext task* is a supervised task constructed automatically from unlabeled data: the target of the prediction is computed from the input by a fixed procedure (rotation angle, masked word, missing pixels), so no human labeling is involved. A *downstream task* is the task one actually cares about — classification, detection, segmentation, retrieval — for which few or no task-specific labels may exist. The central dogma: *structure present in the raw data can be converted into a training signal without labels* — if data carries exploitable regularities, any predictor that is better than chance at a pretext task has extracted some of that structure, and that structure transfers. Example: colorization (predict the colors of a grayscale image). Nobody needs automatic colorization of gray

images as a product; its value is that solving it forces the network to recognize objects and materials, so the learned features serve downstream as generic visual representations.

29.3 A.2 Chapter 2 — Technical foundations

Check Your Understanding

2.1 (Level 2) Let $\boldsymbol{u} = (1, 0)$, $\boldsymbol{v} = (1, 1)$, $\boldsymbol{q} = (-1, 0)$. Compute all pairwise Euclidean distances and all pairwise cosine *distances* ($d_{\cos} = 1 - \text{sim}(\cdot, \cdot)$). Do the two metrics rank-closest any pair differently? Explain what each metric is sensitive to.

2.2 (Level 3) A hash function h is $(r, 4r, 0.9, 0.1)$-sensitive for a distance d: pairs within r collide with probability at least 0.9, pairs beyond $4r$ with probability at most 0.1. You build $L = 5$ independent hash tables. (a) For a query whose true neighbor is within distance r, what is the probability that *no* table contains it in the query's bucket? (b) The dataset contains 1000 points farther than $4r$ from the query; how many far points do you expect to distance-check in one table's bucket? Why does increasing L trade off recall against query cost?

2.3 (Level 4) Run Lloyd's algorithm on the 1D dataset $X = \{0, 1, 2, 10, 11, 12\}$ with $k = 2$, initialized at $\mu_1 = 0$, $\mu_2 = 2$ (break distance ties toward μ_1). Show every reassignment and update until convergence, and compute the final within-cluster sum of squares (WCSS). Then argue why k-means++ would almost never pick this bad an initialization.

Worked Example — Solution 2.1 — two notions of distance

Writing d_E for Euclidean distance:

$$d_E(\boldsymbol{u}, \boldsymbol{v}) = \|(0, -1)\| = 1, \qquad d_E(\boldsymbol{u}, \boldsymbol{q}) = \|(2, 0)\| = 2,$$
$$d_E(\boldsymbol{v}, \boldsymbol{q}) = \|(2, 1)\| = \sqrt{5} \approx 2.236.$$

For cosine similarity, $\text{sim}(\boldsymbol{u}, \boldsymbol{v}) = \frac{1}{1 \cdot \sqrt{2}} \approx 0.707$, $\text{sim}(\boldsymbol{u}, \boldsymbol{q}) = \frac{-1}{1 \cdot 1} = -1$, and $\text{sim}(\boldsymbol{v}, \boldsymbol{q}) = \frac{-1}{\sqrt{2} \cdot 1} \approx -0.707$, so the cosine *distances* are 0.293, 2.0, and 1.707 respectively. In this toy both metrics agree on the ordering: $\boldsymbol{u}$–$\boldsymbol{v}$ closest, $\boldsymbol{u}$–$\boldsymbol{q}$ farthest. But they measure different things and can disagree: Euclidean distance penalizes both direction and magnitude differences, whereas cosine distance ignores magnitude entirely — $(1, 1)$ and $(100, 100)$ are at cosine distance 0 but Euclidean distance $99\sqrt{2}$. This is why the lecture recommends checking which distance a network's loss actually optimizes: features trained with a cosine/InfoNCE objective are normalized and spread on a hypersphere, making cosine the natural retrieval metric; on unnormalized features, Euclidean distance silently gives large-norm samples more "pull".

Worked Example — Solution 2.2 — LSH arithmetic

(a) One table misses the true neighbor with probability at most $1 - 0.9 = 0.1$. Tables are built from independent random hash constructions, so all five miss with probability at most $0.1^5 = 10^{-5}$; recall is therefore at least 0.99999. Independent constructions are essential — five copies of the same hash function give no improvement at all. (b) Each far point collides with the query in a given table with probability at most 0.1, so the expected number of far points in one bucket is at most $0.1 \times 1000 = 100$. Distance-checking is exact, so false candidates only cost time, never recall. (c) More tables multiply the recall (10^{-5} failure here) but each table adds its own bucket of expected ≤ 100 candidates: query cost grows linearly in L while the miss probability shrinks geometrically. This recall–throughput trade-off is exactly what the ann-benchmarks curves of Lecture 02 (slide 12) visualize for real libraries.

Worked Example — Solution 2.3 — Lloyd by hand

Round 1 (centers $\mu_1 = 0$, $\mu_2 = 2$). Distances: 0 is at distance 0 vs. 2; 1 ties at distance 1 (goes to μ_1 by our tie rule); 2 is at distance 2 vs. 0; $10, 11, 12$ are nearer to $\mu_2 = 2$. Assignments $\{0, 1\}$, $\{2, 10, 11, 12\}$. New centers $\mu_1 = \frac{0+1}{2} = 0.5$ and $\mu_2 = \frac{2+10+11+12}{4} = \frac{35}{4} = 8.75$. **Round 2.** Distances of each point to $(0.5, 8.75)$: $0 \to (0.5, 8.75)$: $d = (0.5, 8.75)$ so cluster 1; 1: $(0.5, 7.75) \to$ cluster 1; 2: $(1.5, 6.75) \to$ cluster 1; 10: $(9.5, 1.25) \to$ cluster 2; likewise 11, 12. Assignments $\{0, 1, 2\}, \{10, 11, 12\}$. New centers $\mu_1 = 1$, $\mu_2 = 11$. **Round 3.** 2 is at distances (1, 9); 10 at (9, 1): no assignment changes, so Lloyd has converged with centers $(1, 11)$. Final WCSS:

$$(0-1)^2 + (1-1)^2 + (2-1)^2 + (10-11)^2 + (11-11)^2 + (12-11)^2 = 4.$$

Could a better split exist? The only other sane 2-partition of six collinear points is $\{0, 1\}, \{2, 10, 11, 12\}$, whose WCSS is $\frac{1}{2} + (2 - 8.75)^2 \cdots$ — round 1 already showed how much worse that is; the global optimum is indeed 4. Why k-means++ avoids our bad start: after drawing the first seed at random, it draws each subsequent seed with probability proportional to the *squared* distance to the nearest existing seed ($D(x)^2$ seeding). If the first seed is 0, the squared distances of the remaining points are $1, 4, 100, 121, 144$ (normalizer 370), so the second seed lands in the right cluster with probability $(100 + 121 + 144)/370 \approx 0.9865$. The initialization we used (0 and 2, both in the left cluster) is precisely the kind that $D(x)^2$ seeding assigns vanishing probability.

29.4 A.3 Chapter 3 — Clustering and classical–SSL hybrids

Check Your Understanding

3.1 (Level 2) Lecture 02's boxed claim was: "anything better than chance can be exploited as a learning signal." Explain in your own words how DeepCluster instantiates this claim, and what precisely "chance" means in its inner loop.

3.2 (Level 3) DeepCluster trains a classifier to predict k-means cluster indices. Name two distinct degenerate solutions that achieve (near-)zero classification loss without learning useful features, and for each, one mechanism that counters it.

3.3 (Level 5) A CRF labels three grayscale pixels in a row. Unary costs ψ_u (cost of assigning each label to each pixel) are

	x_1	x_2	x_3
label A	0.5	0.5	1.0
label B	1.0	1.0	0.4

and adjacent pixels have similar intensities, giving pairwise cost $\psi_p = 2$ whenever two neighbors take *different* labels (and 0 otherwise). The labeling energy is $E(x) = \sum_i \psi_u(x_i) + \sum_{i<j \text{ adj.}} \psi_p(x_i, x_j)$. (a) Which label does the unary term alone assign to each pixel? (b) Find the global energy minimum over all 8 labelings. (c) State in one sentence what qualitatively happened, and what happens if the pairwise weight is raised far above the unary differences.

Worked Example — Solution 3.1 — the above-chance thesis

"Chance" here is specific: with k clusters, blindly guessing a random image's cluster index succeeds with probability $1/k$. Even a randomly initialized convolutional network produces features with some structure — neighboring pixels, edges, and textures create non-random neighborhoods — so k-means in that feature space recovers groups that are non-trivially better aligned with image content than a $1/k$ guess. DeepCluster exploits exactly this residual signal: (1) extract features from all images with the current encoder, (2) compress with PCA, (3) cluster with k-means, (4) treat cluster indices as pseudo-labels and train the network by cross-entropy to predict them. Each cycle sharpens the features so the next clustering is better than the last. The thesis is the rate-limiting step of the whole course: no method needs *correct* supervision; it needs supervision that is statistically better than random guessing, because gradient descent amplifies small systematic advantages.

Worked Example — Solution 3.2 — DeepCluster pathologies

Collapse to a single (or few) clusters. k-means on pathological features may dump almost every image into one cluster; the pseudo-labels are then nearly constant, and predicting a constant distribution gives near-zero loss with no semantic content.

Countermeasures: balanced re-sampling of images by cluster when building training batches (so rare clusters still influence gradients), and re-clustering each epoch so the pathology does not freeze in. **Feature collapse between rounds.** Nothing forces the encoder to keep distinct images distinguishable: an encoder mapping everything to one point makes subsequent clusterings meaningless while the previous epoch's classification loss still looked fine. Countermeasures: the alternating structure itself helps (targets are regenerated from data, not from the network's own predictions), PCA before clustering prevents a single dominant degenerate direction, and successors of DeepCluster replace offline k-means by online clustering with explicit balancing or degeneracy-avoiding losses (e.g. SwAV's balanced assignment). The general lesson — every pseudo-label method must explain what stops the target distribution from going trivial — returns for self-training in Chapter 12 and for center/sharpness tricks in Chapter 11's DINO.

Worked Example — Solution 3.3 — CRF energy minimization

(a) Per pixel, the cheaper unary is the row minimum: $x_1 = A$ ($0.5 < 1$), $x_2 = A$, $x_3 = B$ ($0.4 < 1$). Unary alone gives AAB with unary cost $0.5 + 0.5 + 0.4 = 1.4$. (b) Evaluate all eight labelings, adding 2 for each of the two edges that straddles a label change. Write E = unary + pairwise:

labeling	unary	pairwise	E
AAA	$0.5 + 0.5 + 1.0 = 2.0$	0	**2.0**
AAB	1.4	2	3.4
ABA	$0.5 + 1.0 + 1.0 = 2.5$	4	6.5
ABB	$0.5 + 1.0 + 0.4 = 1.9$	2	3.9
BAA	$1.0 + 0.5 + 1.0 = 2.5$	2	4.5
BAB	$1.0 + 1.0 + 0.4 = 2.4$	4	6.4
BBA	$1.0 + 1.0 + 1.0 = 3.0$	2	5.0
BBB	$1.0 + 1.0 + 0.4 = 2.4$	0	2.4

The global minimum is AAA at $E = 2.0$: pixel 3 is relabeled to A even though its unary prefers B by 0.6, because switching it costs 2 at the x_2–x_3 boundary. (c) The pairwise term performs *spatial smoothing*: weak unary evidence is overruled to keep coherent regions coherent, which is precisely why AAA (and not unary-optimal AAB) wins — neighboring pixels of similar appearance should share labels. If the pairwise weight is raised far above the unary differences, the CRF approaches hard segmentation into single-label regions (AAA and BBB dominate everything, and texture/detail evidence in the unaries is ignored) — the segmentation analog of oversmoothing. This is the same energy structure that the STEGO objective borrows: correspondences in backbone features play the role of the unaries, and the learned low-dim similarities play the role of the pairwise term.

29.5 A.4 Chapter 4 — Autoencoders and VAEs

Check Your Understanding

4.1 (Level 1) State the purpose of the bottleneck in an autoencoder, and why an autoencoder *without* any bottleneck or regularization may learn a representation that reconstructs perfectly yet is useless.

4.2 (Level 3) A VAE has prior $p(\boldsymbol{z}) = \mathcal{N}(0, I)$ and diagonal-Gaussian encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2))$. Show that the KL term of the ELBO has the closed form

$$D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, \mathcal{N}(0, I)) = \tfrac{1}{2} \textstyle\sum_{j=1}^{d_z} \left(\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1 \right).$$

Which values of (μ_j, σ_j^2) minimize it, and what does each summand do?

4.3 (Level 1) Why can backpropagation not pass through the step $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\phi}}, \mathrm{diag}\, \boldsymbol{\sigma}_{\boldsymbol{\phi}}^2)$? State the reparameterization trick, and give $\partial \boldsymbol{z} / \partial \boldsymbol{\mu}$ and $\partial \boldsymbol{z} / \partial \boldsymbol{\sigma}$.

4.4 (Level 2) Compare the score-function (REINFORCE) estimator with the reparameterization estimator: are they biased, how large is their variance, and what does each require of f and of $q_{\boldsymbol{\phi}}$? Explain in one sentence *why* one has lower variance.

4.5 (Level 3) Let $z \sim \mathcal{N}(\mu, \sigma^2)$ and $f(z) = z^2$. (a) Compute $\partial \mathbb{E}[f] / \partial \mu$ exactly. (b) Show that both the reparameterization estimate and the score-function estimate of it are unbiased. (c) At $\mu = \sigma = 1$, compute the variance of each one-sample estimate. (d) Why does the trick fail if z is discrete, and what can be used instead?

Worked Example — Solution 4.1 — why the bottleneck is the method

The bottleneck (dimension $d_z \ll d$, or an equivalent information constraint such as a codebook, sparsity penalty, or the KL term of a VAE) forces the encoder to discard information: it cannot copy the input, so only structure that is statistically shared across many inputs — objects, geometry, illumination patterns — is worth spending the limited capacity on. Without any bottleneck, an autoencoder satisfies its objective by the identity: reconstruction loss is zero and the "representation" is a lookup of the input itself. That representation reconstructs perfectly and compresses or generalizes not at all — the memorization failure mode the lecture illustrates with the codebook-learning example. The bottleneck is what turns compression into learning.

Worked Example — Solution 4.2 — the Gaussian KL

The KL between two Gaussians factorizes over independent coordinates, so fix one coordinate with $q = \mathcal{N}(\mu, \sigma^2)$ against $p = \mathcal{N}(0, 1)$ and recall $D_{\mathrm{KL}}(q \| p) = \mathbb{E}_q[\log q - \log p]$:

$$\log q(z) - \log p(z) = \left[-\tfrac{1}{2} \log(2\pi\sigma^2) - \tfrac{(z-\mu)^2}{2\sigma^2} \right] - \left[-\tfrac{1}{2} \log(2\pi) - \tfrac{z^2}{2} \right] = -\tfrac{1}{2} \log \sigma^2 - \tfrac{(z-\mu)^2}{2\sigma^2} + \tfrac{z^2}{2}.$$

Taking expectations under q, using $\mathbb{E}_q[(z-\mu)^2] = \sigma^2$ and $\mathbb{E}_q[z^2] = \sigma^2 + \mu^2$:

$$D_{\mathrm{KL}} = -\tfrac{1}{2}\log\sigma^2 - \tfrac{1}{2} + \tfrac{\sigma^2+\mu^2}{2} = \tfrac{1}{2}(\mu^2 + \sigma^2 - \log\sigma^2 - 1),$$

and summing over $j = 1..d_z$ gives the claimed formula. Each summand $\phi(\mu, \sigma^2) = \tfrac{1}{2}(\mu^2 + \sigma^2 - \log\sigma^2 - 1)$ is minimized uniquely by $\mu_j = 0$, $\sigma_j^2 = 1$ (check: differentiating in σ^2 gives $1 - 1/\sigma^2 = 0$, and $\log\sigma^2 \le \sigma^2 - 1$ proves $\phi \ge 0$ with equality only at $(0, 1)$). Intuitively the term has two forces: μ_j^2 pushes every latent code toward the origin, and $\sigma_j^2 - \log\sigma_j^2$ pushes each posterior variance toward 1 — preventing both exploding spread and collapse to deterministic points. The KL therefore acts as a *regularizer* that keeps the aggregate posterior organized like the unit Gaussian, while the reconstruction term of the ELBO pulls codes apart enough to identify $\boldsymbol{x}$.

Worked Example — Solution 4.3 — why the sample blocks backprop

The ELBO's reconstruction term is $\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})]$, and $\boldsymbol{\phi}$ sits in the *distribution* being sampled, not in the function being averaged. A drawn value such as $z = 0.73$ has no derivative with respect to $\boldsymbol{\mu}_{\boldsymbol{\phi}}$: changing $\boldsymbol{\mu}$ changes which values are *likely*, not the value already drawn. Pushing the gradient inside the expectation is wrong, $\nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}}[f] \neq \mathbb{E}_{q_{\boldsymbol{\phi}}}[\nabla_{\boldsymbol{\phi}} f]$ — the right side is zero.

The trick writes the sample as $\boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}} \odot \boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. This has the same distribution, but the randomness is now a parameter-free input, so $\nabla_{\boldsymbol{\phi}}\mathbb{E}_{\boldsymbol{\epsilon}}[f(\boldsymbol{z})] = \mathbb{E}_{\boldsymbol{\epsilon}}[\nabla_{\boldsymbol{\phi}} f(\boldsymbol{z})]$ and ordinary backprop works. The derivatives are $\partial\boldsymbol{z}/\partial\boldsymbol{\mu} = \mathbf{1}$ and $\partial\boldsymbol{z}/\partial\boldsymbol{\sigma} = \boldsymbol{\epsilon}$ (element-wise).

Worked Example — Solution 4.4 — score function versus reparameterization

	Score function	**Reparameterization**
estimate	$f(\boldsymbol{z})\,\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\boldsymbol{z})$	$\nabla_{\boldsymbol{\phi}} f(\boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}} \odot \boldsymbol{\epsilon})$
unbiased	yes	yes
variance	high	low
f must be differentiable	no	yes
q must be reparameterizable	no	yes (e.g. Gaussian)

Why: the score-function estimate uses only the *value* of f, multiplied by a random direction, while the reparameterization estimate uses the *slope* $\nabla_{\boldsymbol{z}} f$ — it knows which way to move.

Worked Example — Solution 4.5 — both estimators by hand

(a) $\mathbb{E}[z^2] = \mu^2 + \sigma^2$, so $\partial\mathbb{E}[f]/\partial\mu = 2\mu$.

(b) Write $z = \mu + \sigma\epsilon$. *Reparameterization:* the estimate is $f'(z) \cdot \partial z/\partial\mu = 2z$, and $\mathbb{E}[2(\mu + \sigma\epsilon)] = 2\mu$. ✓ *Score function:* $\partial_\mu \log q(z) = (z - \mu)/\sigma^2 = \epsilon/\sigma$, so the estimate is $z^2\epsilon/\sigma$, and

$$\mathbb{E}\left[\frac{(\mu + \sigma\epsilon)^2\,\epsilon}{\sigma}\right] = \frac{\mu^2\mathbb{E}[\epsilon] + 2\mu\sigma\mathbb{E}[\epsilon^2] + \sigma^2\mathbb{E}[\epsilon^3]}{\sigma} = \frac{0 + 2\mu\sigma + 0}{\sigma} = 2\mu. \checkmark$$

(c) At $\mu = \sigma = 1$, with $\mathbb{E}[\epsilon^2] = 1$, $\mathbb{E}[\epsilon^4] = 3$, $\mathbb{E}[\epsilon^6] = 15$ and odd moments zero: *reparameterization* $\mathrm{Var}(2+2\epsilon) = 4$; *score function* $\mathbb{E}[(1+\epsilon)^4\epsilon^2] - 2^2 = (1+6\cdot3+15) - 4 = 30$. The score-function estimate is 7.5× noisier (for σ the same calculation gives 136 versus 12).

(d) For discrete z there is no smooth map from $\boldsymbol{\phi}$ to the sample: a small change either leaves the choice unchanged (gradient 0) or flips it (a jump). Use a Gumbel-Softmax / Concrete relaxation, a straight-through estimator, or REINFORCE with a baseline.

29.6 A.5 Chapter 5 — Masked autoencoders

Check Your Understanding

5.1 (Level 2) MAE's recipe masks 75% of ViT patches before the encoder. A 224×224 image with 16×16 patches yields 196 patches. (a) How many patches does the MAE encoder see per image, and how does that roughly affect encoder compute per image compared to processing all patches (attention cost $O(N^2)$ in the number N of tokens)? (b) Why does hiding most of the input *help* learning, given that inpainting methods mask only a modest region?

5.2 (Level 4) MAE reaches 84.9% top-1 on ImageNet with finetuning (ViT-L), but its linear-probe accuracy is lower than DINO-family models'. Explain, in terms of what each objective forces the representations to keep, why *reconstruction*-based pretraining lags in frozen-feature linear probes while favoring finetuned transfer.

Worked Example — Solution 5.1 — the 75% design

(a) The encoder sees $196 \times 0.25 = 49$ patches. Quadratic attention cost scales as N^2, so the heavy part of the encoder drops to roughly $(49/196)^2 = 1/16 \approx 6\%$ of the full-token cost; linear terms (patch embeds, MLPs, biases) drop to 25%. The discarded 147 positions enter only in the small decoder as learned mask tokens. This asymmetry is what makes 75% masking affordable, not just decorative. (b) Masking most of the image destroys the trivial solution. With a small mask, low-level statistics suffice: a missing block of sky is completed by copying neighboring sky — no semantics acquired. At 75% masking the visible patches are too sparse for local interpolation to work well, so minimizing the reconstruction error on the held-out parts requires knowledge of objects, layout, and

regularities of the visual world — exactly the generic understanding we hope to transfer. Heavy masking converts a low-level completion task into a holistic recognition task; the lecture's masking-ratio curves show accuracy peaking around this regime precisely because the task ceases to be solvable by shortcut statistics.

Worked Example — Solution 5.2 — reconstruction vs. joint-embedding features

A linear probe with frozen features can only sense properties that vary *linearly* across the feature space at its output. DINO/SimSiam-style joint-embedding objectives are *explicitly* about invariances: their loss pushes two augmented views of one image to the same code, so the learned feature varies smoothly with semantic content and is inspected well by a linear classifier — that linear separability *is* the objective. MAE's objective instead measures per-pixel reconstruction; the features it must produce are interface features to the decoder — rich in spatial and photometric detail to fill in masked pixels. Detail-rich features are good for spatial transfer after *finetuning*, but a substantial part of their variance (illumination, texture, exact layout) is irrelevant to class labels and not linearly organized around them; a frozen linear probe therefore underperforms one trained on invariance-shaped features. Finetuning closes most of the gap because the decoder-readout detail is repurposed by a few epochs of task gradients, which is why MAE is the stronger finetuning starting point (e.g. 84.9% ViT-L) and DINO-style features the stronger frozen ones. Neither verdict is absolute: this is a statement about which *teleology* the loss function encodes, verified empirically in the course's comparison tables, not a theorem.

29.7 A.6 Chapter 6 — Flows and GANs

Check Your Understanding

6.1 **(Level 3)** In dimension 1, let $z \sim \mathcal{N}(0,1)$ and $x = f(z) = \mu + sz$ with $s > 0$. Derive $p_X(x)$ from the change-of-variables formula and verify that it integrates to 1. Then answer in one sentence why normalizing flows need *invertible* maps with *tractable* Jacobian determinants.

6.2 **(Level 4)** GAN training inverts an uncomfortable direction: the generator must map latent space → data space, the same direction in which real data have no supervised examples, while the discriminator learns density ratios for free. Explain (i) why mode collapse is a rational solution of the generator under the minimax objective, (ii) why GANs give you samples but not likelihoods, and (iii) why evaluation of a GAN is intrinsically indirect compared to flows or VAEs.

Worked Example — Solution 6.1 — change of variables

The general 1D formula is $p_X(x) = p_Z(f^{-1}(x)) \,|\det \partial f^{-1}/\partial x| = p_Z((x-\mu)/s)\,|1/s|$ (here det is just the scalar derivative). With $p_Z(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$:

$$p_X(x) = \frac{1}{\sqrt{2\pi}\, s} \exp\Big(-\tfrac{(x-\mu)^2}{2s^2}\Big) = \mathcal{N}(\mu, s^2),$$

the expected result: affine transports of a standard Gaussian are Gaussians with transformed mean and variance. It integrates to 1 because change of variables is exactly the substitution rule of integration. In general: the change-of-variables formula needs f^{-1} *evaluated at* x and $|\det J|$ of that inverse map — so a flow's layers must be invertible by construction (coupling with triangular Jacobians in Real NVP; 1x1 convolutions with LU determinants in GLOW), and the determinant must be cheap, because exact likelihood training evaluates it at every layer, at every data point, at every step.

Worked Example — Solution 6.2 — what the GAN's minimax pays for

(i) The generator wins the minimax game whenever the discriminator cannot distinguish its outputs from real data — it never promised to *cover* all data modes. If ten modes of real images are equally real, a generator that always produces one perfect mode still confuses an imperfect discriminator locally; coverage of other modes gives no extra reward, only optimization risk. Mode collapse is a degenerate-but-rational equilibrium of the game's local incentives. (ii) The discriminator measures a density *ratio* — how much more probable real versus fake is at a point — never an absolute $p_X(x)$; the generator is an implicit sampler, not a density model. So likelihoods are simply undefined in GAN space. (iii) Without likelihoods, evaluation standards are indirect proxies: human/judged sample quality, feature-space statistics of generated versus real sets (recognizability via a pretrained classifier, diversity vs. recall diagnostics), or downstream task usefulness — each couples evaluation to another learned model, unlike flows (exact likelihood) or the VAE's ELBO bound. This is why the taxonomy matters: the more flexible the sampling machinery, the more evaluative structure one has to give up.

29.8 A.7 Chapter 7 — Denoising, diffusion, flow matching

Check Your Understanding

7.1 (Level 3) In rectified flow / flow matching, interpolation is $x_t = (1-t)x_0 + tz$ with x_0 a data point, $z \sim \mathcal{N}(0, I)$, and the target velocity is $v = z - x_0$, constant along the interpolation path. A trained perfect model $\hat{v}$ satisfies $\hat{v}(x_t, t) = z - x_0$ at every t. Trace Euler sampling $x_{t+\Delta t} = x_t + \Delta t\, \hat{v}(x_t, t)$ backward from $t = 1$ to $t = 0$ in

one step ($\Delta t = -1$), and show you land exactly on the data endpoint. Why does straightness of the paths matter for the number of sampling steps?

7.2 (Level 4) Classifier-free guidance forms the sampling update from an unconditional model $u(x)$ and a conditional model $c(x)$ as $\hat{c}(x) = u(x) + w\,(c(x) - u(x))$ with guidance weight w. (Chapter 7 writes the same rule as $(1 + w')\,c - w'\,u$, i.e. $w = 1 + w'$; there $w' = 0$ is plain conditioning.) (a) Show that $w = 1$ recovers pure conditioning and $w = 0$ ignores the condition entirely. (b) Why does CFG double the sampling cost? (c) What qualitative failure appears at very large w, and explain it in terms of what the linear extrapolation $u + w(c - u)$ does to the "contrast" between conditioned and unconditioned predictions.

Worked Example — Solution 7.1 — flow matching arithmetic

Backward Euler in *decreasing* t: from the endpoint $x_1 = z = (0 + 1)z$ of the interpolation (with x_0 the data point), take one step with $\Delta t = -1$:

$$x_0' = x_1 + (-1)\,\hat{v}(x_1, 1) = z - (z - x_0) = x_0.$$

One step, landing *exactly* on the data point: this works because in rectified flow the true velocity field along each linear path is the constant $z - x_0$ and x_t moves on the segment $[x_0, z]$, so one Euler step is exact — no discretization error at all. For general (learned, real-data) couplings paths are only approximately straight; curvier (longer, conditional-average) fields need smaller steps to stay on manifold. The rectified objective *straightens* the learned vector field (the lecture's "why rectified"), which is precisely what buys few-step sampling: 5 lines of PyTorch, a handful of Euler steps.

Worked Example — Solution 7.2 — guidance mechanics

(a) $w = 1$: $\hat{c} = u + 1 \cdot (c - u) = c$, the conditional prediction itself. $w = 0$: $\hat{c} = u$, unconditional sampling. (b) Both model evaluations are needed at every denoising step, so the forward cost of sampling doubles — the "2x cost caveat" on the guidance slide. (c) For $w \gg 1$, $\hat{c} - u = w(c - u)$ extrapolates *beyond* the conditional prediction, hard amplifying the direction in which c differs from u. Score-space interpretation: the pushed field moves samples toward the modes where the condition is *maximally distinctive*, sharpening contrast with the unconditional distribution. Small $w > 1$ yields crisper, more on-prompt samples; very large w produces over-saturated, low-diversity artifacts — everyone mode-collapses onto the peak of the contrast, a trade between fidelity to the prompt and sample diversity. It is a knob without a free right setting; course-style answers name both the '$w = 1$ recovers condition' sanity check, the doubled cost, and the saturation/diversity failure as one coherent picture.

29.9 A.8 Chapter 8 — Pretext tasks and augmentations

Check Your Understanding

8.1 (Level 2) RotNet trains a classifier to recognize which of four rotations $\{0°, 90°, 180°, 270°\}$ was applied to an image. A dataset of N images becomes how many labeled training samples for the pretext task, and what is chance-level accuracy? Why do these two numbers together make the pretext signal "free" yet non-trivial?

8.2 (Level 2) Distinguish an *invariance* from a *covariance* of a representation f under a transformation family $\mathcal{T}$, give one example transformation that a classifier should treat each way, and explain why augmentation-based SSL methods must choose this split *per augmentation* rather than globally.

8.3 (Level 3) A jigsaw pretext task defines $P = 100$ permutations of 9 tiles and a classifier must predict which one was applied. Compute the chance-level accuracy and the entropy of the pretext label distribution (in bits, assuming uniform use of all permutations). Contrast with a binary same/different-location pretext task on 2 patches, and state the lesson this course draws from the comparison.

8.4 (Level 4) You train a rotation-invariant contrastive model and evaluate it on a benchmark whose classes include "6" vs. "9" and "70°-tilted sign" vs. "20°-tilted sign." Walk through the failure: what did the invariance do, why did the training loss not punish it, and where in the augmentation-selection pipeline should the choice have been corrected? (This is the transfer-failure side of the invariance-vs-covariance discussion.)

Worked Example — Solution 8.1 — RotNet's free supervision

N images become $4N$ labeled pretext samples — one per rotation, with the rotation index as label. Chance is $1/4 = 25\%$. The signal is free because the labels come from the augmentation itself; it is non-trivial because telling a 0° from a 270° image robustly requires knowing which way is "up" in the visual world (lighting from above, gravity in scenes, object orientation statistics), so a $> 25\%$ solution must already encode real structure. The thesis from Lecture 02 applies verbatim: better than chance is exploitable, and $4N$ of anything free beats N of anything expensive as a training signal.

Worked Example — Solution 8.2 — invariance vs. covariance

f is *invariant* to $t \in \mathcal{T}$ when $f(t(\boldsymbol{x})) = f(\boldsymbol{x})$ — the transformation is erased. f is *covariant* (equivariant, in the loose usage of the lecture) to t when $f(t(\boldsymbol{x})) = t'(f(\boldsymbol{x}))$ for a corresponding transform t' of the feature — the transformation is *preserved*, in structured form. A classifier should be invariant to horizontal flips for mammals but covariant to pixel translation for detection: if the object moves, its located features must move. The split cannot be global because the downstream task defines what counts

as nuisance: a contrastive augmentation strong enough to erase rotation is fatal for orientation tasks (Solution 8.4 makes this concrete). Hence augmentation selection is modeling: you declare, per transformation, whether it is nuisance (make the loss demand invariance) or signal (let it survive). Pretext-based methods encode the same choice implicitly: predicting the augmentation back (RotNet) forces covariance to rotation and invariance to never-occurring label changes.

Worked Example — Solution 8.3 — how much signal does a pretext task carry?

Chance accuracy is $1/100 = 1\%$. Label entropy under uniformity is $\log_2 100 \approx 6.64$ bits per sample. The binary same place/different place task carries $\log_2 2 = 1$ bit and 50% chance. Two lessons, not one: (i) a larger, well-designed pretext label space injects far more supervision energy per unlabeled example (and correspondingly finer pressure on features) — this is a coarse but honest accounting of "free" signal; (ii) yet signal quantity is not signal *quality*: any task is merely a chassis for the inductive bias you actually want, and the jigsaw's extra bits help only because recognizing tile permutations forces features to encode spatial layout, which transfers. State both halves; one-sided answers lose credit in this course's exams.

Worked Example — Solution 8.4 — when invariance attacks your task

Rotation augmentation entered training as a declared nuisance: the InfoNCE (or related) loss asks the representations of x and $\mathrm{rot}_\theta(x)$ to coincide, so angle information is actively unlearned. On evaluation, the benchmark demands the opposite split: a "6" and a "9" differ by a 180° rotation, and the two signs by a 50° tilt, so the angle is the *label*. The training loss cannot punish the over-invariance — its view pairs are optimization targets, not held-out evidence of downstream need; nothing in the contrastive objective senses "you erased something the user needed." The failure is therefore a mismatch between the designer's implicit task model (orientation is nuisance) and the benchmark's (orientation is signal). Correction point: at augmentation selection, before training — decide per-transformation invariance vs. covariance from the *intended* downstream class split (6 vs. 9 forces covariance to rotation), or use weaker, safe ranges. If noticed late, partial remedies are task-aware evaluation of candidate augmentations on a small labeled set, and preserving orientation-carrying branches/heads. The meta-lesson of the chapter: augmentations are not free regularizers; they are *assumptions* about what variation is nuisance.

29.10 A.9 Chapter 9 — Contrastive learning

Check Your Understanding

9.1 (Level 1) Order the contrastive loss ladder by how many negatives the loss couples to one anchor: Triplet, N-pair, InfoNCE. State in one sentence for each what is being pulled together and what is pushed apart, and why the move from pairwise to softmax-over-negatives matters.

9.2 (Level 3) A batch contains one positive pair and $K = 3$ negatives. The similarity scores before the softmax are $s_p = 2.0$ for the positive and $s_n = 1.0$ for each negative; temperature $\tau = 0.5$. (a) Compute the InfoNCE loss $\mathcal{L}_{\text{NCE}} = -\log \frac{e^{s_p/\tau}}{e^{s_p/\tau} + \sum_n e^{s_n/\tau}}$ numerically. (b) Repeat with $\tau = 2.0$ and with $\tau = 0.05$, keeping the logits fixed. (c) Conclude what the temperature controls; what goes wrong at both extremes?

9.3 (Level 4) In SimCLR the $2N - 2$ "negatives" of an unordered pair are just the other images in the batch. (a) Define a *false negative* in this setting. (b) Argue why their frequency shrinks with the number of distinct classes but grows with batch size, and (c) give two mitigations, one statistical within the batch and one structural from MoCo's design.

Worked Example — Solution 9.1 — the loss ladder

Triplet (one negative): pull anchor close to its positive, push it from *one* chosen negative past a margin — the objective is local and margin-based; which negative you pick dominates training, hence hard-negative mining. *N-pair* (many negatives): the positive is contrasted against *a whole set* simultaneously via a softmax, converting the margin problem into a smooth, always-nontrivial discrimination. *InfoNCE* (many negatives, temperature): the same softmax over similarities, now read as a mutual-information lower bound between the two views — it links the practical loss to the question "how much shared structure must the embedding carry." The conceptual jump from triplet to softmax-over-K is what turns "find one hard negative" engineering into "learn a geometry where shared content dominates" learning: every negative participates in every gradient, and the temperature-plus-batch-size pair governs the effective hardness of the task.

Worked Example — Solution 9.2 — InfoNCE numerics

Logits over τ: (a) $\tau = 0.5$: $s_p/\tau = 4$, each $s_n/\tau = 2$, so

$$\mathcal{L}_{\text{NCE}} = -\log \frac{e^4}{e^4 + 3e^2} = \log(1 + 3e^{-2}) \approx \log(1.4060) \approx 0.3406.$$

(b) $\tau = 2.0$: $s_p/\tau = 1$, $s_n/\tau = 0.5$, so $\mathcal{L} = \log(1 + 3e^{-0.5}) \approx \log(2.8196) \approx 1.0368$. $\tau = 0.05$: $s_p/\tau = 40$, $s_n/\tau = 20$, so $\mathcal{L} = \log(1 + 3e^{-20}) \approx 6.2 \times 10^{-9} \approx 0$. (c) τ scales the sharpness of the softmax discrimination. At $\tau \to 0$ the winning logit dominates —

loss and gradient vanish when the positive already leads (under-training; and with hard negatives, effectively hard-margin behavior with vanishing signal for everything else), at $\tau \to \infty$ all similarities look equal and the loss approaches the chance level $\log(K+1)$ with flat, nearly task-less gradients. Standard operating points sit between (SimCLR used $\tau = 0.1$-ish; DINO's τ pair makes the *teacher* sharper, not just flatter). The knob is doing real work: it decides how hard the example is, not just the scale of the loss.

Worked Example — Solution 9.3 — false negatives

(a) A false negative is another batch sample that shares the anchor's class (or strong semantics) but is used as a repelling term in the InfoNCE denominator — the loss *pushes apart* genuinely similar content. (b) If the batch samples $2N$ images over C classes uniformly, the expected number of adversaries of the anchor's class among the remaining $2N-2$ grows like $(2N-2)/C$: linear in batch size, inverse in class count. More (random) contrast always helps the signal-to-noise of the *average* negative; for a multi-class world it simultaneously adds contradictory pressure. (c) Mitigations — *within the batch (statistical):* a larger batch does *not* lower the fraction of false negatives (it stays about $1/C$), but it dilutes each one among many more true negatives, so any single false negative carries less of the softmax mass; more cleanly, exclude or reweight suspected same-class samples using current features (rejection by nearest-neighbor consistency). *Structural (MoCo):* decouple the negative pool from the batch — MoCo's queue holds tens of thousands of feature keys accumulated across many batches by an EMA encoder, so the effective negative set is much larger and statistically diverse without the batch having to be; the cost is staleness, controlled by the slow EMA drift. Also legitimate as a third line of defense: methods of Chapters 10–11 that drop negatives entirely (SimSiam, BYOL, DINO) are, in one sense, the industrial answer to this very failure mode.

29.11 A.10 Chapter 10 — Beyond negatives

Check Your Understanding

10.1 (Level 2) Write the symmetric SimSiam loss $\mathcal{L} = \frac{1}{2}D(\boldsymbol{p}_1, \boldsymbol{z}_2) + \frac{1}{2}D(\boldsymbol{p}_1 \leftrightarrow \boldsymbol{p}_2)$ (i.e. $D(\boldsymbol{p}_2, \boldsymbol{z}_1)$) explicitly in terms of the encoder f and predictor h, stating precisely where the stop-gradient applies. Then answer: which input sees gradient in each term, and why is the symmetrization more than a cosmetic trick?

10.2 (Level 4) BYOL/MoCo-style teachers follow the student by EMA: $\boldsymbol{\theta}_t \leftarrow m\,\boldsymbol{\theta}_t + (1-m)\,\boldsymbol{\theta}_s$. Suppose one SGD step relocates the student from $\boldsymbol{\theta}_s^{(k)}$ to $\boldsymbol{\theta}_s^{(k+1)} = \boldsymbol{\theta}_s^{(k)} + \boldsymbol{\delta}$, while the rest of history's drifts decay by factor m per step. Show that the teacher's displacement after the same step is $m\,\Delta\boldsymbol{\theta}_t^{(k)} + (1-m)\boldsymbol{\delta}$, unravel for the first three steps, and conclude why the EMA teacher both (i) reacts to but (ii) smooths the student's motion — the property that blocks instantaneous collapse.

Worked Example — Solution 10.1 — SimSiam, exactly

Let $\boldsymbol{z}_i = f(\boldsymbol{x}_i)$ (backbone + projection MLP) and $\boldsymbol{p}_i = h(\boldsymbol{z}_i)$ (prediction MLP); $D(\boldsymbol{p}, \boldsymbol{z}) = -\frac{\boldsymbol{p}}{\|\boldsymbol{p}\|_2} \cdot \frac{\boldsymbol{z}}{\|\boldsymbol{z}\|_2}$, and in implementation the *second* argument is detached, $\boldsymbol{z} \leftarrow \boldsymbol{z}$.detach(). Then

$$\mathcal{L} = \tfrac{1}{2}\, D(h(f(\boldsymbol{x}_1)),\ \mathrm{sg}(f(\boldsymbol{x}_2))) + \tfrac{1}{2}\, D(h(f(\boldsymbol{x}_2)),\ \mathrm{sg}(f(\boldsymbol{x}_1))).$$

In term 1, gradients flow through $\boldsymbol{x}_1$'s branch (through f *and* h); $\boldsymbol{x}_2$'s branch contributes the fixed target. Term 2 mirrors it. So per input pair, *both* branches carry gradient — *alternately* — while the symmetric pairing means branch roles (student/teacher-by-detachment) are exchanged, not duplicated. The symmetrization is essential, not cosmetic: it makes the view ordering irrelevant (augmentation pairs have no privileged side) and doubles the per-pair supervision without extra forward passes through two separate weight sets. (The lecture's empirics: without stop-gradient entirely, kNN accuracy stays near chance and the loss hits its floor immediately — instantaneous collapse; with it, accuracy climbs smoothly to $\sim$50% over 100 epochs. The exact mechanism is debated; what is not debated is the empirical picture.)

Worked Example — Solution 10.2 — EMA as a low-pass filter on parameters

One step rewrites the teacher displacement directly from the update rule:

$$\Delta\boldsymbol{\theta}_t^{(k+1)} = \boldsymbol{\theta}_t^{(k+1)} - \boldsymbol{\theta}_t^{(k)} = m\boldsymbol{\theta}_t^{(k)} + (1-m)\boldsymbol{\theta}_s^{(k+1)} - \boldsymbol{\theta}_t^{(k)} = m\,\Delta\boldsymbol{\theta}_t^{(k)} + (1-m)\,\boldsymbol{\delta},$$

where the last step subtracts the update at step k, $\boldsymbol{\theta}_t^{(k)} = m\boldsymbol{\theta}_t^{(k-1)} + (1-m)\boldsymbol{\theta}_s^{(k)}$, from the update at step $k+1$: the teacher terms give $m\,\Delta\boldsymbol{\theta}_t^{(k)}$ and the student terms give $(1-m)(\boldsymbol{\theta}_s^{(k+1)} - \boldsymbol{\theta}_s^{(k)}) = (1-m)\boldsymbol{\delta}$. Start with teacher and student equal, so $\Delta\boldsymbol{\theta}_t^{(0)} = 0$. Unravel three steps with student moves $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \boldsymbol{\delta}_3$ from the teacher at rest:

$$\begin{aligned}
\Delta_t^{(1)} &= (1-m)\boldsymbol{\delta}_1, \\
\Delta_t^{(2)} &= m(1-m)\boldsymbol{\delta}_1 + (1-m)\boldsymbol{\delta}_2, \\
\Delta_t^{(3)} &= m^2(1-m)\boldsymbol{\delta}_1 + m(1-m)\boldsymbol{\delta}_2 + (1-m)\boldsymbol{\delta}_3.
\end{aligned}$$

Every past student step enters the teacher's state, geometrically downweighted: the teacher's displacement is a trailing average of the student's motion. Two consequences. (i) *It does follow*: with factor $1-m$ on the newest move, m close to 1 keeps drift nonzero — the teacher still learns, laggardly. (ii) *No instantaneous agreement*: because the teacher cannot copy the student within one step, the loss-driving move "predict a constant" cannot simultaneously please the (detached, lagging) targets — the chasing-game dynamics that collapse would need are broken. Formally the EMA is a discrete-time low-pass filter on parameter noise; the $m \to 1$ limit freezes the teacher (stable targets, no learning), the

$m \to 0$ limit chases noise and re-opens collapse. Practical schedules warm m up toward 1 over training for exactly this reason.

29.12 A.11 Chapter 11 — Self-distillation and the DINO family

Check Your Understanding

11.1 (Level 2) DINO's teacher output is sharpened and centered: $p_t = \text{softmax}((\boldsymbol{g}_{\theta_t}(\boldsymbol{x}) - \boldsymbol{c})/\tau_t)$ with $\boldsymbol{c} \leftarrow m\,\boldsymbol{c} + (1-m)\,\frac{1}{B}\sum_i \boldsymbol{g}_{\theta_t}(\boldsymbol{x}_i)$. Describe the two complementary collapse modes of "student mimics teacher over K prototypes" and which of {centering, sharpening} blocks which.

11.2 (Level 3) KoLeo regularizes a batch of $n = 4$ unit-normalized feature vectors with nearest-neighbor distances within the batch $d_{n,1\ldots4} = \{0.5,\ e^{-1},\ e^{-1},\ 0.25\}$. Compute $\mathcal{L}_{\text{koleo}} = -\frac{1}{n}\sum_i \log d_{n,i}$ exactly (simplify), and qualitatively: what geometry does KoLeo encourage, and why would one want that on top of the DINO*i*BOT-type losses?

11.3 (Level 4) DINOv2's data engine maps 1.2 B crawled images to LVD-142M by: embedding with a self-supervised model, deduplication (hash-based exact copies, then near-duplicate removal), retrieval of new samples against curated "seed" sets, and NSFW filtering. (a) The pipeline uses clustering and kNN-style retrieval of Lecture 02/Chapter 3 content — name exactly where each one appears. (b) What is qualitatively wrong with skipping curation and training on raw crawl at the same final size? (Give two distinct, concrete mechanisms.)

Worked Example — Solution 11.1 — centering and sharpening vs. collapse

With K prototypes and cross-entropy between teacher and student distributions, "everyone agrees" can be reached two ways. *Mode 1 (one-hot domination):* a single prototype takes all mass for every input — cross-entropy of a constant peaked distribution with itself is trivially minimized; the code ignores the input entirely. *Mode 2 (uniform spread):* outputs go flat and uniform, again input-independent, with the loss near its entropy floor. *Sharpening* (τ_t small in the teacher) actively prevents uniform distributions: the teacher's softmax is pointy, so matching it pushes the student to be peaked too. *Centering* then prevents the pointiness from consolidating onto one prototype: the batch-mean is subtracted in logit space, so any component the teacher overuses across the batch is suppressed. The two operations are the matched pair the collapse taxonomy needs — neither alone suffices: sharpening alone invites mode 1, centering alone invites mode 2. (The EMA teacher and stop-gradient complete the mechanism: targets move, but slowly and without participation in the gradient; see Solution 10.2 for why that lag matters.)

Worked Example — Solution 11.2 — KoLeo arithmetic and meaning

With the given nearest-neighbor distances:

$$\begin{aligned}\mathcal{L}_{\text{koleo}} &= -\tfrac{1}{4}\big(\log 0.5 + \log e^{-1} + \log e^{-1} + \log 0.25\big)\\ &= -\tfrac{1}{4}(-0.6931 - 1 - 1 - 1.3863) = \tfrac{4.0794}{4} \approx 1.0199.\end{aligned}$$

Exactly: $-\frac{1}{4}(-\log 2 - 2 - 2\log 2) = \frac{1}{4}(2 + 3\log 2) = 0.5 + 0.75\log 2$. Minimizing it pushes each $d_{n,i}$ *up* — neighbors spread apart — so the geometry is a uniform covering of feature space: no tight duplicate clumps, no dead empty zones. The DINO and iBOT losses align different views of the same content but say nothing about how *different* samples sit relative to each other; KoLeo fills that gap with a batch-local spread prior, improving the usefulness of frozen features as a metric space (neighbors correspond to semantically graded similarity, retrieval works, and the Kozachenko–Leonenko view reads this as maximizing a sample-space entropy estimate).

Worked Example — Solution 11.3 — the data engine as an SSL pipeline

(a) Old material inside the new pipeline: *deduplication* uses hashing (exact) and kNN-style nearest-neighbor retrieval in embedding space for near-copies — Lecture 02's approximate nearest-neighbor machinery, run at billion scale; *retrieval against curated seeds* finds the unlabeled images most similar (in feature space) to curated concepts, again kNN; DINOv3's Part A additionally runs *hierarchical k-means* over embeddings to balance the pool (Vo et al.), closing the loop: Lecture 02's k-means becomes a data-selection algorithm. (b) Skipping curation on raw crawl fails for two distinct reasons, not one: *(i) distribution* — raw crawls are massively long-tailed and redundant (near-duplicate-heavy), so compute is spent re-seeing popular content rather than covering rare concepts; the model's effective capacity is spent on an accidental distribution; *(ii) content* — crawls mix in unfilterable-at-scale noise (near-identical product shots, image noise, unmodelable content, policy-violating material), which degrades both optimization (useless gradients on memorization-prone duplicates) and downstream alignment. Deduplication alone (mechanism ii, midpoint) still leaves the long tail (mechanism i) — retrieval and hierarchical sampling exist precisely to rebalance the mixture, not only to clean it. Exam credit requires naming both halves.

29.13 A.12 Chapter 12 — Self-training, distillation, cycle consistency

Check Your Understanding

12.1 (Level 2) Pseudo-labeling (Lee, 2013) turns a small labeled set plus a large unlabeled set into a bigger training set. State (i) the loop, (ii) the choice that separates confident from unconfident unlabeled samples, and (iii) the failure mode when early teacher errors are systematic. How does the Noisy Student variant alter the loop and why does its student get to surpass the teacher?

12.2 (Level 3) Hinton-style distillation uses soft targets with temperature T: probabilities $\text{softmax}(z/T)$. Consider teacher logits $\boldsymbol{z} = (4, 2, 0.2, 0)$. (a) Compute the teacher's class-1 probability at $T = 1$ and at $T = 4$; repeat for class 2. (b) Kullback–Leibler in the distillation loss is computed *between the temperature-stretched distributions* and then scaled by T^2; explain what the dark knowledge is in this example (hint: compare ratios p_2/p_1 at the two temperatures) and what T smooths away.

12.3 (Level 4) Lectures group methods by *paired vs. unpaired* supervision. For each of Pix2Pix, CycleGAN, augmentation-based contrastive learning, and cross-modal teacher–student distillation on a sensor rig, classify the pairing and state whether the pair acquisition is "for free." Then draw the line the course actually draws between self-supervised and semi-supervised in this taxonomy.

Worked Example — Solution 12.1 — pseudo-labels and noise as curriculum

(i) Train a teacher on the labeled set; predict on the unlabeled set; add the teacher's predictions as new targets; retrain (possibly the same or a fresh model) on labeled + pseudo-labeled data; iterate or scale. (ii) Confidence thresholding: only predictions whose maximum probability exceeds a cutoff enter the pseudo-labeled set — the rest are discarded this round. (iii) If early errors are systematic (a confidently mislabeled subclass), pseudo-labels amplify them: the student trains *to* the mistake, becomes more confident in it, and future rounds escalate — confirmation bias in its purest form. Noisy Student alters the loop in two ways: the student is *at least as large* as the teacher (usually bigger; the noise matters more than the size gap, since same-size students also improve) and is trained on the *noise-ridden* distribution (augmented, noised input), with pseudo-labels kept *soft* or unfiltered enough to preserve information but inputs made harder; noise stops memorization of idiosyncratic teacher errors and effectively distills the stable part of the teacher's function, so on the cleaner test distribution the noise-hardened student surpasses the teacher. Failure-mode name for credit: label noise compounds unless something (noise, capacity gap, soft targets) prevents the student from exactly fitting the teacher's mistakes.

Worked Example — Solution 12.2 — temperature, numbers first

At $T = 1$: $p = \text{softmax}(4, 2, 0.2, 0)$. Exponentiating: $e^4 \approx 54.598$, $e^2 \approx 7.389$, $e^{0.2} \approx 1.221$, $e^0 = 1$; sum ≈ 64.208. $p_1 \approx 0.8502$, $p_2 \approx 0.1151$, $p_3 \approx 0.0190$, $p_4 \approx 0.0156$. At $T = 4$: logits $(1, 0.5, 0.05, 0)$, $p \propto (e, e^{0.5}, e^{0.05}, 1) = (2.7183, 1.6487, 1.0513, 1)$, sum 6.4183, so $p_1 \approx 0.4235$, $p_2 \approx 0.2569$, $p_3 \approx 0.1638$, $p_4 \approx 0.1558$. (b) The dark knowledge is the *relative mass* of the wrong answers: at $T = 1$, second-best is disfavored $p_2/p_1 \approx 0.135$, and classes 3 and 4 barely register (under 2% each). At $T = 4$, $p_2/p_1 \approx 0.607$ and p_3, p_4 are visible for the first time — the teacher's full similarity structure (class 2 is "almost as right; 3 and 4 are wrong in different ways") is exposed. As $T \to \infty$ all structure within the distribution is washed out toward uniform; $T = 0$ recovers argmax hard labels with zero dark knowledge. So T is a knob revealing how much *error structure* transfers, and the T^2 rescaling compensates for the scale-shrinking of softmax gradients at high temperature.

Worked Example — Solution 12.3 — where the pairs come from

Pix2Pix: pairs are *collected* (aligned input-output images); strongly paired, expensive — strictly, supervised pairs, not "for free". *CycleGAN*: unpaired domains (two unaligned sets); the cycle loss manufactures pairing at the level of distributions without any matched sample. *Contrastive-by-augmentation*: pairs are exactly two augmented views of one image — paired by construction, free by definition. *Cross-modal distillation on a rig*: paired by the *system's clock* (RGB/depth/audio recorded simultaneously) — free in robotics contexts because the pairing is a property of the sensor platform, not of any labeling effort. The course's line: pairing itself does not disqualify a method from self-supervision — what matters is *who made the pair*. Pairs acquired by an augmentation, a sensor timestamp, or a rendering pipeline with zero human intervention count as self-supervised; pairs that needed any human matching or labeling labor move the method to the semi-supervised (or fully supervised) side. "Strictly semi-supervised" (Lecture 08's phrase) is the label for the latter case, and Pix2Pix is the canonical occupant.

29.14 A.13 Chapter 13 — World models

Check Your Understanding

13.1 (Level 3) A frozen DINOv2 feature extractor maps a 224×224 frame to a d-dim embedding; DINO-WM plans by treating embedding distance to a goal observation as cost and searching over action sequences through a learned dynamics model $\phi(\boldsymbol{z}_t, a_t) \approx \boldsymbol{z}_{t+1}$ in that frozen space. (a) Why is planning in a *frozen* self-supervised space attractive relative to a raw-pixel rollout? Name two distinct, concrete savings. (b) Explain what visual information DINOv2's invariances throw away that a planner operating on *pose* or *contact* would need — and what mitigation the DINO-WM design implies.

13.2 (Level 4) An action-conditioned video predictor is trained on robot logs: predict future frames $\boldsymbol{x}_{t+1:t+H}$ from current frame $\boldsymbol{x}_t$ and planned actions $a_{t:t+H}$. Lecture 08 hosted a debate: is this self-supervised? Take both sides in two short paragraphs — the "labels are free" argument and the "this is just conditional modeling of observed signals" counter — and then resolve it in one sentence using the provenance criterion of Chapter 14.

Worked Example — Solution 13.1 — why plan in frozen features

(a) Two distinct savings. *Compute:* a pixel-level rollout runs the full decoder per imagined step and compounds rendering error; embedding-space rollouts run one lightweight dynamics network per step with no decoder and no rendering (the patch-token state is not much smaller than the image — $256 \times 384 \approx 10^5$ numbers — but no pixels are ever produced), and the encoder is run once per real frame, not per imagined one. *Alignment:* pixel distances measure brightness and texture; frozen SSL-feature distances were *trained* to track semantic content — so the cost function $d(\boldsymbol{z}_t, \boldsymbol{z}_{\text{goal}})$ evaluates *task-relevant* closeness ("is the mug on the shelf") rather than photometric closeness, and does so without ever relabeling or finetuning. (b) What invariance removes: precisely the nuisances — mild lighting shifts, small jitter, moderate viewpoint change; but the same machinery partially erases exact pose, contact geometry, and free-space occupancy, which a manipulation/navigation planner requires. DINO-WM's implied mitigation: keep the backbone frozen for *semantics*, and place pose/contacts in the *learned dynamics model* that consumes actions; where absolute spatial fidelity matters, retain spatially structured patch tokens (not only the global embedding) so the dynamical map still has pixel-anchored spatial layout to bind to.

Worked Example — Solution 13.2 — both sides, then a criterion

Pro-SSL side: All targets are slices of the recorded stream itself; no human labeled anything. The future frames play exactly the role of the masked word or the rotation label: a target synthesized from data by a fixed procedure. On the "learning signal for free" thesis, action-conditioned prediction is a pretext task with the world's physics as its label generator — maximally self-supervised. *Skeptical side:* The predictor is doing conditional density estimation over observables; "actions" and "frames" are jointly *recorded* variables, and predicting some recorded variables from others is plain modeling, not self-supervision in the augmentation sense. Nothing about the objective forces transferable representation — an overfit conditional simulator solves it — so calling it SSL rests on the *hope* that prediction pressure yields abstraction. *Resolution:* By Chapter 14's provenance criterion it *is* system-level self-supervised: every target traces to the robot's own sensors and its own action log, with zero external labeling — regardless of whether one also chooses to view it as "just" conditional modeling, a distinction about the loss, not about the

supervision's source; the debate dissolves once one separates the *objective* language from the *provenance* language.

29.15 A.14 Chapter 14 — System-level SSL

Check Your Understanding

14.1 (Level 2) The lecture's five case studies are (1) map-rendered pseudo-labels with curiosity-driven planning, (2) depth/pose/flow cross-consistency (DF-Net), (3) traversability from proprioception plus DINO-feature novelty (WVN), (4) an audio student taught by RGB/thermal/depth teachers (MM-DistillNet), and (5) autonomous reinforcement learning from ensemble uncertainty and self-resets (ReLMM). For *each*, state in one line: what supervises what, and which of the closing taxonomy's three axes it primarily uses (paired modalities, physical laws/geometry, uncertainty).

14.2 (Level 4) In MM-DistillNet, an audio-only student is trained to match a multi-teacher (RGB, thermal, depth) representation via a focal loss plus an attention-based KL term. Argue (i) why a single-modality teacher *must* lose to a fused teacher under exactly the conditions the rig was designed for, (ii) what each loss term is for — one sentence each — and (iii) one realistic failure mode of student-below-teacher-even-at-optimum.

Worked Example — Solution 14.1 — inventory and axes

(1) *Map supervision*: a volumetric semantic map, fused from many frames with SLAM poses and re-rendered into the camera views, supervises the per-frame segmentation network — primarily *paired modalities* (camera plus pose estimate manufacture the frame–map correspondence for free), with curiosity (uncertainty-driven) planning closing data collection. (2) *DF-Net*: depth, ego-motion, and optical flow must be jointly geometrically consistent — a *physical laws* axis method (rigid-scene geometry supplies the targets that no labeler did). (3) *WVN*: what the legs felt while driving (proprioceptive slip/ vibration) supervises the camera's future traversability, with a DINO-feature autoencoder flagging visual novelty — *paired modalities* (embodiment as a pairing device) with an *uncertainty*/ novelty accessory. (4) *MM-DistillNet*: visual/thermal/depth teachers supervise an audio student recorded synchronously — *paired modalities*, engineered knowledge transfer across sensor types. (5) *ReLMM*: ensemble disagreement tells the agent when its own policy is unreliable and resets it into a supervised episode — *uncertainty* as the primary axis (with self-reset as the autonomy machinery around it). The taxonomy's point: most systems mix axes, but each has one load-bearing supervision source; naming it correctly is what the question tests.

Worked Example — Solution 14.2 — students, teachers, and their limits

(i) The rig exists for sensors with *partially* overlapping content: RGB misses scene content in the dark; thermal misses texture and see-through surfaces; depth misses material/appearance. A single-modality teacher's representation carries only what its sensor saw; the fused teacher's representation is (roughly) the union of what *any* sensor saw, so under the rig's diverse-conditions design the fused target is strictly richer on the conditions where each sensor is weak. Supervising with that richer target passes more information to the audio student. (ii) The *focal* term reweights distillation so that hard, low-agreement samples dominate the loss instead of being drowned out by trivial background, and the *attention-KL* term matches spatial *attention maps* between teacher and student, transferring "where to look" structure that pure feature matching might not transmit. (iii) A realistic asymmetry failure: the student's sensing modality cannot express the teacher's target for some scene aspects — no acoustic signature exists for a silent smooth glass wall that depth saw clearly — so the KL/feature loss on those dimensions is *unachievable* and acts as label noise; the student optimizes a compromise and underperforms even its own ceiling. Practical recognition: student–teacher agreement saturates while teacher quality keeps climbing; the remedy is to weight distillation by modality predictability, or to accept the student's own modality ceiling in the evaluation protocol.

29.16 A.15 Chapter 15 — Why does SSL work?

Check Your Understanding

15.1 (Level 2) Three explanations of SSL's success: *mutual-information bounds*, *augmentation-overlap* arguments, and *spectral/collapse* analyses of joint embeddings. Classify each as theorem, empirical observation, or hypothesis — with the caveat which parts of each are rigorous, and which are extrapolations a careful author must flag.

15.2 (Level 5) A colleague proposes: "SSL works because the InfoNCE loss is a lower bound on the mutual information between augmented views." Articulate two distinct technical objections — one about tightness/estimators, one about the direction of implication (does a bound being optimized actually maximize I?) — and sketch, in two lines, how an empirical research program could *separate* correlation from causation here.

Worked Example — Solution 15.1 — epistemic status, honestly

Mutual-information bounds ($I(\boldsymbol{x}_1;\boldsymbol{x}_2) \geq \log(K+1) - \mathcal{L}_{\text{InfoNCE}}$-type statements): exist as *theorems* about specific estimators under specific assumptions (a known data distribution and an exact softmax contrast); the *hypothesis* is that optimizing the bound drives a

practical network's representation to share the cause of MI with downstream utility — that inference step is not proven, and the book marks it as interpretation. *Augmentation-overlap:* the factual half — augmentation distributions of distinct images overlap after strong transforms, so contrastive objectives can connect semantically right or wrong pairs — is an *empirical observation*; theorem-strength statements exist only in stylized model (synthetic classes, Gaussian data, linear models). Real-data reach: hypothesis marked as such. *Spectral/collapse analyses:* of the three, the cleanest theorems — for linear models and Barlow-Twins/VICReg-type objectives, the eigenvalue dynamics and anti-collapse roles of variance terms *are* provable; the extrapolation to nonlinear networks trained by SGD at scale is a careful *extrapolation*, credible but not derived. The taxonomy matters in writing: honest chapters distinguish these registers sentence by sentence.

Worked Example — Solution 15.2 — against the MI-bound story

Objection 1 (tightness/estimation): the bound $I \geq \log(K+1) - \mathcal{L}_{\text{NCE}}$ with K negatives is loose unless K approaches the effective support size of the competing sample space — practical K (batch-scale) leaves a slack whose size is unknown and possibly dominant; optimizing a loose bound's upper end need barely move the true MI at all, so "we raise MI" is empirically under-evidenced relative to "we lower the loss." *Objection 2 (implication direction):* even granting the bound is tight, maximizing $I(\boldsymbol{x}_1; \boldsymbol{x}_2)$ between views says what the representation *keeps*, not whether what it keeps is what downstream needs: an MI-optimal code could specialize in shared augmentation artifacts (identical JPEG block grid, shared sensor noise) that are MI-rich and transfer-poor. *A two-line empirical separation:* (i) vary the estimator's K/tightness while holding architecture, data, and downstream protocol fixed — if transfer tracks a better proxy than estimated MI, the MI story fails its causal test; (ii) intervention: inject an MI-rich, semantics-free shared pattern into both views and observe whether transfer degrades even as measured MI rises — dissociation between the two refutes the sufficiency half of the claim. Both are doable studies; neither, to this book's knowledge, has yet been done at decisive scale, which is why the chapter files the MI story as interpretation rather than explanation.

29.17 A.16 Chapter 16 — Comparing the families

Check Your Understanding

16.1 (Level 2) Your constraints: (i) frozen-feature linear probes are your evaluation, (ii) modest batch size and no TPU pod, (iii) the downstream task is classification over natural images. Rank the three families — reconstruction (MAE), contrastive (SimCLR/MoCo), self-distillation (DINO-like) — and name the single decision-changing fact that would move reconstruction to first place.

16.2 (Level 4) Two papers report "77% ImageNet" for self-supervised backbones. Paper A used a linear probe on frozen features; paper B used kNN ($k = 20$) on frozen features. Can both numbers coexist with *identical* backbones? Compute-free but concrete: explain what each protocol is capable of expressing, give one realistic reason linear beats kNN and one realistic reason kNN can beat linear, and name the reporting detail without which the comparison is meaningless.

Worked Example — Solution 16.1 — choose with your constraints, not the leaderboard

Under those constraints the ranking is self-distillation (DINO-family) > contrastive (MoCo, then SimCLR) > reconstruction (MAE). Rationale, constraint by constraint: (i) frozen linear probing is what joint-embedding objectives *shape* (their loss demands linearly-separable semantic structure), and MAE's detail-rich features intentionally lag there (Solution 5.2); (ii) DINO/MoCo-family procedures are not batch-size hostage — MoCo's queue decouples negatives from batch size and DINO drops negatives entirely — while SimCLR-style training is sensitive to large batches; (iii) for *classification*, invariance-shaped features are the closest match to the label structure. The decision-changing fact moving reconstruction to first place: a shift in the *downstream* contract — if the task becomes dense prediction (segmentation, depth) or requires finetuning anyway (labels exist in moderate quantity), MAE-family reconstruction targets exactly that setting and jumps the queue; the comparison table's verdicts are all relative to the *stated* evaluation protocol.

Worked Example — Solution 16.2 — linear vs. kNN is a comparison of rulers

Yes: reported numbers across protocols are not commensurable, and identical features can legitimately produce both values. What each expresses: a linear probe learns the best *global affine classifier* on frozen features — it can succeed only where classes are (at least mostly) linearly separable, and it benefits from every direction the probe weight vector can exploit; kNN assesses *local metric structure* — it wins when same-class points cluster tightly in the raw geometry, regardless of whether a single hyperplane exists. *Why linear can beat kNN:* the probe can learn one dedicated direction (its weight vector) that combines many weakly informative axes — improving on the geometry's raw neighborship, which averages over all coordinates including nuisance-variance dimensions. *Why kNN can beat linear:* on curved manifolds (class structure bent around in the space) local neighborhoods are faithfully class-pure while no single hyperplane separates them; kNN reads local purity that a global linear classifier cannot see. The meaningless-comparison point: without stating protocol (probe/kNN, temperature or weighting for kNN, which layer, augmentations at eval), a number of "77%" is a datum about a pipeline, not a feature extractor — the course's repeated distinction (evaluation taxonomy of Lecture 07, slide 10) exists exactly to force this reporting discipline.

Part B: How to write a strong exam answer

The course's exam mixes knowledge questions, case studies, and implementation-on-paper (Lecture 07, slide 2), and the three example questions of slides 3–5 illustrate the intended register. Full-credit answers share a shape worth internalizing:

- **Define the terms you use**, even if the question assumes them. "Density-based" means nothing until you have said "clusters are dense regions connected through core points with at least min_samples neighbors within ε."
- **State the loss or algorithmic step** when the question touches a mechanism. In a coding question, the template is the skeleton and the credit lives in which symbols you fill in and where stop-gradients, detachments, or cluster re-computations sit.
- **Name your assumptions** explicitly: metric choices, known versus unknown k, memory limits, label budgets. Case-study answers are graded on this awareness.
- **Name one failure mode**, preferably the one your chosen method is famous for, with a sentence on mitigation. It demonstrates you know where the method's guarantee ends.

Below, each of the three slide questions is answered in full exam register. These are intentionally longer than a timed-exam answer needs to be, so that the structure is visible; in the exam, each paragraph below compresses to two or three sentences.

29.17.1 B.1 Exemplar answer — question 1 (the DBSCAN multiple choice)

Worked Example — Full-credit exemplar, question 1

Answer: (C). Statement by statement, as the exam expects them justified rather than circled:

(A) is correct. DBSCAN discovers the number of clusters from data: it grows clusters from any unvisited core point by depth-first expansion through connected ε-neighborhoods, and terminates whenever no core point is reachable — the count of clusters is an *output*, not an input. This is its primary contrast with k-means, where k must be supplied.

(B) is correct; it is the definition. Cluster membership is decided by local density — a point joins a cluster if it lies within distance ε of a *core point*, i.e. a point whose ε-ball contains at least min_samples other points. Shape restrictions of centroid methods (roughly convex, isotropic clusters) do not apply.

(C) is the wrong statement, and the intended answer. Naively DBSCAN needs the pairwise neighborhood structure, so memory grows as $O(n^2)$ in the dataset size and time is quadratic without an index; at millions of points, as in the course's own second question, one is immediately forced into the scaling toolbox: OPTICS (memory-friendlier, usually slower), dimensionality reduction (PCA in feature space), and subsampling or pre-clustering to exemplars. By contrast k-means minibatches cleanly and its memory is $O(k \cdot d)$, so relative to the alternatives DBSCAN is precisely the method that does *not* scale efficiently.

(D) is correct and operationally important: any point that is neither a core point nor density-reachable from one is labeled noise. This gives DBSCAN an explicit outlier mechanism that k-means lacks — k-means assigns the outlier to its nearest center and thereby skews it.
Failure mode to mention for top marks: DBSCAN is sensitive to ε; with multiple densities a single ε cannot serve every cluster, which is exactly the gap OPTICS/hierarchical variants address.

29.17.2 B.2 Exemplar answer — question 2 (the plant case study)

Worked Example — Full-credit exemplar, question 2

Restating the constraints (always do this first in a case study): 2,000,000 images; unknown number of classes k; 2000 budgeted labels; no domain expertise available. The budget is 0.1% of the data, so the method must be *label-free at its core*, with labels spent on *calibration and naming*, not on fitting a classifier.
Step 1 — representation learning on everything. Train a self-supervised feature extractor on all 2×10^6 images — a joint-embedding method such as a DINO-style student–teacher, or a contrastive method; what is essential is that the objective needs no labels: two augmented views of one image are required to map nearby. Joint-embedding training is deliberately chosen here over pure reconstruction because frozen features from such methods are linearly and geometrically well organized, which is what the next step exploits. *Assumption made explicit:* the taxonomy's invariance set should not erase what distinguishes species — strong blur is safe, aggressive color distortion is risky if species differ by coloration, and cropping scale must respect that discriminative plant detail can be small. (This is Chapter 8's rule that augmentations encode a task model.)
Step 2 — cluster in feature space, at scale. Embed all images, apply PCA for speed and decorrelation, then cluster. Because k is unknown, two honest options exist and both should be named: (a) a density method such as DBSCAN discovers k automatically — but naive DBSCAN on 2×10^6 points needs a neighborhood structure of size $\sim n^2$ (the trap from question 1), so use subsampled exemplars or OPTICS-style memory control; (b) the more standard route at this scale is minibatch k-means over a grid of k values, with the choice of k *validated* using the labeled subset (the k maximizing agreement between clusters and the labeled species, e.g. by cluster-purity or adjusted-rand-type scores). A practical hybrid: k-means at large k to get fine groups, then merge groups that the labeled members show to be one species.
Step 3 — spend the 2000 labels on naming. The colleague's labels are not training data for the feature extractor; they calibrate the structure. Assign each cluster the majority species among its labeled members; clusters containing no labeled member flag under-covered regions of the dataset. With 2000 labels, a held-out split (say 1500/500)

simultaneously evaluates the pipeline: kNN or linear classification on frozen features against held-out labels is the honest internal accuracy estimate.

Step 4 (optional amplifier) — deep clustering. Iterate: use cluster assignments as pseudo-labels and train the network end-to-end by cross-entropy, exactly the deep-clustering loop of question 3; re-extract features, re-cluster, repeat. This closes the virtuous circle: each round sharpens features so the next clustering is better aligned with species structure.

Failure mode to name: pseudo-label/ cluster collapse — if k-means produces imbalanced or trivial assignments one epoch, the cross-entropy phase can lock the network onto the artifact; counter with balanced re-sampling per cluster and by re-validating against the labeled set every round. A secondary named risk: 2000 labels spread over k species are thin if k is large — hence the validation-using-labels design instead of the train-classifier-on-labels design, which would overfit.

29.17.3 B.3 Exemplar answer — question 3 (deep clustering code completion)

Worked Example — Full-credit exemplar, question 3

Setup sentence (define terms): DeepCluster alternates two phases per epoch: a clustering phase that converts the *current* features of all images into pseudo-labels, and a classification phase that trains the network to reproduce those pseudo-labels by cross-entropy. The completed template:

```
for epoch in range(num_epochs):
    # Phase 1: clustering step (uses the encoder *before* this epoch)
    feats = encoder(all_images)          # extract features for all images
    feats = pca(feats, output_dim=256)   # reduce dimension, decorrelate
    centroids, pseudo = kmeans(feats, k) # pseudo[i]: cluster index of image i

    # Phase 2: network update with pseudo-labels as targets
    for x, y in minibatches(X, pseudo):
        # x: input minibatch, shape (batch_size, input_dim)
        logits = classifier_head(encoder(x))
        loss = cross_entropy(logits, y)  # CE against the cluster index
        sgd(loss).backwards()
```

Why each line is where it is (the reasoning the grader looks for): `encoder(all_images)` sits *before* the minibatch loop because targets must be generated by one consistent network — re-clustering after each gradient step would chase a moving definition of the pseudo-labels. The PCA line is part of the course's version of the algorithm (Lecture 02, slide 24), both for speed and because raw high-dim features degrade k-means. `kmeans` outputs per-image indices that become integer targets; `cross_entropy` is then ordinary supervised classification loss — there is nothing exotic left, which is the point of the method.

Stated assumptions: k supplied in advance (k-means needs it — contrast with DBSCAN in question 1); pseudo-labels regenerated once per epoch; the classifier head is separate from and retrained jointly with the encoder.
Failure mode and mitigation: degenerate pseudo-label distributions (cluster collapse) let the network reach zero loss by predicting constant logits without learning semantics; DeepCluster-family practice mitigates by balanced sampling from clusters during training and by the guaranteed refresh of targets each epoch. Mentioning the "better-than-chance" thesis closes the loop with the course's central message: this works because even weak initial features cluster non-randomly, and gradient descent amplifies the non-random part.

Part C: Capstone problems

The problems below are integrative: each requires combining material from several chapters, and each is stated exactly the way an ill-specified real-world project is. The solution outlines that follow are condensed but complete in structure — every design choice is motivated and every unavoidable failure mode named. Numbers that appear are worked as they arise; none are borrowed from benchmarks.

Capstone 1 — the underwater robot with only a depth camera

Check Your Understanding

Problem. You operate an underwater inspection robot equipped *only* with a depth camera (no RGB — turbidity makes color images useless) plus its standard proprioception (IMU, thruster commands, depth pressure sensor) and an accurate clock. The robot must learn visual representations for inspection tasks — finding damage on hulls, recognizing valve types — with zero labeled data and zero opportunity to send a diver. Design, end to end: (a) where the supervision comes from, (b) the architecture and losses, (c) the failure modes specific to this platform and how you would detect them, (d) an evaluation protocol with no labels.

Worked Example — Solution outline — Capstone 1

(a) Supervision sources, all on-board, hence all self-supervised by the provenance criterion of Chapter 14. *(i) Geometric consistency* (DF-Net's axis): consecutive frames of a depth camera plus IMU odometry obey rigid-scene geometry. Backproject depth to a point cloud, transform by the ego-motion estimate, reproject into the next frame: discrepancy between warped and measured depth is a free, dense target. *(ii) Self-augmentation invariance:* augmentation pairs exist for any modality — crops, small rotations, depth noise injection — so joint-embedding objectives (SimSiam/DINO-style, no negatives needed, which matters in part (c)) apply verbatim to depth as a one-channel image. *(iii) Proprioception pairing* (WVN's axis): vibration and thrust load measured during contact maneuvers label the depth footage retroactively ("this view precedes hard contact"), a free if weak signal for affordance-like structure. **(b) Architecture and losses.** A small ViT or ResNet encoder over single-channel range images; three loss heads: the geometric reprojection/consistency loss between consecutive frames; a SimSiam-style symmetric negative-cosine loss with predictor and stop-gradient on augmented views (no negatives — underwater sequences have heavy temporal correlation, so in-batch negatives would be mostly false); a lightweight masked-depth reconstruction head (MAE-style, high masking ratio) to force geometric rather than photometric detail into the features. Teacher by EMA, exactly as in Solution 10.2, so the geometric losses' occasionally wild gradients do not destabilize targets. **(c) Failure modes and detection.** *Mode collapse* is the first risk: stop-gradient plus the EMA lag are the structural defense; monitor the entropy of the head's prototype distribution over a window — a collapsing entropy is the alarm

before accuracy symptoms exist. *False negatives* in any contrastive variant are severe because consecutive frames are near duplicates — this is why the design avoids negatives altogether. *Geometry failures*: non-rigid scenes (fish, swaying vegetation) violate the rigidity assumption; detect per-pixel residual maps clustering on vegetation-like textures and mask those regions from the consistency loss. *Turbidity drift*: medium changes break photometric consistency — depth-with-confidence weighting keeps the geometric loss honest. **(d) Evaluation without labels.** (i) kNN retrieval sanity: nearest neighbors of any frame in feature space should be geometrically co-visible frames — a measurable, label-free consistency rate. (ii) Linear-probe protocol against *proprioceptive* weak labels (contact/no-contact): a training-free proxy for whether features encode manipulation-relevant structure. (iii) Emergent-segmentation inspection: DINO-style attention rollout maps on the depth patches should localize hull surfaces and objects; inspection by a human on 50 random frames is a legitimate, bounded audit. None of these replaces labeled evaluation; together they bound the risk of shipping degenerate features.

Capstone 2 — unlabeled histology, one week of compute

Check Your Understanding

Problem. A medical lab holds 400,000 whole-slide histology images (each gigapixel-scale; pre-tiled into $\sim 2 \times 10^7$ patches of 224×224 pixels), no diagnostic labels, and one week on a 4-GPU node. Pathologists will, later, label a few hundred slides. Design a plan whose output — in one week — is a frozen feature extractor plus a defensible internal evaluation, such that the later slide-level label effort is as small as possible. State your method, your budget arithmetic, the augmentation decisions a vision generalist would get wrong, and how you will convince the lab that the features are worth labeling against.

Worked Example — Solution outline — Capstone 2

Method. DINO-style self-distillation on patches, in the multiple-instance spirit histology demands: patches, not slides, are the SSL unit; slides enter only in evaluation. Self-distillation (no negatives) is chosen for two concrete reasons: histology batches are dominated by self-similar tissue (false negatives plague contrastive batches), and the frozen-feature kNN/linear-probe quality this family is known for is exactly the quality that minimizes the pathologists' labeling burden afterwards. iBOT-style masked-patch loss as a second head where GPU time allows it — it forces the spatial structure that tumor architecture evaluation will want. **Budget arithmetic (worked, self-consistent).** 2×10^7 patches, one week = 168 h. If sustained throughput is 3000 images/s across the node (the only unverified number in this plan, and the one to measure on day one), an epoch costs $2 \times 10^7 / 3000 \approx 6700$ s ≈ 1.9 h; the week then affords ~ 85 epochs with $\sim 10\%$ reserved for evaluation and failure. Sensitivity: at 1500 im/s you get ~ 42 epochs — still in the regime where DINO-training curves are climbing usefully, which

is why the week plan is a self-distillation plan and not a boundary- case large-batch contrastive plan. **Domain-critical augmentation decisions.** Two generalist defaults must be reversed: *color normalization* — H&E staining varies by scanner/protocol, so *stain augmentation* (or explicit stain-invariant augmentation) is load-bearing — naive heavy color jitter as in natural images would erase the diagnostic signal itself, precisely the invariance-vs-covariance failure of Solution 8.4; *orientation* — tissue architecture has no canonical "up," so full 90° rotation *invariance* is legitimate here (the opposite choice from the digit/sign benchmark of that same solution). Multi-crop with a hierarchy of scales matters more than in natural images: diagnosis often lives in patch–context relations. **Convincing the lab (evaluation without diagnosis labels).** (i) Frozen-feature retrieval audit: pathologists inspect kNN retrievals for 100 hand-searched query patches — precision-by-inspection on a budget of hours. (ii) Cluster forensics: k-means at several k over PCA-reduced features; report cluster–slide and cluster–stain-protocol breakdowns — a method whose clusters track the *scanner* rather than tissue morphology has failed silently, and this is the test that catches it. (iii) Probe with free metadata where it exists (e.g. tissue-type fields in slide headers without diagnostic content): a linear probe against those fields is a label-free-ish lower bound on later diagnostic utility. The contract with the pathologists: slides labeled later are spent exclusively on slide-level heads over frozen patch features (the linear-probe, low-label protocol of the evaluation taxonomy), which is the regime the whole design was optimized for.

Capstone 3 — learning from your own fleet's logs

Check Your Understanding

Problem. A delivery-drone fleet of 200 drones has flown 10^4 flight-hours total, logging synchronized RGB, IMU, GNSS, and motor telemetry — all timestamped, none labeled. Two crash classes keep recurring (branch strikes at dusk, GPS-denied drift near buildings). Design the self-supervised program that turns the logs into (a) a general visual encoder for downstream autonomy modules and (b) precursor detectors for the two crash classes, without a labeling budget beyond one engineer-month of log review. Name, for each supervision signal you use, which axis of the closing taxonomy it sits on, and at least one way each can silently lie to you.

Worked Example — Solution outline — Capstone 3

(a) The general encoder. Cross-modal distillation is the backbone of the design (MM-DistillNet pattern): IMU/GNSS/motor streams are cheap, robust "teachers" about ego-motion and dynamics; RGB learns to predict their latent structure, acquiring physics-aware features no offline-augmentation-only method gets — supervision sits on the *paired modalities* axis with pairs free by the fleet's clock. On top, temporal joint-embedding

(nearby frames as positive views, EMA teacher) adds scene content structure; the second source sits on the same axis via time. *Silently lies:* sensor agreement is not ground truth — systematic GNSS multipath near buildings mispairs "where the drone thought it was" with what the camera showed, and distillation faithfully transfers the error. Detection: residual distributions conditioned on building proximity (a telemetry-derived flag) expose exactly this strata. **(b) Precursor detectors.** Both crash classes admit free retrospective pseudo-labels: flight termination and hard-acceleration events in telemetry mark crash time t_c; RGB frames in the seconds before t_c are candidate precursors — the map-rendered-pseudo-label logic of the Frey-style case study with the log as the renderer (physical-laws axis: the label ground truth is dynamics, not annotation). Train sequence models to predict t_c-imminence from encoder features. The engineer-month is spent *auditing*, not labeling: review ∼200 auto-flagged precursor windows for correctness, which converts an unknown-precision detector into a measured-precision one — the same labels-spent-on-calibration pattern as the plant case study and Capstone 2. *Silently lies (i):* dusk-branch strikes correlate with exposure — the detector may learn "dusk," not "branch"; counter with negative mining on non-crashing dusk flights (abundant in the logs) and an explicit feature-level check that features local to vegetation carry the signal (attention/attention-KL audit, as in Solution 14.2). *Silently lies (ii):* for GPS-denied drift, the encoder may bind to building *appearance* rather than drift *dynamics*; counter by requiring the detector to use IMU-conditioned features, not frozen visual features alone — and by accepting (and stating) that the residual capability ceiling is the modality's, per the MM-DistillNet asymmetry lesson.

Capstone 4 — the museum archive of a million photos

Check Your Understanding

Problem. A museum wants its archive of 1,000,000 digitized object photographs (unlabeled; object types unknown; many near-duplicate shots of popular exhibits) turned into a searchable, browsable-by-similarity collection, with a small volunteer team available to spot-check results. Deliver: deduplication, concept discovery, and a similarity-search service — no classification labels exist or will be created. Lay out the pipeline as a composition of course methods, give the scaling decisions at each stage, and state where the volunteers' time buys the most quality per hour.

Worked Example — Solution outline — Capstone 4

The pipeline is deliberately a replay of the DINO-family data engine (DINOv2's path 1.2 B → 142 M, scaled down about three orders of magnitude — the design pattern transfers even when the budget does not). **Stage 1 — embed everything.** A frozen, pretrained self-supervised encoder (a DINOv2-style off-the-shelf backbone) maps all 10^6

photos to features. This is a data-engine insight, not laziness: an existing self-supervised feature space is good enough to dedupe and organize, so the museum never pays full SSL-training cost for its first working system. **Stage 2 — deduplicate.** Two-tier, per Lecture 02: exact duplicates via hashing; near-duplicates via approximate kNN in feature space. Scaling: 10^6 points is where naive $O(n^2)$ (all pairs form a $10^6 \times 10^6$ matrix: 10^{12} entries, about 4 TB in float32, infeasible in memory) definitively ends, but indexed ANN search over 10^6 vectors of a few hundred dimensions is a minutes-scale job on one machine — the whole reason LSH-style indexes or HNSW/Faiss libraries exist (recall-vs-throughput trade of Solution 2.2, run at a generous recall setting since dedup errors are cheap relative to the downstream audit below). **Stage 3 — concept discovery.** PCA to compress, then clustering at multiple granularities with minibatch k-means over a grid of k — *not* DBSCAN as the primary engine: the exam question's own trap applies, memory scaling of naive DBSCAN at 10^6 points needs exactly the exemplary/trick layer, and the density estimate's sensitivity to ε is poor match for a collection spanning photographs, documents, and artifacts. Hierarchy: coarse k for browse-by-theme, finer k within clusters for near-duplicate-adjacent browsing — the two-level structure of Vo et al.'s hierarchical k-means simplified to the museum's scale. **Stage 4 — the similarity service.** ANN index over the deduplicated representatives (not all 10^6 frames — the index serves *concepts*, then fans out to copies on demand). The volunteers, in descending value per hour: (i) audit of near-duplicate merge groups at the chosen similarity threshold — the single decision that most changes user trust; (ii) naming/coarse validation of the largest clusters (the calibration-not-training label spend of the plant case study); (iii) a final pass judging retrieval quality on 100 random queries, which produces the reportable precision number. Note what was deliberately not done: no classifier was trained, no taxonomy imposed, no label acquisition pipeline built — the task was organization, and organization is a frozen-feature job.

Capstone 5 — pretraining in the representation space you will plan in

Check Your Understanding

Problem. A startup wants a household-robot manipulation stack that (a) learns visual representations from unlabeled teleoperation-free camera logs, (b) learns a *world model* in the learned feature space conditioned on the robot's actions (the DINO-WM pattern), and (c) plans by feature-space distance to goal observations. Their question to you: which choices in stage (a) silently determine the quality ceiling of (b) and (c)? Give a decision list ordered by irreversibility — what can be fixed later by adding the dynamics model, and what is baked in at pretraining time — each entry with the reason and the failure it avoids.

Worked Example — Solution outline — Capstone 5

Ordered by irreversibility (things baked into the frozen encoder first):
1. What the invariances erase is unrecoverable downstream. Augmentation choices at pretraining decide which variation the planner's cost function is blind to. Erase rotation and the planner cannot reach an orientation-specified goal; erase color and knob—LED state becomes invisible, not just unseen. This is Solution 8.2/8.4 industrialized: build the augmentation set from the *manipulation task distribution*, not from the natural-image defaults, and prefer covariance-preserving choices (weak geometric jitter; avoid aggressive hue/rotation invariance). Fixable later? No — no dynamics model re-introduces information the encoder deleted. **2. Whether you keep spatial structure or only a global embedding.** A single global embedding (the [CLS]-only pipeline) is a clean-distance, poor-geometry representation: contact, occlusion, and free-space structure live in patch tokens. Keeping patch tokens (an iBOT-augmented recipe) is nearly free at pretraining and decisive for whether the world model has spatial layout to bind actions to. Fixable later? Only by retraining the encoder. **3. Freeze vs. finetune boundary.** The DINO-WM choice — freeze the encoder, learn dynamics on top — commits the system to feature stability (the planner's cost landscape does not drift as data accumulates) at the price of accepting the encoder's resolution limits. Fixable? Partially: a later finetune stage is possible, but then every learned dynamics model and every recorded goal embedding invalidates; decide at stage (a) and log the decision as an interface contract. **4. What the world model is allowed to predict.** Action-conditioned rollout error compounds in whichever directions the features are sharp; if the encoder carries photometric detail (reconstruction-heavy pretraining), the dynamics model wastes capacity predicting lighting drift and gets penalized for scene-content errors less — inverse of the planning need. Fixable later? Yes, at stage (b) cost: auxiliary losses, feature weighting, shorter horizons — all mitigations, none free. **5. Evaluation protocol for the composed stack.** Bake in at (a) an evaluation the later stages must satisfy: kNN retrieval consistency of the frozen space (is semantic nearness geometrically meaningful?), then in (b) multi-step rollout consistency (does ϕ compose with itself without drifting off the feature manifold), and only then planner success on goal-specified tasks (c). The avoidable failure this ordering prevents: debugging a planner failure that is, in truth, a stage-(a) representational blindness — the classic attribution error of stacked self-supervised systems, and the reason the decision list is ordered by irreversibility.

End of the exercise chapter. Concept maps for where each method family connects forward and backward in the book are collected in each technical chapter's closing box; this chapter's job was to make every one of those connections answerable.

Part VI

Reference

CHAPTER A

Mathematical Background

This appendix is a refresher, not a textbook. It collects the definitions and results actually used in the main chapters, states them precisely, and closes each topic with one worked example that mirrors how the result is applied in this book. If a chapter cites this appendix, it means you can get by with what is written here.

How to use this appendix

You do not read this appendix front to back. You come here when a chapter uses something you do not recognise, read the one topic you are stuck on, and go back. Three habits make that work.

1. **Read the grey "In plain words" box first, always.** Every topic below opens with one. It says what the object *is* with no symbols. If the plain-words box alone unsticks you, stop there and return to the chapter — you do not owe anyone the formal version.
2. **Then read the formal statement, and name every symbol out loud.** A formula you cannot read aloud is a formula you cannot use. If a symbol is unfamiliar, look it up in the Notation table in the front matter, or in Section A.6 at the end of this appendix, which catalogues every symbol the book uses that the Notation table does not.
3. **Then do the worked example with a pen.** Each section ends with one. The numbers are small on purpose: you can check them by hand in a minute, and doing so converts "I have seen this" into "I can do this".

In plain words — what this appendix is for

Nothing here is new material. Everything here is a tool the main chapters pick up and use. The five sections are, in order: **probability** (how to talk about uncertain things and average over them), **linear algebra** (vectors, distances, and what a matrix does to space), **calculus** (slopes, gradients, and the chain rule), **information theory** (how to measure surprise and disagreement between distributions), and **optimization** (how a network actually gets better). If you know all five, skip to the chapters. If you know

none, read the plain-words boxes only — about fifteen minutes — and come back for detail when a chapter forces you to.

New words used from here on — five words that appear everywhere below

- **Scalar** — a single number, like 3.7.
- **Vector** — an ordered list of numbers, like $(3.7, -1, 0)$. We write vectors in bold: $\boldsymbol{x}$, $\boldsymbol{z}$. A d-dimensional vector is a point in $\mathbb{R}^d$.
- **Matrix** — a rectangular grid of numbers. A matrix acting on a vector produces another vector; that is the only thing matrices do in this book.
- **Distribution** — a rule that says how likely each possible value is. Written p.
- **Map / function** — anything that takes an input and returns an output. A neural network is a map.

A.1 Probability and Statistics

Needed in Chapters 4 (ELBO), 6 (change of variables, GAN divergence), 7 (flow matching, conditional sampling), 8–11 (softmax over sampled sets), *and* 12 (soft targets).

In plain words — probability, in one paragraph

A *random variable* is a quantity whose value you do not know yet: the next image drawn from a dataset, the noise added to a pixel, the class a classifier will pick. A *distribution* says how the possible values share out the total certainty of 1. For things you can count (which of ten classes?) the distribution gives each option a probability. For things you measure on a continuous scale (a pixel brightness) there is no probability of any exact value, so we use a *density* instead: a curve whose height says how concentrated the possibilities are, and whose total area is 1.

Random variables, expectations, common distributions. A discrete random variable X takes values x with probabilities $p(X = x)$; a continuous one has a density $p(x)$ with $\int p(x)\,dx = 1$. The expectation is $\mathbb{E}_{x\sim p}[g(x)] = \sum_x p(x)g(x)$ or $\int p(x)g(x)\,dx$.

In plain words — what an expectation is

An *expectation* is a weighted average. Nothing more. You have a quantity $g(x)$ that depends on a random outcome x, and you want one number summarising it. So you take every outcome, multiply the value $g(x)$ by how likely that outcome is, and add everything up. Outcomes that happen often count for a lot; outcomes that almost never happen barely count.

The notation $\mathbb{E}_{x\sim p}[g(x)]$ reads, left to right: "the average value of $g(x)$, when x is drawn from the distribution p". The little $x \sim p$ is the only part that is easy to skip and

important to keep — it says *which* weighting you are averaging with. The same g averaged under two different p's gives two different numbers.
Two consequences worth remembering. An expectation is a *number*, not a function — once you average, the randomness is gone. And if you cannot do the sum or integral (you almost never can, for a neural network), you approximate it by drawing samples and taking the plain average of g over them. That is all a minibatch is.

Three distributions appear so often that we fix them here:

- **Categorical/softmax.** Given logits $s_1, \dots, s_K$, the softmax probability of class c is $\text{softmax}(s)_c = e^{s_c} / \sum_j e^{s_j}$. A *logit* is just an unconstrained score the network produced: it may be negative, it may be huge, and it does not sum to anything in particular. Exponentiating makes every score positive; dividing by the total makes them sum to one. Nearly every loss in Part II is a softmax cross-entropy in disguise (InfoNCE, DINO, pseudo-labeling).
- **Gaussian.** $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ on $\mathbb{R}^d$ has density $(2\pi)^{-d/2}|\boldsymbol{\Sigma}|^{-1/2} \exp\big(-\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\big)$. The log-likelihood of a Gaussian with *fixed* variance, $\log \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \eta I) = -\frac{1}{2\eta}\|\boldsymbol{x}-\boldsymbol{\mu}\|^2 + \text{const}$, is why squared L2 error and Gaussian likelihood are the same objective — the link exploited by autoencoders (Ch. 4) and diffusion (Ch. 7).
- **Uniform on** $[0, 1]$**.** Used in flow matching to sample the interpolation time $t \sim U[0, 1]$ (Ch. 7).

In plain words — what a Gaussian's mean and covariance control

A Gaussian is the "blob" distribution. It has exactly two knobs, and they do completely separate jobs.
The **mean** $\boldsymbol{\mu}$ says *where* the blob sits. It is the centre of the cloud, the average point. Moving $\boldsymbol{\mu}$ slides the whole cloud without changing its shape at all.
The **covariance** $\boldsymbol{\Sigma}$ says *what shape* the blob is: how wide, in which directions, and whether the coordinates lean on each other. Read it entry by entry. The diagonal entry Σ_{jj} is the variance of coordinate j — how far that coordinate typically strays from its mean, squared. The off-diagonal entry Σ_{ij} is the covariance between coordinates i and j: positive means "when i is above average, j tends to be too", negative means they pull opposite ways, and zero means knowing one tells you nothing about the other.
Three shapes cover almost every use in this book (Figure A.1). *Isotropic*, $\boldsymbol{\Sigma} = \sigma^2 I$: a round ball, same spread in every direction, no coordinate related to any other. This is the prior $\mathcal{N}(0, I)$ a VAE pushes its latents toward (Ch. 4) and the noise a diffusion model adds (Ch. 7). *Diagonal*, $\boldsymbol{\Sigma} = \text{diag}(\sigma_1^2, \dots, \sigma_d^2)$: an axis-aligned ellipse, each coordinate free to have its own spread but still unrelated to the others. This is what a VAE encoder actually outputs — one mean and one variance per latent coordinate — and it is why the KL term factorises into a sum over j. *Full*: a tilted ellipse, coordinates correlated. This is the shape Barlow Twins is fighting (Ch. 10): it wants the cross-correlation matrix to be

the identity, that is, it wants the tilt gone, because a tilt means two feature dimensions are carrying the same information twice.

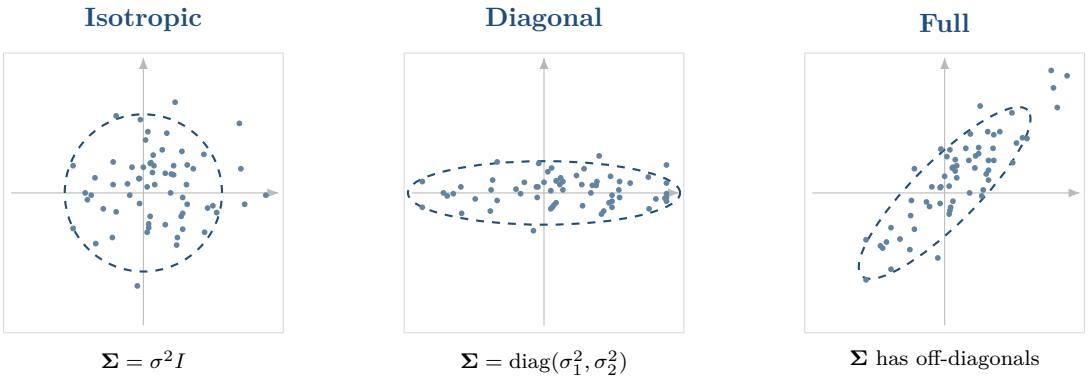


Figure A.1. The same 64 random draws, pushed through three different covariance matrices. The mean is the same in all three panels (the origin); only $\boldsymbol{\Sigma}$ changes. Isotropic gives a round cloud; a diagonal $\boldsymbol{\Sigma}$ stretches along the coordinate axes; a $\boldsymbol{\Sigma}$ with non-zero off-diagonal entries tilts the cloud, which is exactly what "these two coordinates are correlated" looks like.

How to read it: compare the dashed ellipse in each panel. It is the one-standard-deviation contour. Round means "no direction is special". Axis-aligned means "each coordinate has its own scale, but they are independent". Tilted means "knowing coordinate 1 tells you something about coordinate 2" — redundancy, and the thing Barlow Twins removes.

Expectation identities. Linearity: $\mathbb{E}[aX + bY] = a\mathbb{E}X + b\mathbb{E}Y$. Variance: $\mathrm{Var}(X) = \mathbb{E}X^2 - (\mathbb{E}X)^2$. A *Monte Carlo estimate* replaces an expectation by a sample average, $\mathbb{E}_{x\sim p}[g(x)] \approx \frac{1}{L}\sum_{l=1}^{L} g(x^{(l)})$ with $x^{(l)} \sim p$; the VAE approximates its reconstruction term exactly this way (Ch. 4).

Conditional probability and Bayes' rule. The conditional probability of x given y is $p(x \mid y) = p(x, y)/p(y)$, and *Bayes' rule* rearranges it into

$$\underbrace{p(x \mid y)}_{\text{posterior}} = \frac{\overbrace{p(y \mid x)}^{\text{likelihood}}\ \overbrace{p(x)}^{\text{prior}}}{\underbrace{p(y)}_{\text{evidence}}}.$$

This underlies the derivation of classifier-free guidance as extrapolation between conditional and unconditional scores (Ch. 7), and it is the reason a VAE's encoder is called an *approximate posterior* (Ch. 4).

In plain words — conditional probability and Bayes' rule, in words

Conditional probability is what is left after you rule things out. $p(x \mid y)$ — read "probability of x *given* y" — means: forget every world in which y did not happen, and ask how likely x is among the worlds that remain. That is literally the formula. $p(x, y)$ counts the worlds where both happened; dividing by $p(y)$ rescales them so the surviving worlds add up to 1 again.

Bayes' rule is how you update a belief when evidence arrives. You start with a belief about x before seeing anything: the *prior* $p(x)$. Then you observe y. You ask, for each possible x, "if x were true, how likely was I to see this y?" — the *likelihood* $p(y \mid x)$. Multiply the two: a candidate x survives only if it was plausible *and* it explains what you saw. Finally divide by $p(y)$, which does not depend on x at all and is only there to make the numbers add to one. The result, $p(x \mid y)$, is the *posterior* — your belief after the evidence.

The sentence to keep: *posterior* $\propto$ *prior* $\times$ *likelihood*. Everything else is bookkeeping. In this book, x is usually a hidden cause (a latent code, a class, a clean image) and y is what you actually see (an image, a noisy image). "Inference" always means going backwards from the visible y to the hidden x, and going backwards always means Bayes.

Worked Example — Bayes' rule with numbers you can check

A screening test for a condition that 1 person in 100 has. The test catches 90% of true cases ($p(\text{positive} \mid \text{ill}) = 0.9$) and wrongly fires on 5% of healthy people ($p(\text{positive} \mid \text{well}) = 0.05$). You test positive. How worried should you be?

Prior: $p(\text{ill}) = 0.01$, so $p(\text{well}) = 0.99$. Evidence: $p(\text{positive}) = 0.9 \times 0.01 + 0.05 \times 0.99 = 0.009 + 0.0495 = 0.0585$. Posterior:

$$p(\text{ill} \mid \text{positive}) = \frac{0.9 \times 0.01}{0.0585} = \frac{0.009}{0.0585} \approx 0.1538.$$

About 15% — not 90%. The prior did most of the work: healthy people are so much more common that their 5% false-positive rate produces five times more positives than the ill people's 90% true-positive rate. This is the single most useful sanity check in the appendix: *a likelihood is not a posterior*, and ignoring the prior is the most common probability mistake there is.

Maximum likelihood and cross-entropy. Given i.i.d. samples $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$, maximum likelihood picks $\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_i p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$. Taking logs ("log trick", numerical safety) gives $\arg\max_{\boldsymbol{\theta}} \sum_i \log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$, and negating turns it into a minimization. When the model outputs a softmax distribution q over K classes and the target is the one-hot vector for class c, the loss is the *cross-entropy* $-\log q_c$. Datasets do not change the math: minimizing average cross-entropy on the data *is* maximum likelihood (Ch. 5, 8, 9, 11).

In plain words — maximum likelihood

Maximum likelihood asks one question: *of all the settings of the parameters I could choose, which one makes the data I actually collected look least surprising?* You multiply together the model's probability for every observed example and pick the parameters that make that product biggest. Products of thousands of small numbers underflow to zero on a computer, so we take a logarithm and add instead of multiplying — the answer is the same, because the logarithm never reorders things. Then we flip the sign, because optimizers are written to go downhill. "Minimize negative log-likelihood" and "maximize likelihood" are the same instruction said twice.

Worked Example — Why L2 regression *is* maximum likelihood

Setting (Chapter 7, flow matching). We train $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ to regress the velocity $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$ with loss $\|f_{\boldsymbol{\theta}} - \boldsymbol{v}\|^2$. Why not a fancier objective? *Computation.* Assume at fixed $(\boldsymbol{x}_t, t)$ the target velocity is Gaussian around the truth: $v \sim \mathcal{N}(\boldsymbol{v}, \eta I)$ for some unknown observation spread η. Then

$$\log p_{\boldsymbol{\theta}}(v) = -\frac{1}{2\eta}\|v - f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\|^2 - \frac{d}{2}\log(2\pi\eta).$$

Maximizing the log-likelihood over $\boldsymbol{\theta}$ drops the constant term and the positive factor $1/(2\eta)$, leaving exactly $\min_{\boldsymbol{\theta}} \|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{v}\|^2$ when $v = \boldsymbol{v}$ is observed. The same equivalence explains why a Gaussian decoder turns the VAE reconstruction term into squared error (Ch. 4): L2 losses are Gaussian likelihoods in disguise.

Key takeaway — probability

An expectation is a weighted average, and a minibatch average is how you compute one when you cannot do the sum. Bayes' rule is *posterior* $\propto$ *prior* $\times$ *likelihood* and is how every "recover the hidden cause" step in this book works. A Gaussian has two knobs: the mean moves the cloud, the covariance shapes it.

A.2 Linear Algebra

Needed in Chapters 2–3 (kNN distances, PCA, clustering), 6 (Jacobians of flows), 7 (trace estimators), 10–11 (cross-correlation regularizers).

In plain words — what linear algebra is doing here

Two jobs, and it helps to keep them apart. The first job is **measuring**: a representation is a point in space, and we constantly need to ask how far apart two points are, or how aligned two directions are. That is the dot product and the norms below. The second job is **transforming**: a matrix takes every point in space and moves it somewhere else, and we need to know what that does to volumes, because generative models track

probability mass and probability mass lives in volume. That is the determinant and the Jacobian below. Measuring dominates Chapters 2–3 and 9–11; transforming dominates Chapters 6–7.

Vectors, angles, distances. For $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$: the dot product is $\boldsymbol{u}^\top \boldsymbol{v} = \sum_i u_i v_i$; the Euclidean norm is $\|\boldsymbol{u}\|_2^2 = \boldsymbol{u}^\top \boldsymbol{u}$; the cosine *similarity* is $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v}) = \dfrac{\boldsymbol{u}^\top \boldsymbol{v}}{\|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2} = \cos\theta$, the cosine *distance* is $1 - \cos\theta$. Cosine similarity ignores magnitude, which is learned to vary with confidence in some methods — hence the lecture's advice (Ch. 2): check whether your training loss optimizes a distance, and which one. Because cosine similarity is cos of the angle, unit-normalized embeddings live on a hypersphere, and Euclidean distance between them satisfies $\|\boldsymbol{u} - \boldsymbol{v}\|_2^2 = 2 - 2\cos\theta$: sorting by Euclidean distance and by cosine similarity gives the *same* neighbor order. This equivalence is why kNN evaluation (Ch. 2) and InfoNCE (Ch. 9) interchange the two without comment.

In plain words — dot product, norm, cosine

The **norm** $\|\boldsymbol{u}\|_2$ is the length of the arrow from the origin to $\boldsymbol{u}$ — Pythagoras in d dimensions. The **dot product** $\boldsymbol{u}^\top \boldsymbol{v}$ measures how much the two arrows point the same way, scaled by both their lengths: large and positive if they agree, zero if they are at right angles, negative if they oppose. **Cosine similarity** divides that by both lengths, which cancels the scaling and leaves pure direction: $+1$ same direction, 0 perpendicular, -1 opposite. When people say "normalize the embeddings", they mean divide each vector by its own length so every point sits on a sphere of radius one; after that, dot product and cosine similarity are the same thing.

Matrix decompositions.

- **Eigen-decomposition / PCA.** A symmetric covariance matrix $\boldsymbol{\Sigma} = \frac{1}{N} X^\top X$ (centered data) decomposes as $\boldsymbol{\Sigma} = V \boldsymbol{\Lambda} V^\top$ with orthonormal V and eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq 0$. PCA keeps the top-$k$ columns of V and projects $\boldsymbol{z} = V_k^\top (\boldsymbol{x} - \bar{\boldsymbol{x}})$; the kept variance fraction is $\sum_{i \leq k} \lambda_i / \sum_i \lambda_i$ (Jolliffe, 2002). In words: find the directions in which the data spreads out the most, and keep only those.
- **Singular value decomposition.** $X = U \Sigma V^\top$; PCA's eigenvalues are the squared singular values divided by N. Useful because U, V are orthogonal and Σ diagonal — properties coders rely on when reasoning about Jacobians. In words: *every* matrix, square or not, is a rotation, then a stretch along axes, then another rotation.
- **LU decomposition.** A square matrix factors as $W = PL(U + \mathrm{diag}(s))$ (permutation P, unit lower-triangular L, strictly upper-triangular U, diagonal s). Then $\log|\det W| = \sum_i \log|s_i|$ at cost $O(c)$ instead of $O(c^3)$ — exactly the trick that makes Glow's invertible 1×1 convolutions tractable (Ch. 6) (Kingma and Dhariwal, 2018).

- **QR / orthogonality.** ViTs with weight decay, and the visibility of "whitening" in Barlow-type losses, both reference near-orthogonality; always remember that $Q^\top Q = I$ makes the Jacobian of an orthogonal map have $|\det| = 1$. An orthogonal matrix rotates and reflects but never stretches, so it cannot change volume.

Determinants: the volume story. For a square matrix $A \in \mathbb{R}^{d\times d}$, the determinant $\det A$ is the signed factor by which the linear map $\boldsymbol{x} \mapsto A\boldsymbol{x}$ scales volume. A set of volume V is carried to a set of volume $|\det A| V$. Two facts follow and are used constantly in Chapter 6: $\det(AB) = \det A \det B$ (compose maps, multiply the factors), and the determinant of a triangular matrix is the product of its diagonal entries.

In plain words — what a determinant actually is

Take the unit square: the little square with corners at $(0,0)$, $(1,0)$, $(1,1)$, $(0,1)$. Its area is 1. Now apply the matrix to it. The square becomes a parallelogram — the two sides are wherever the matrix sent the two unit arrows. **The determinant is the area of that parallelogram.** That is the whole definition. In three dimensions, replace "square" with "cube" and "area" with "volume"; in d dimensions the words run out but the idea does not.

So $\det A = 3$ means "this map triples volume everywhere". $\det A = 0.5$ means it halves it. $\det A = 1$ means it moves things around but does not squash or inflate — a rotation does this. And $\det A = 0$ is the fatal case: the map has flattened space onto something lower-dimensional, so it has thrown information away and cannot be undone. That is precisely why normalizing flows (Ch. 6) insist on $\det J \neq 0$: they must be reversible.

The sign is a bonus, not a mystery: a negative determinant means the map also flipped orientation, like a mirror. Because we only ever care about volume, the formulas in this book always use $|\det|$.

Why does a generative model care? Because probability behaves like a fluid of fixed total amount. Squeeze a region to half its volume and the probability *density* there must double, or you have created probability out of nothing. The $\log|\det J|$ term in the change-of-variables formula is exactly that bookkeeping correction, and nothing more.

Jacobians and determinants. For a differentiable map $g : \mathbb{R}^d \to \mathbb{R}^d$, the Jacobian $J_g(\boldsymbol{x}) = \partial g / \partial \boldsymbol{x}$ collects partial derivatives; its determinant measures local volume change. Rule-of-thumb consequences used in Ch. 6–7:

- If J is triangular, $\det J$ is the product of the diagonal — the reason affine coupling layers (Real NVP, Ch. 6) have $O(d)$ log-determinants.
- $\det(AB) = \det A \det B$, so composing flows adds their log-determinants.
- The trace estimator of Hutchinson (1990), $\operatorname{tr}(A) = \mathbb{E}_{\boldsymbol{\epsilon}}[\boldsymbol{\epsilon}^\top A \boldsymbol{\epsilon}]$ for zero-mean, identity-covariance $\boldsymbol{\epsilon}$, replaces exact traces in continuous normalizing flows (Ch. 7) — no closed form, but $O(1)$ samples suffice in practice.

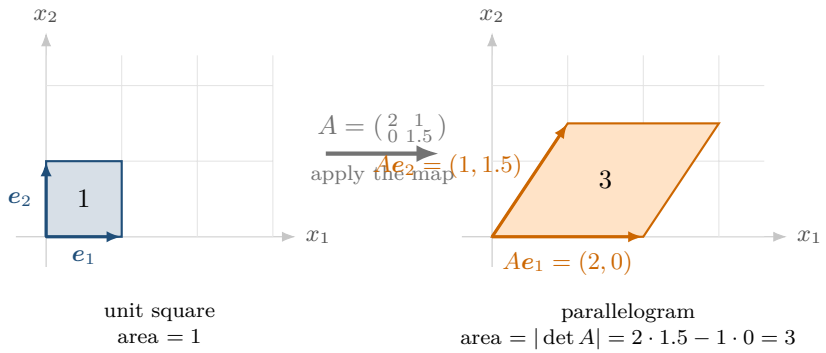


Figure A.2. The determinant is a volume-scaling factor. The map A sends the two unit arrows $\boldsymbol{e}_1, \boldsymbol{e}_2$ to the columns of A; the unit square they spanned becomes the parallelogram those columns span. Its area, 3, is $|\det A|$. Every region of the plane — not just this square — has its area multiplied by 3.

How to read it: count squares. The blue region covers one grid cell; the orange region covers three. That ratio is *the determinant. If the orange shape had collapsed onto a single line it would have area zero, the determinant would be zero, and the map could not be undone.*

In plain words — what a Jacobian is

A determinant is about a *linear* map — one matrix, the same everywhere. Real networks are not linear. A Jacobian is what you use instead.

Zoom far enough into any smooth map at any single point and it stops looking curved; it looks like a straight stretch-and-rotate. **The Jacobian is that local straight-line approximation, written as a matrix.** Entry (i, j) is $\partial g_i / \partial x_j$: "if I nudge input coordinate j a little, how much does output coordinate i move?" Collect all $d \times d$ of those answers and you have the best linear picture of the map right there.

Two things follow immediately. First, the Jacobian is a *function of where you are* — $J_g(\boldsymbol{x})$, not J_g. A map can double volume near one point and halve it near another. Second, since the Jacobian is the local matrix, its determinant is the **local** volume-scaling factor: how much a tiny blob at $\boldsymbol{x}$ swells or shrinks as it passes through g. That is the only property Chapter 6 ever needs, and it is why $\log|\det J_g(\boldsymbol{x})|$ appears once per flow layer in the change-of-variables formula.

One more useful reading: for a scalar-valued function the Jacobian has a single row, and that row is the gradient. Gradient and Jacobian are the same idea at different output widths.

Common Confusion

"The Jacobian" sometimes means the matrix and sometimes means its determinant. Papers write "the Jacobian of a coupling layer is cheap" when they mean "the *determinant* of its Jacobian matrix is cheap". Read which one is meant from the context:

if it is being multiplied by a vector it is the matrix; if it is being added to a log-likelihood it is $\log|\det J|$, a single number.

Worked Example — Log-determinant of an affine coupling layer

Setting (Ch. 6, Real NVP). Split $\boldsymbol{x} \in \mathbb{R}^4$ as $(x_1, x_2) \mid (x_3, x_4)$. An affine coupling fixes the first half and scales the second with scale $s(x_1, x_2) = (1, 0)$ and shift $t(x_1, x_2) = (0, 1)$:

$$y_1 = x_1, \quad y_2 = x_2, \quad y_3 = x_3 e^1 + 0 = e\, x_3, \quad y_4 = x_4 e^0 + 1 = x_4 + 1.$$

Jacobian.

$$J = \partial \boldsymbol{y} / \partial \boldsymbol{x}^\top = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ * & * & e^1 & 0 \\ * & * & 0 & e^0 \end{pmatrix}, \qquad \log|\det J| = 1 + 0 = 1.$$

The $*$ entries — derivatives through $s(x_1, x_2)$ — do not enter the determinant because the matrix is (block) lower-triangular. So the layer's contribution to the change-of-variables log-likelihood (Ch. 6) is simply the sum of the log-scales, computed in $O(d)$ time; that is the entire point of the architecture.

Sanity check in volume language. $\det J = e^1 \cdot e^0 = e \approx 2.718$. The layer leaves coordinates $1, 2$ alone, stretches coordinate 3 by e, and merely slides coordinate 4 along. So a tiny box around any point comes out e times bigger — and $\log e = 1$, which is the number above.

Key takeaway — linear algebra

Dot products and norms answer "how close / how aligned". Determinants answer "by how much did this squash space". A Jacobian is the determinant question asked about a curved map, one point at a time. Triangular matrices are loved in this book for exactly one reason: their determinant is the product of the diagonal, so it costs $O(d)$ instead of $O(d^3)$.

A.3 Calculus: Derivatives, Gradients, and the Chain Rule

Needed in every chapter that trains anything — so, all of them. Most directly in Chapters 2 (SGD), 4 (reparameterized gradients), 7 (score functions, which are gradients of a log-density) and 13 (planning as optimization over actions).

In plain words — what a gradient is

Start with one input. The **derivative** of a function at a point is its slope there: if you nudge the input by a hair, how much does the output move, and in which direction? Positive slope means "increasing the input increases the output".

Now allow many inputs — millions of them, which is what a network's parameters are. There is no single slope any more, because you can nudge in many different directions. So you record one number per input: the *partial derivative* $\partial\mathcal{L}/\partial\theta_j$, which is the slope you would feel if you nudged parameter j and held every other parameter still. Stack all those numbers into one vector and you have the **gradient**, written $\nabla_{\boldsymbol{\theta}}\mathcal{L}$.

Here is the fact that makes the gradient worth a name, rather than just being "a list of slopes". Out of all the directions you could step in, the gradient points in the direction where the function increases *fastest*, and its length says how steep that steepest climb is. So $-\nabla_{\boldsymbol{\theta}}\mathcal{L}$ points straight downhill, which is why every training loop in this book takes a small step in that direction and repeats. A gradient of zero means locally flat — no direction improves anything — which is what "converged" means, and also what "stuck" means.

Two readings worth keeping. A gradient is an *arrow in parameter space*, not a number: it has a direction and a length. And it is *local*: it describes the landscape only at the point where you computed it, which is why you must recompute it after every step.

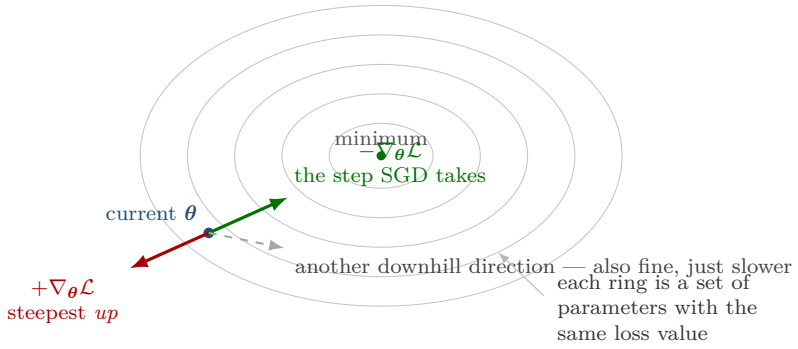


Figure A.3. The gradient of a loss, drawn on its contour map. Rings join parameter settings with equal loss, like height lines on a hiking map. At the blue point the gradient (red) is the direction of steepest *increase*; it is always perpendicular to the ring through that point. Gradient descent steps the other way (green).

How to read it: note that the red and green arrows cross the rings at right angles, while the grey dashed arrow does not. Both the green and the dashed arrows reduce the loss; only green reduces it fastest per unit of step length. That is the entire content of "the gradient is the direction of steepest ascent".

The chain rule. If $\mathcal{L}$ depends on $\boldsymbol{\theta}$ through intermediate quantities, the derivative multiplies along the path: for $\mathcal{L} = h(g(f(\boldsymbol{\theta})))$,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \frac{\partial h}{\partial g}\,\frac{\partial g}{\partial f}\,\frac{\partial f}{\partial \boldsymbol{\theta}}.$$

Backpropagation is this identity applied layer by layer, from the loss backwards to the parameters, reusing each partial result exactly once.

In plain words — the chain rule, and why backpropagation exists

A network is a chain of stages: input, layer, layer, …, loss. Suppose you want to know how much wiggling a weight in the *first* layer changes the loss at the *end*. The chain rule says: work out how much the weight moves its own layer's output, how much that moves the next layer's output, and so on to the loss — then multiply all those factors together. Sensitivities multiply along a chain, exactly as gear ratios do.

Backpropagation is not a different idea. It is the chain rule evaluated in the economical order: start at the loss, walk backwards, and carry along the running product so far. Doing it forwards would recompute the same products once per parameter; doing it backwards computes them once in total. That is the whole trick, and it is why one backward pass costs about the same as one forward pass no matter how many parameters there are.

Common Confusion

A gradient does not tell you how big a step to take. It gives a direction and a steepness, not a distance to the minimum. The step size is the learning rate η, a separate choice. A large gradient on a sharply curved loss can easily overshoot, which is what a diverging training run looks like.

Worked Example — A gradient you can do in your head

Let $\mathcal{L}(\theta_1, \theta_2) = (\theta_1 - 3)^2 + 4\theta_2^2$ and take the current point $(\theta_1, \theta_2) = (1,\, 0.5)$.

Partial derivatives: $\partial\mathcal{L}/\partial\theta_1 = 2(\theta_1 - 3) = 2(1 - 3) = -4$, and $\partial\mathcal{L}/\partial\theta_2 = 8\theta_2 = 8(0.5) = 4$. So $\nabla\mathcal{L} = (-4, 4)$. Read it: increasing θ_1 *lowers* the loss (the slope is negative), increasing θ_2 raises it.

One gradient-descent step with learning rate $\eta = 0.1$: $\theta_1 \leftarrow 1 - 0.1(-4) = 1.4$ and $\theta_2 \leftarrow 0.5 - 0.1(4) = 0.1$. Check that the loss went down: before, $(1-3)^2 + 4(0.5)^2 = 4 + 1 = 5$; after, $(1.4 - 3)^2 + 4(0.1)^2 = 2.56 + 0.04 = 2.60$. It did.

Now notice the asymmetry, because it is the reason Adam exists. The θ_2 direction is four times more curved than the θ_1 direction, so the same learning rate that is comfortable for θ_1 is nearly too aggressive for θ_2. Per-coordinate step-size scaling is exactly the problem adaptive optimizers solve.

Key takeaway — calculus

The gradient is the list of "how much does the loss change if I nudge this one number", packed into a vector that happens to point straight uphill. Training steps the other way. The chain rule multiplies sensitivities along a chain; backpropagation is that rule, computed back-to-front so the work is done once.

A.4 Information Theory

Needed in Chapters 4 (ELBO = reconstruction − KL), 9 (InfoNCE and the MI lower bound), 4 and 12 (soft-target cross-entropies), 15 (the MI explanations of why SSL works).

In plain words — the one idea behind this whole section

Information theory measures things in units of *surprise*. If an event was almost certain and it happens, you learned nearly nothing; if it was unlikely and it happens, you learned a lot. The formal version of "how surprised was I" is $-\log p(x)$: it is zero when $p(x) = 1$ and grows without bound as $p(x) \to 0$.
Everything below is an average of that quantity under some distribution. **Entropy** is your average surprise using the right probabilities. **Cross-entropy** is your average surprise when you used the *wrong* probabilities. **KL divergence** is the gap between the two — the extra surprise you paid for being wrong. **Mutual information** is how much knowing one thing reduces your surprise about another. Read the formulas below with that sentence in hand and they stop being arbitrary.

Entropy, KL, cross-entropy. For distributions p, q over the same set:

- **Entropy** $H(p) = -\sum_x p(x) \log p(x)$ — average code length under p's own coding. High entropy means spread out and unpredictable; low entropy means peaked and predictable.
- **Cross-entropy** $H(p, q) = -\sum_x p(x) \log q(x) = H(p) + D_{\mathrm{KL}}(p \,\|\, q)$ — the expected code length when coding p with q.
- **KL divergence** $D_{\mathrm{KL}}(p \,\|\, q) = \sum_x p(x) \log \dfrac{p(x)}{q(x)} \geq 0$, with equality iff $p = q$. It is *not* symmetric. For two Gaussians in k dimensions it has the closed form (used in every VAE chapter):

$$D_{\mathrm{KL}}(\mathcal{N}(\boldsymbol{\mu}_q, \boldsymbol{\Sigma}_q) \,\|\, \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)) = \tfrac{1}{2}\Big(\operatorname{tr}\left(\boldsymbol{\Sigma}_p^{-1}\boldsymbol{\Sigma}_q\right) + (\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)^\top \boldsymbol{\Sigma}_p^{-1}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q) - k + \log \tfrac{|\boldsymbol{\Sigma}_p|}{|\boldsymbol{\Sigma}_q|} \Big).$$

  With $\boldsymbol{\mu}_p = \mathbf{0}$, $\boldsymbol{\Sigma}_p = I$ and diagonal $\boldsymbol{\Sigma}_q = \operatorname{diag}(\sigma_1^2, \ldots, \sigma_k^2)$ this collapses to the familiar $\tfrac{1}{2}\sum_j (1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2)$, negated, in the VAE ELBO (Kingma and Welling, 2019).

Common Confusion

KL is not a distance. $D_{\mathrm{KL}}(p\|q)$ and $D_{\mathrm{KL}}(q\|p)$ are different numbers, and neither obeys the triangle inequality. The asymmetry has teeth: $D_{\mathrm{KL}}(q\|p)$ is huge wherever q puts mass and p does not, so minimizing it makes q hide *inside* p (mode-seeking), whereas $D_{\mathrm{KL}}(p\|q)$ punishes q for ignoring any region p covers, so minimizing it makes q spread out over everything (mass-covering). A VAE uses both: its encoder minimizes the mode-seeking $D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}\|p_{\mathrm{posterior}})$, while maximum-likelihood training of the decoder is the mass-covering $D_{\mathrm{KL}}(p_{\mathrm{data}}\|p_{\mathrm{model}})$ — which, together with the Gaussian decoder's averaging, is why VAE samples come out smooth rather than sharp.

Mutual information. $I(X;Z) = D_{\mathrm{KL}}(p(x,z)\,\|\,p(x)p(z)) = H(X) - H(X\,|\,Z)$. In words: how far the joint distribution of X and Z is from the distribution they would have if they were unrelated — equivalently, how much of your uncertainty about X disappears once someone tells you Z. It is zero exactly when the two are independent. Beyond this book, the only properties we lean on are non-negativity, the data-processing inequality ($I(X;Z) \geq I(X;g(Z))$ for deterministic g — post-processing can destroy information but never create it), and the variational bound

$$I(X;Z) \;\geq\; H(X) + \mathbb{E}\big[\log q(x\,|\,z)\big] \quad \text{for any decoder } q,$$

which turns mutual information into a trainable objective (the contrastive/InfoNCE instantiation is the one used in Ch. 9, with $I(\boldsymbol{f}(\boldsymbol{x});\boldsymbol{f}(\boldsymbol{x}^+)) \geq \log(K+1) - \mathcal{L}_{\mathrm{NCE}}$ for K negatives (van den Oord et al., 2018b; Poole et al., 2019)).

Jensen's inequality. For convex φ, $\varphi(\mathbb{E}[X]) \leq \mathbb{E}[\varphi(X)]$; reversed for concave φ. Since log is concave, $\mathbb{E}[\log X] \leq \log \mathbb{E}[X]$ — the single step that turns the intractable $\log p(\boldsymbol{x})$ of a latent-variable model into the tractable ELBO (Ch. 4).

In plain words — Jensen's inequality, with no symbols

A function is *convex* if it curves upwards, like a bowl. Pick any two points on a bowl-shaped curve and join them with a straight line. That line sits above the curve everywhere in between — that is what "curves upwards" means, drawn.

Now read the two sides of the inequality off the picture (Figure A.4). "Average the inputs first, then apply the function" lands you *on the curve*, at the point above the average input. "Apply the function first, then average the outputs" lands you *on the straight line*, because averaging two output heights is exactly what the chord does. The line is above the curve, so $\varphi(\mathbb{E}[X]) \leq \mathbb{E}[\varphi(X)]$. Nothing else is going on.

Flip the curve upside down and you get a *concave* function, like log, and the chord now sits below the curve, so the inequality reverses: $\mathbb{E}[\log X] \leq \log \mathbb{E}[X]$. That reversed version is used exactly once in this book, in Chapter 4, and it is the entire reason the

ELBO is a *lower* bound: we swap the impossible log-of-an-average for the computable average-of-a-log and accept that we have moved downwards by doing so.

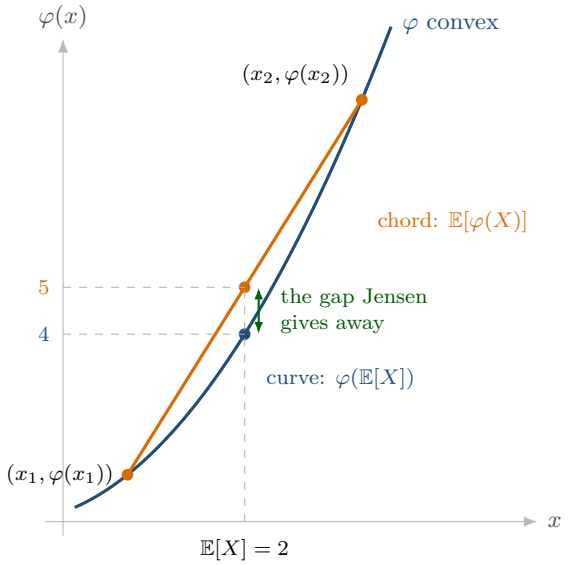


Figure A.4. Jensen's inequality is a picture, not an algebraic trick. With $\varphi(x) = x^2$ and X equal to 1 or 3 with probability $\frac{1}{2}$ each: averaging first gives $\varphi(\mathbb{E}[X]) = \varphi(2) = 4$ (on the curve), while applying φ first gives $\mathbb{E}[\varphi(X)] = \frac{1}{2}(1) + \frac{1}{2}(9) = 5$ (on the chord). The chord of a convex function lies above it, so $4 \leq 5$.

How to read it: the two dots at $x = 2$ are the two sides of the inequality. The blue one sits on the curve, the orange one on the straight chord, and the green arrow is the slack. For a concave *function such as* log *the picture flips vertically, the chord falls below the curve, and the same slack becomes the gap between* $\log p(\boldsymbol{x})$ *and the ELBO in Chapter 4.*

Worked Example — ELBO via Jensen, with numbers

Setting (Ch. 4). One data point $\boldsymbol{x}$, one latent sample $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$, prior $p(\boldsymbol{z}) = \mathcal{N}(0, I)$. Define $F(\boldsymbol{z}) = p(\boldsymbol{z})p(\boldsymbol{x} \mid \boldsymbol{z})/q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$. Then

$$\log p(\boldsymbol{x}) = \log \int q_{\boldsymbol{\phi}}(\boldsymbol{z}) \, \frac{p(\boldsymbol{z}, \boldsymbol{x})}{q_{\boldsymbol{\phi}}(\boldsymbol{z})} \, d\boldsymbol{z} = \log \mathbb{E}_{q_{\boldsymbol{\phi}}}[F(\boldsymbol{z})] \;\geq\; \mathbb{E}_{q_{\boldsymbol{\phi}}}\big[\log F(\boldsymbol{z})\big],$$

by Jensen applied to the concave log. Expanding $\log F$ splits the bound into $\mathbb{E}[\log p(\boldsymbol{x} \mid \boldsymbol{z})] - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}} \,\|\, p)$. *Concrete check.* Let $\boldsymbol{x} \in \mathbb{R}^2$, decoder mean $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{z}) = \boldsymbol{z}$, noise $\eta = \frac{1}{2}$; $\boldsymbol{z} = (1, 0)^\top$, $\boldsymbol{x} = (1.1,\, 0.2)^\top$; encoder has mean $\boldsymbol{\mu}_q = (0.5, 0)^\top$, $\sigma_q = 1$. Reconstruction: $-\frac{1}{2\eta}\|\boldsymbol{x} - \boldsymbol{z}\|^2 + \text{const} = -\|\boldsymbol{x} - \boldsymbol{z}\|^2 + \text{const} = -(0.1^2 + 0.2^2) + \text{const} = -0.05 + \text{const}$. Prior term: $\frac{1}{2}\sum_j (1 + \log 1 - \mu_{q,j}^2 - 1) = -\frac{1}{2}(0.5^2) = -0.125$. Negated ELBO $\approx 0.05 + 0.125 = 0.175$: the single numbers that appear in the VAE's practical objective.

Key takeaway — information theory

Everything here averages "how surprised am I", which is $-\log p$. Entropy averages it with the right distribution, cross-entropy with the wrong one, KL is the difference, and mutual information is how much one variable cuts your surprise about another. Jensen's inequality is the observation that a chord lies above a bowl — and it is the single step that produces the ELBO.

A.5 Optimization

Needed in Chapters 4 (reparameterized gradients), 9–11 (SGD+EMA+stop-gradient in contrastive/self-distillation loops), 12 (distillation temperatures), 13 (test-time planning).

In plain words — what training actually is

Training is repeated blind hill-descending. You are standing somewhere on a landscape whose height is the loss and whose horizontal coordinates are the millions of parameters. You cannot see the landscape. All you can do is feel the slope where you stand — that is the gradient — and take a small step downhill. Then feel again, and step again.
Every refinement in this section is a patch on that basic loop. *Stochastic* gradient descent feels the slope using only a handful of examples, because using all of them would be far too slow; the slope you feel is noisy but cheap. *Momentum* remembers the last few steps and keeps rolling in that direction, which smooths the noise out and cuts through narrow valleys. *Adam* keeps a separate step size per coordinate, so a coordinate with consistently tiny gradients still moves. And the three SSL-specific tricks below — EMA weights, stop-gradient, temperature — are not about speed at all: they change *what* the loss encourages, which is why they get their own boxes.

Gradient descent and momentum. SGD updates $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \boldsymbol{g}_t$ from a minibatch gradient. Momentum accumulates $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} + \boldsymbol{g}_t$ and steps $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{m}$; modern optimizers such as Adam add per-coordinate second-moment normalization. Two derived notions recur in SSL:

- **Exponential moving average (EMA) of weights.** $\boldsymbol{\theta}_t \leftarrow \beta\, \boldsymbol{\theta}_t + (1-\beta)\, \boldsymbol{\theta}_s$ with β near 1 (e.g. 0.996). MoCo's key encoder, BYOL's target network, and DINO's teacher are all EMA copies — not trained directly — so their outputs drift slowly and provide stable targets (Ch. 9–11) (He et al., 2020; Caron et al., 2021).
- **Stop-gradient.** $\mathrm{sg}(\boldsymbol{z})$ equals $\boldsymbol{z}$ in the forward pass and contributes zero gradient in the backward pass. Combined with EMA (or with an asymmetric predictor), it is the minimal operation that prevents joint-embedding losses from collapsing to constant outputs (Ch. 10) (Chen and He, 2021).

- **Temperature.** Scaling logits by τ, softmax($\boldsymbol{s}/\tau$), sharpens ($\tau<1$) or flattens ($\tau>1$) the distribution. DINO uses a low teacher temperature (*sharpening*) plus a centered target to keep the teacher ahead of the student (Ch. 11).
- **Optimization with the network frozen.** CRF post-processing and test-time planning both minimize an energy while the network weights stay fixed (pixel-label smoothness for CRFs in Ch. 3; the planning loss $\|\hat{\boldsymbol{z}}_T - \boldsymbol{z}_g\|^2$ for DINO-WM in Ch. 13). What is optimized is the labeling or the *action sequence* — in DINO-WM by sampling-based search (CEM), though gradient descent through the frozen predictor is an alternative — which is why the book says "planning is optimization at test time."

In plain words — EMA and stop-gradient, said once more

An **exponential moving average** of the weights is a lagging copy of the network. Every step it moves a tiny fraction of the way towards the live weights and leaves the rest where it was. With $\beta = 0.996$ it absorbs 0.4% of the current network per step, so it behaves like a smoothed average over roughly the last few hundred steps. Why bother? Because a target that changes as fast as the student is not a target — it is a mirror, and chasing a mirror teaches nothing.

Stop-gradient is a one-line instruction to the autodifferentiator: "use this value, but do not send any blame back through it". Forward, it is invisible. Backward, it is a wall. Putting it on one branch of a two-branch loss means only one branch can move to reduce the loss, which breaks the symmetry that would otherwise let both branches agree on the useless constant answer.

Worked Example — Does EMA alone prevent collapse?

Setting (Ch. 9–10). A Siamese encoder f has weight copy $\bar{\boldsymbol{\theta}}$. Each step: minimize $\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_1) - f_{\bar{\boldsymbol{\theta}}}(\boldsymbol{x}_2)\|^2$ by SGD on $\boldsymbol{\theta}$, then update $\bar{\boldsymbol{\theta}} \leftarrow 0.996\,\bar{\boldsymbol{\theta}} + 0.004\,\boldsymbol{\theta}$. *Trace one collapse step.* Suppose some batch gradient pushes all outputs toward $\mathbf{0}$: before the update, $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = g(\boldsymbol{x})$, and g has $\|g(\boldsymbol{x})\| = 1$. If $\nabla_{\boldsymbol{\theta}}\mathcal{L} = -c\,(g(\boldsymbol{x}_1) - g(\boldsymbol{x}_2))$ points toward greater collapse, one SGD step shrinks outputs by factor $(1-\delta)$. Meanwhile $\bar{\boldsymbol{\theta}}$ attracts back only 0.4% of the gap: after the step, the two encoders' outputs differ by $\sim \delta \cdot 0.996$. Iterating, both collapse together within $O(1/\delta)$ steps. EMA delays but does not stop collapse; Chen and He (2021) show empirically that stop-grad (with an added predictor) is the load-bearing piece — without it the loss runs to -1 and kNN accuracy stays at chance (their Fig. 2/ablation, reproduced in Ch. 10). The lesson: in non-contrastive SSL, read the gradient paths, not just the loss shapes.

Key takeaway — optimization

SGD is "feel the slope on a few examples, step downhill, repeat". Momentum and Adam only change how the step is sized and smoothed. EMA and stop-gradient are different in

kind: they do not speed training up, they change which solutions the loss will accept — which is exactly why they are the load-bearing parts of Chapters 9–11.

A.6 Symbol Index: Everything Not in the Notation Table

The Notation table in the front matter fixes the book's core symbols ($\boldsymbol{x}$, $\boldsymbol{z}$, $\mathcal{L}$, D_{KL}, τ, sg, and the rest). This section catalogues the remaining symbols that appear in the chapters, so that *every* symbol used anywhere in the book is findable in one of the two places. Where a symbol is reused with different meanings, both are listed and the chapter disambiguates.

In plain words — how to use this list

Look up the shape you are staring at, not the concept — that is the point of a symbol index. If the symbol you want is not here, it is in the Notation table at the front. If it is in neither, it is defined on the page where it appears and used nowhere else.

Greek letters.

Symbol	Meaning (and where)
α	interpolation weight (latent interpolation, Ch. 4, 6); focal-loss class weight α_t (Ch. 14); mean-term weight in the ReLMM exploration score (Ch. 14)
β	momentum / EMA coefficient in the optimizer sense (Ch. 2, this appendix); BatchNorm learned shift β_j (Ch. 4); uncertainty-term weight in ReLMM (Ch. 14)
γ	BatchNorm learned scale γ_j (Ch. 4); focal-loss focusing exponent (Ch. 14)
δ_i	annotation indicator: 1 if segment i has a label, else 0 (WVN, Ch. 14)
ϵ, ε	numerical stabilizer inside a square root (Ch. 4, 5); triplet-loss margin (Ch. 9); DBSCAN neighbourhood radius (Ch. 2–3)
$\boldsymbol{\epsilon}$, $\boldsymbol{\varepsilon}$	a noise vector, usually $\mathcal{N}(\mathbf{0}, I)$: the reparameterization draw (Ch. 4), the diffusion noise target (Ch. 7), the Hutchinson probe vector (Ch. 7)
η	learning rate (Ch. 2, this appendix); Gaussian observation variance in a decoder (Ch. 4, 7)
θ (angle)	the angle between two embeddings, so sim $= \cos\theta$ (Ch. 2, 9). Distinct from $\boldsymbol{\theta}$, the parameters
$\boldsymbol{\theta}_s, \boldsymbol{\theta}_t$	student and teacher (EMA) parameters (Ch. 9–12)
$\boldsymbol{\Lambda}$	diagonal matrix of eigenvalues in PCA (Ch. 2–3)
λ	a loss-balancing weight: Barlow Twins' off-diagonal weight (Ch. 10), VICReg's term weights (Ch. 10), CycleGAN's cycle weight (Ch. 12); also the i-th eigenvalue λ_i (Ch. 2–3)
$\boldsymbol{\mu}$, μ_j	a mean vector, or its j-th coordinate: Gaussian mean, encoder output, cluster centroid, BatchNorm batch mean
σ, σ_j	a standard deviation (encoder output, BatchNorm batch statistic); $\sigma(\cdot)$ also denotes the ensemble standard deviation in ReLMM (Ch. 14)
$\boldsymbol{\Sigma}$	a covariance matrix (Ch. 4, this appendix). Note the clash with Σ, the diagonal singular-value matrix of an SVD
τ_s, τ_t	student and teacher softmax temperatures in DINO (Ch. 11)
φ	a generic convex or concave function, used only in Jensen's inequality. Distinct from $\boldsymbol{\phi}$, the encoder parameters
ψ_u, ψ_p	unary and pairwise potentials of a CRF energy (Ch. 3)

Latin letters and words.

Symbol	Meaning (and where)
$\boldsymbol{a}$, $\boldsymbol{a}_{0:t}$	an action, and an action sequence, in world models (Ch. 13)
c	a class index (Ch. 2); DINO's centering vector (Ch. 11); a cluster assignment c_i (Ch. 2–3)
$\mathcal{C}_{ij}$	the cross-correlation matrix between two views' embeddings (Barlow Twins, Ch. 10)
D, E	decoder and encoder as deterministic maps (Ch. 4–5); D is also the GAN discriminator (Ch. 6) — context separates them
d, d_z	input dimension and latent dimension
$d_{n,i}$	distance from sample i to its nearest in-batch neighbour (KoLeo, Ch. 11)
G	the GAN generator (Ch. 6); also the grasp-success predictor G^i in ReLMM (Ch. 14)
g	a projection head or flow map, depending on chapter; as $g_{\boldsymbol{\theta}_t}$ it is DINO's teacher head (Ch. 11)
H, W, C	image height, width, channels
h	KDE bandwidth (Ch. 6). Distinct from $\boldsymbol{h}$, a pre-projection representation
$J_g(\boldsymbol{x})$	the Jacobian matrix of g at $\boldsymbol{x}$ (Ch. 6–7, this appendix)
K	number of classes, clusters, prototypes, or negatives — the chapter says which
k	the number of neighbours in kNN (Ch. 2–3); the number of retained components in PCA; a key index
$\mathcal{M}$	the set of masked patch indices (Ch. 5, 11); also a memory bank or replay buffer (Ch. 8–9, 14)
m	the EMA / momentum coefficient for a teacher or key encoder (Ch. 9–11). Written as a plain scalar, never bold
P	patch side length in a ViT (Ch. 5, 11); also the permutation matrix of an LU factorization (Ch. 6)
p_s, p_t	student and teacher output distributions (Ch. 11)
$\boldsymbol{q}$, $\boldsymbol{k}$	query and key vectors in MoCo's dictionary view (Ch. 9)
s	a logit vector before softmax (Ch. 2, 9, 11); the log-scale output of a coupling layer $s(\cdot)$ (Ch. 6)
T	a time horizon or number of flow layers; $\mathcal{T}$ is the augmentation family and $t \sim \mathcal{T}$ one sampled augmentation
t	diffusion / flow time, $t \in [0,1]$ or $t \in \{1,\dots,T\}$ (Ch. 7); also a sampled augmentation (Ch. 8–9) and a coupling layer's shift $t(\cdot)$ (Ch. 6)
$\boldsymbol{v}$	the velocity field target $\boldsymbol{z} - \boldsymbol{x}$ in flow matching (Ch. 7); also a generic vector in this appendix
w	the classifier-free guidance weight (Ch. 7); $\boldsymbol{w}$ is StyleGAN's style vector (Ch. 6)
$\boldsymbol{x}^+, \boldsymbol{x}^-$	a positive and a negative sample relative to an anchor (Ch. 9)
$\boldsymbol{x}_t$	the noisy / interpolated sample at time t (Ch. 7). Distinct from $\boldsymbol{x}_t$ meaning the observation at timestep t in Ch. 13
$\hat{\boldsymbol{x}}$, $\bar{\boldsymbol{x}}$, $\tilde{\boldsymbol{x}}$	a prediction, an average, and a corrupted version of $\boldsymbol{x}$
$\boldsymbol{z}_g$	the goal latent a planner steers towards (Ch. 13)

Operators and other marks.

Symbol	Meaning (and where)
$\nabla_{\boldsymbol{x}}$	gradient with respect to $\boldsymbol{x}$; $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$ is the score function (Ch. 7)
$\partial g / \partial \boldsymbol{x}$	the matrix of partial derivatives, i.e. the Jacobian (Ch. 6–7)
$\det A$, $\lvert \det A \rvert$	determinant and its absolute value: the signed and unsigned volume-scaling factor
$\operatorname{tr}(A)$	trace: the sum of the diagonal entries (Ch. 7)
$\lVert \cdot \rVert_2$, $\lVert \cdot \rVert_2^2$	Euclidean norm and its square
$\odot$	element-wise product
$\circ$	function composition, as in $f_T \circ \cdots \circ f_1$ (Ch. 6)
$\propto$	"proportional to": equal up to a factor that does not depend on the variable in question
$\sim$	"is distributed as" / "is sampled from"
$\mathbb{1}[\cdot]$	indicator: 1 when the condition holds, else 0
$x_{<t}$, $\boldsymbol{x}_{0:t}$	everything before timestep t; the sequence from 0 to t inclusive (Ch. 7, 13)
$\overset{\text{i.i.d.}}{\sim}$	independent and identically distributed draws
$[n]$	the set $\{1, \ldots, n\}$
$\supseteq$	set containment, used for amodal versus visible masks (Ch. 14)
f^{-1}, $g = f^{-1}$	the inverse map of a bijection (Ch. 6)

Key takeaway — the appendix in one line

Every symbol in the book is either in the front-matter Notation table or in Section A.6; every mathematical object the chapters lean on has a plain-words box above. If a chapter still loses you, the missing piece is almost always one of five things: an expectation, a gradient, a determinant, a KL, or Bayes' rule.

CHAPTER B

Glossary

Entries give a one-line plain meaning, then a precise definition, then the main chapter(s). Related terms are separated by semicolons. Chapter numbers refer to this book; methods with a capitalized name are in the method comparison table (Appendix C) and the paper index.

Action-conditioning — predicting the next observation given the past *and* an action, so the model can answer "what if". Formally, a model $p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_{0:t}, \boldsymbol{a}_{0:t})$ where the action sequence $\boldsymbol{a}$ is an input; used by world models for planning. See Ch. 13. Related: world model; autoregressive model; planning loss.

Adaptive Instance Normalization (AdaIN) — normalizing a feature map and then shifting and scaling it by a style vector. Formally, $\text{AdaIN}(\boldsymbol{x}, \boldsymbol{y}) = \sigma(\boldsymbol{y}) \frac{\boldsymbol{x} - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})} + \mu(\boldsymbol{y})$ with $\sigma(\boldsymbol{y}), \mu(\boldsymbol{y})$ computed per channel from the style; the conditioning mechanism of StyleGAN's synthesis network. See Ch. 6. Related: style vector; mapping network; batch normalization.

Affine coupling layer — an invertible layer that updates half the dimensions by scaling and shifting them with a function of the other half. Formally, $\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}$ and $\boldsymbol{y}_{d:+1:D} = \boldsymbol{x}_{d:+1:D} \odot \exp(s(\boldsymbol{x}_{1:d})) + t(\boldsymbol{x}_{1:d})$, whose Jacobian is triangular with $\log|\det J| = \sum_i s_i(\boldsymbol{x}_{1:d})$; the building block of Real NVP and Glow. See Ch. 6. Related: normalizing flow; change of variables; triangular Jacobian.

Amodal mask — a mask that estimates the full extent of an object, including the parts hidden from view. Formally, an output mask $M_a \supseteq M_v$ that includes occluded regions beyond the visible mask M_v; used in MM-DistillNet to push the audio student to predict objects even off-camera. See Ch. 14. Related: cross-modal distillation; focal loss.

Anchor — the reference sample in a metric-learning tuple against which others are pulled or pushed. Formally, in a triplet $(\boldsymbol{x}, \boldsymbol{x}^+, \boldsymbol{x}^-)$, the anchor is $\boldsymbol{x}$, and the loss reduces $d(f(\boldsymbol{x}), f(\boldsymbol{x}^+))$ while increasing $d(f(\boldsymbol{x}), f(\boldsymbol{x}^-))$. See Ch. 9. Related: positive pair; negative sample; triplet loss.

Approximate nearest neighbor (ANN) search — finding a point that is near the query, possibly not the closest, but much faster. Formally, an algorithm returning $\hat{\boldsymbol{x}}$ with recall/effect near exact search while trading accuracy for sublinear query time (KD-trees in

low dimensions, LSH, Faiss/HNSW-style indexes in high dimensions). See Ch. 2. Related: KD-tree; LSH; recall.

Approximate density model — a generative model that gives only a bound or estimate of $\log p(\boldsymbol{x})$. Formally, models like the VAE that optimize a lower bound (the ELBO) on the log-likelihood instead of the exact value; part of the generative-model taxonomy. See Ch. 6–7. Related: explicit density; ELBO; tractable density model.

Autoregressive model — a generative model that factorizes a sequence into a product of next-step conditionals. Formally, $p(\boldsymbol{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1})$ by the chain rule of probability; applies to pixels (PixelRNN) and text (GPT). See Ch. 7, 11. Related: chain rule; maximum likelihood; world model.

Bandwidth — the smoothing width of a kernel density estimate. Formally, the scale h in $\hat{f}_h(\boldsymbol{x}) = \frac{1}{nh} \sum_i K\big((\boldsymbol{x} - \boldsymbol{x}^{(i)})/h\big)$; small h overfits, large h over-smooths. See Ch. 6. Related: kernel density estimation; kernel.

Batch normalization (BatchNorm) — standardizing each feature of a layer to zero mean and unit variance using the current minibatch statistics, then re-scaling with learned gains. Formally, $\hat{x}_j = (x_j - \mu_{\mathcal{B},j})/\sqrt{\sigma^2_{\mathcal{B},j} + \epsilon}$ followed by $\gamma_j \hat{x}_j + \beta_j$; mitigates vanishing gradients in deep autoencoders. See Ch. 4. Related: internal covariate shift (name only); EMA.

Bidirectional encoding — letting a representation of a token look both left and right in the sequence. Formally, each layer attends over all positions (as in BERT), unlike causal attention which only attends to earlier positions. See Ch. 11. Related: MLM; transformer; GPT.

Bottleneck — a narrow layer that forces information to be compressed. Formally, an intermediate representation $\boldsymbol{z} \in \mathbb{R}^{d_z}$ with $d_z \ll d$ placed between encoder and decoder; the information bottleneck view formalizes its role via mutual information. See Ch. 4. Related: encoder; information bottleneck; codebook.

[CLS] token — a special prepended token whose output embedding is used as a sequence-level summary. Formally, an extra learnable token appended to the input of a transformer whose final-layer output feeds downstream heads (classification in BERT, prototype matching in DINO). See Ch. 11. Related: BERT; DINO; [MASK] token.

Centering — subtracting a running mean from the teacher's outputs before the softmax in DINO. Formally, the teacher logit vector is shifted by a center c updated as $c \leftarrow mc + (1-m)\frac{1}{B}\sum_i g_{\boldsymbol{\theta}_t}(\boldsymbol{x}_i)$; prevents one prototype dimension from dominating. Ch. 11. Related: sharpening; EMA teacher; prototype.

Change of variables formula — the rule that gives the density of a transformed random variable. Formally, for invertible g with inverse f, $\log p_X(\boldsymbol{x}) = \log p_Z(g(\boldsymbol{x})) + \log |\det J_g(\boldsymbol{x})|$; the backbone of normalizing flows. See Ch. 6, 7. Related: Jacobian; normalizing flow; instantaneous change of variables.

Chain rule of probability — the identity that decomposes a joint distribution into conditionals. Formally, $p(x_1, \ldots, x_T) = \prod_t p(x_t \mid x_{<t})$; justifies autoregressive factorization. See Ch. 7. Related: autoregressive model; maximum likelihood.

Classifier-free guidance (CFG) — trading diversity for prompt adherence by extrapolating between a conditional and an unconditional model. Formally, sample with $\boldsymbol{v}^{\text{cfg}} = (1 + w)\, \boldsymbol{v}^{y} - w\, \boldsymbol{v}^{\emptyset}$ where both velocities come from the *same* network trained with dropped conditions; doubles sampling cost. See Ch. 7. Related: conditioning; velocity field; guidance weight.

Clustering — grouping samples by similarity without labels. Formally, a partition (or density-based grouping) of the dataset minimizing an intra-cluster criterion; k-means, DBSCAN, OPTICS are canonical. See Ch. 2–3. Related: k-means; DBSCAN; pseudo-labels.

Cluster loss — a loss that turns cluster assignments into training targets. Formally, cross-entropy between a predicted class distribution and cluster-index pseudo-labels, or contrastive objectives over cluster prototypes (SwAV); the lecture's key slide: "the classification of pseudo-labeled clusters is just another contrastive objective." See Ch. 3, 9, 10. Related: DeepCluster; pseudo-labels; SwAV.

Codebook — a lookup table an over-capacity autoencoder can learn instead of a useful representation. Informally: enough capacity lets the AE memorize indices into the dataset rather than compress structure. Formally, a map $\boldsymbol{x}^{(i)} \mapsto i$ stored implicitly in a wide decoder; the failure mode of unlimited-capacity AEs. See Ch. 4. Related: memorization; bottleneck; regularization.

Collapse — the failure mode where the encoder outputs the same vector for every input. Formally, $\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}) = c$ for all $\boldsymbol{x}$, which trivially minimizes similarity-based losses (SimSiam without stop-grad achieves $-\cos = -1$ and kNN accuracy at chance). See Ch. 10. Related: stop-gradient; predictor head; variance regularization.

Common corruption — not a formal term; here a short name for augmentations like blur, noise, and color jitter that simulate realistic image degradations. Used informally in augmentation design discussions. See Appendix C terminology card. Related: augmentation.

Concept — identity preserved across augmentations. When we say "the model should be invariant to t" we mean the semantic identity of $\boldsymbol{x}$ should not change. Formally tied to the invariance assumption $\boldsymbol{f}(t(\boldsymbol{x})) \approx \boldsymbol{f}(\boldsymbol{x})$. See Ch. 8–10. Related: augmentation; invariance; covariance.

Conditional GAN (cGAN) — a GAN whose generator and discriminator both receive side information y. Formally trained with the minimax objective on pairs (G, y) so D judges (x', y) vs $(G(z, y), y)$, often with an L1 reconstruction term (Pix2Pix). Requires paired data, hence only "semi-self-supervised". See Ch. 6, 12. Related: GAN; Pix2Pix; paired data.

Conditional normalizing flow — a flow whose base density and couplings depend on a conditioning input x. Formally, $p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}(\boldsymbol{x}), \boldsymbol{\sigma}^2(\boldsymbol{x}))\, |\det J_g|$ with conditioned affine couplings $\boldsymbol{y}_0 = s(\boldsymbol{z}_1, \boldsymbol{x}) \cdot \boldsymbol{z}_0 + t(\boldsymbol{z}_1, \boldsymbol{x})$ (where $g = f^{-1}$); used for super-resolution. Beware the

notation flip: condition x, data y. See Ch. 6. Related: normalizing flow; affine coupling layer; super-resolution.

Conditioning — giving the model an auxiliary input at training and generation time. Formally, learning $p(\boldsymbol{y} \mid \boldsymbol{x})$ or $\boldsymbol{v}(\boldsymbol{x}_t, y, t)$ rather than the unconditional version; enables control at sample time. See Ch. 6, 7. Related: conditional GAN; classifier-free guidance; conditional normalizing flow.

Contamination of the reality gap — informal course term for the mismatch between synthetic or curated training data and real deployment distributions; motivates system-level SSL (Ch. 14 treats the problem at length). Related: system-level SSL; distribution shift.

Continual learning with replay — keeping a small buffer of past examples (or self-generated supervision) while adapting online. Formally, training on a replay buffer $\mathcal{M}$ mixed with the newest pseudo-labels to avoid forgetting; used by Frey et al.'s continual segmentation pipeline. See Ch. 14. Related: pseudo-labels; experience replay; adapted student.

Contrastive learning — learning representations by making similar things close and dissimilar things far apart. Formally, any objective that pulls positive pairs together and pushes negatives apart (triplet, N-pair, NCE, InfoNCE, clustering); implemented through softmax cross-entropy in InfoNCE/SimCLR. See Ch. 9. Related: InfoNCE; positive pair; projection head.

Cosine similarity — the normalized dot product measuring angle, not length. Formally, $\mathrm{sim}(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^\top \boldsymbol{v} / (\|\boldsymbol{u}\| \, \|\boldsymbol{v}\|) \in [-1, 1]$; the default similarity in contrastive learning and DINO-feature comparison. See Ch. 2, 9. Related: cosine distance; hypersphere; temperature.

Cosine distance — one minus cosine similarity. Formally, $d_{\cos} = 1 - \cos\theta \in [0, 2]$; behaves as a local metric on the unit hypersphere. See Ch. 2. Related: cosine similarity; Euclidean distance.

Cross-correlation matrix — the empirical covariance between two views' embeddings, computed per feature dimension. Formally, $\mathcal{C}_{ij} = \frac{\sum_b z^A_{b,i} z^B_{b,j}}{\sqrt{\sum_b (z^A_{b,i})^2} \sqrt{\sum_b (z^B_{b,j})^2}}$; Barlow Twins drives it toward identity. See Ch. 10. Related: Barlow Twins; redundancy-reduction term; invariance term.

Cross-entropy — the loss of putting probability mass on the wrong class. Formally, $H(p, q) = -\sum_c p_c \log q_c$ between target p and prediction q; appears in softmax classification, InfoNCE, DINO's teacher– student alignment, and pseudo-label training. See Ch. 2, 4, 8–12. Related: softmax; KL divergence; temperature.

Cross-modal consistency — the requirement that predictions from different modalities (e.g. depth, flow, pose) agree. Formally, losses that compare quantities computed along different sensor paths, such as synthesized rigid flow versus predicted optical flow (DF-Net); a source of free supervision. See Ch. 14. Related: cross-task consistency; forward–backward consistency; rigid flow.

Cross-modal distillation — training a student on one modality to imitate teachers on others. Formally, the student receives only audio while teachers run on RGB/thermal/depth, with losses matching detections (focal) and feature-maps (MTA); MM-DistillNet is the lecture's example. See Ch. 14. Related: multimodal distillation; focal loss; MTA loss.

Cross-task consistency — enforcing that two related predictions (depth, flow) derived from the same scene satisfy a known relation. Formally, $\mathcal{L}_{\text{ctc}}$ aligning flow synthesized from depth+pose against independently predicted flow; DF-Net's third loss. See Ch. 14. Related: cross-modal consistency; rigid flow synthesis.

Curse of dimensionality — the empirical fact that distances concentrate and near-neighbor search degrades as dimension grows. Formally, relative contrast $(d_{\max} - d_{\min})/d_{\min}$ shrinks, so KD-trees fall back to near-linear scans (mitigated by PCA or LSH). See Ch. 2. Related: LSH; PCA; hypersphere.

Cycle consistency — translating to another domain and back should restore the input. Formally, $F(G(x)) \approx x$ and $G(F(y)) \approx y$ in CycleGAN's $\mathcal{L}_{\text{cyc}}$; lets us train without paired data. See Ch. 12. Related: CycleGAN; unpaired data; adversarial loss.

Data augmentation — a random transformation applied to training data to encode invariances or create views. Formally, $t \sim \mathcal{T}$ with $t : \mathcal{X} \to \mathcal{X}$; the design axis of contrastive and pretext methods. See Ch. 6, 8, 9. Related: positive pair; augmentation in other modalities; augmentation composition.

Data balancing — re-weighting samples so the empirical distribution is more uniform. Formally, hierarchical k-means partitions the data and sampling caps subtrees to sizes n chosen so $\sum_j \min(n, s_j) \approx N$; a step in the DINOv3 data engine. See Ch. 11. Related: hierarchical clustering; data curation; long tail.

Data curation / data engine — the pipeline that turns raw crawls into training sets without labels. Formally, deduplication, NSFW and near-duplicate filtering, retrieval against seed datasets, and hierarchical rebalancing (DINOv2's LVD pipeline); the lecture's scaling thesis. See Ch. 11. Related: curated vs crawl data; deduplication; retrieval-based curation.

Decoder — the network that maps a latent back to data space. Formally, $p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})$ or a deterministic $D_{\boldsymbol{\theta}}(\boldsymbol{z})$; the generative half of an autoencoder or VAE. See Ch. 4. Related: encoder; latent variable; reconstruction loss.

Deduplication — removing exact or near-duplicate samples from a crawl. Formally, hash-based exact dedup plus embedding-based copy detection; part of the DINOv2/DINOv3 data engines. See Ch. 11. Related: data curation; near-duplicate; data balancing.

Denoising autoencoder (DAE) — an autoencoder trained to recover a clean signal from a corrupted one. Formally, the model receives $\tilde{\boldsymbol{x}} \sim q(\tilde{\boldsymbol{x}} \mid \boldsymbol{x})$ and minimizes $\|D(E(\tilde{\boldsymbol{x}})) - \boldsymbol{x}\|^2$, forcing it to learn robust structure; stacks give a deep layerwise-pretrained network. See Ch. 4, 7. Related: autoencoder; stacked DAE; layerwise pretraining.

Descriptor / descriptor set — a list of feature vectors describing an image, one per patch or region. Formally, $\{a_1, \ldots, a_N\}$ extracted by a backbone; comparison between sets needs a metric. See Ch. 2. Related: embedding; Euclidean distance; cosine similarity.

Diffusion feature (DIFT) — intermediate U-Net activations of a diffusion model, repurposed as a dense descriptor. Formally, given timestep t, extract activations, upsample to pixel resolution, and match by cosine similarity; produces semantic correspondence without supervision. See Ch. 7, 13. Related: diffusion model; correspondence; U-Net.

Diffusion model — a generative model that learns to remove noise step by step. Formally, a predictor $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ trained to denoise $\boldsymbol{x}_t$ and applied iteratively from $\boldsymbol{x}_1 \sim p_{\text{noise}}$; score-based and flow-matching variants are its continuum limits. See Ch. 7. Related: denoising; rectified flow; classifier-free guidance.

Diffusion policy — a robot policy that treats action sequences as a denoising problem. Formally, a conditional diffusion model over future-action chunks given visual input; powerful but supervised-imitation, *not* self-supervised. See Ch. 7. Related: diffusion model; conditioning; world model.

Dimensionality reduction — mapping data to fewer coordinates while keeping structure. Formally, PCA, t-SNE, and the encoders of AEs/VAEs; used for visualization, denoising, and as preprocessing for clustering. See Ch. 2, 4. Related: PCA; t-SNE; encoder.

DINO self-distillation — a student–teacher loop where both networks are the same architecture and the teacher is an EMA of the student. Formally, cross-entropy $-\sum_k p_t(k) \log p_s(k)$ between the teacher's sharpened soft targets and the student's predictions over augmentations; "distillation with no labels". See Ch. 10–12. Related: self-distillation; centering; sharpening.

Discriminator — the network in a GAN that distinguishes real from generated samples. Formally, $D(\boldsymbol{x}) \approx P(\boldsymbol{x} \text{ is real})$, trained against the generator in a minimax game; provides no density. See Ch. 6. Related: generator; minimax; implicit density.

Disentanglement — the degree to which single latent dimensions map to single semantic factors. Formally, linear subspaces of $\mathcal{Z}$ should correspond to interpretable changes in $\mathcal{X}$ (pose vs. hair color); the lecture's "Sims character creator" desideratum. See Ch. 6. Related: latent space; style vector; linear subspace.

Distillation — training a small student to match a teacher's full output distribution. Formally, cross-entropy between student and teacher softmax outputs (optionally temperature-smoothed), possibly at multiple layers; used for compression (Gemini Flash), multi-teacher fusion (DUNE), and self-distillation (DINO). See Ch. 12. Related: soft targets; EMA teacher; DUNE.

Downstream task — the labeled task by which a representation is judged. Formally, any supervised task (classification, detection, segmentation, depth, pose) trained on frozen or fine-tuned features after self-supervised pretraining. See Ch. 3, 5, 8–11. Related: evaluation protocol; linear probing; fine-tuning.

Downstream evaluation — measuring representation quality by its performance on downstream tasks, as opposed to looking at features directly. One of the two top-level branches of the evaluation taxonomy (the other is direct inspection). See Ch. 8, 10–11. Related: downstream task; linear probe; kNN evaluation.

Embedding — the vector a model produces for an input. Formally, $\boldsymbol{z} = f(\boldsymbol{x}) \in \mathcal{Z}$; the object all downstream evaluation reads. Synonym: representation (this book uses both). See Ch. 2–14. Related: representation; feature space; projection head.

Empirical risk minimization (ERM) — choosing parameters by minimizing the average loss on the training set. Formally, $\arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_i \mathcal{L}(f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)})$, justified by the law of large numbers; in SSL the labels come from data augmentations or clusters. See Ch. 8–12. Related: maximum likelihood; pseudo-labels.

Encoder — the network that maps data into the latent space. Formally, $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ in a VAE or a deterministic $E(\boldsymbol{x})$; where representation learning "happens". See Ch. 4–14. Related: decoder; backbone; bottleneck.

Energy function (CRF) — the scalar scoring joint label assignments in a CRF. Formally, $E(\boldsymbol{x}) = \sum_i \psi_u(x_i) + \sum_{i<j} \psi_p(x_i, x_j)$; low energy means coherent labels, and minimizing it refines segmentations. See Ch. 3. Related: unary potential; pairwise potential; pairwise term.

Entropy — the average uncertainty of a distribution. Formally, $H(p) = -\sum_x p(x) \log p(x)$; appears in cross-entropy, mutual information, and the temperature/sharpening trade-off (low entropy = sharp). See Appendix A, Ch. 4, 11. Related: cross-entropy; KL divergence; softmax.

Evaluation protocol — the rules for turning a self-supervised model into a scalar quality. Formally, a fixed recipe: freeze or fine-tune, which head, on which dataset, with which metric; fairness requires the same protocol across methods. See Ch. 3, 5, 8–11. Related: linear probe; kNN evaluation; fine-tuning.

Evidence — the likelihood of the data under the generative model. Formally, $p(\boldsymbol{x}) = \int p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z})\, d\boldsymbol{z}$; the quantity the ELBO lower-bounds. See Ch. 4. Related: ELBO; maximum likelihood; Jensen.

E-step / M-step — the two alternating steps of EM-style algorithms and k-means. Formally, E = assign points to centroids, M = recompute centroids as means of their assigned points; iterated until assignments stop changing. See Ch. 2. Related: k-means; centroid; Lloyd's algorithm.

Expected log-likelihood — the "reconstruction" term appearing inside the ELBO. Formally, $\mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})]$; Gaussian decoders turn it into a weighted L2 error. See Ch. 4. Related: reconstruction loss; ELBO; Gaussian observation model.

Explicit density model — a generative model whose $p(\boldsymbol{x})$ is computable. Formally, tractable (autoregressive, normalizing flows) or approximate (VAE) in the Goodfellow taxonomy. See Ch. 6–7. Related: implicit density; tractable density; normalizing flow.

Exploration (uncertainty-based) — using model uncertainty to pick informative actions instead of random ones. Formally, sampling $\tilde{G}(o, a) = \alpha\, \mathbb{E}[G^i(o, a)] + \beta\, \sigma(G^i(o, a))$

over an ensemble, favoring both likely success and disagreement; used by ReLMM to decide where to grasp. See Ch. 14. Related: ensemble uncertainty; self-reset; curiosity.

False negative — a sample treated as a negative although it is semantically identical to the positive. Formally, a batch element $\boldsymbol{x}_j$ with the same underlying class as $\boldsymbol{x}_i$ that contributes to the denominator of InfoNCE and pushes the model away from its own class; their number grows with batch size (about $(2N-2)/C$ per anchor for C balanced classes), though each one is diluted among more true negatives. See Ch. 9. Related: in-batch negatives; batch size; InfoNCE.

Feature extractor — the shared backbone network that maps raw input to features. Formally, $f_{\boldsymbol{\theta}}$ in Figure/Algorithm diagrams; it is what pretraining produces. See Ch. 6–14. Related: encoder; backbone; representation.

Feature space — the coordinate system in which representations live. Formally, $\mathcal{Z} \subseteq \mathbb{R}^d$; a "good" feature space is one where semantic classes correspond to geometry. See Ch. 2, 6, 8–14. Related: embedding; abstraction; meaningful distance metric.

Feature visualization — qualitative inspection of what a representation encodes, e.g. PCA projections, t-SNE plots, attention maps, or first-layer filters. One half of the direct-inspection side of the evaluation taxonomy. See Ch. 8, 10. Related: direct inspection; attention map; t-SNE.

Fiducial point / anchor in kNN classifier — we use *anchor* for the triplet-loss reference; the nearest-neighbor query point is a separate concept. See anchor; k-nearest neighbors.

Fine-tuning — continuing training the pretrained backbone on a downstream task. Formally, unfreeze the encoder and update *all* parameters (or all but a head) with a small learning rate; the "adaptation" half of pretrain+fine-tune. See Ch. 5, 8–12. Related: linear probing; transfer learning; backbone.

Focal loss — a weighted cross-entropy that downweights easy samples. Formally, $\mathcal{L}_{\text{focal}} = -\alpha_t(1-p_t)^\gamma \log p_t$; emphasizes hard or rare detections; used by MM-DistillNet to align audio predictions with multi-modal teacher detections. See Ch. 14. Related: cross-entropy; cross-model distillation.

Forward–backward consistency — requiring synthesized depth/flow to agree in both directions between two frames. Formally, depth of frame A composed with the relative pose must reproduce depth of frame B and vice versa; a geometric loss used by DF-Net. See Ch. 14. Related: cross-modal consistency; rigid flow synthesis; invalid mask.

Generative adversarial network (GAN) — a generator trained to fool a discriminator. Formally trained with the minimax objective $\min_G \max_D \mathbb{E}[\log D(x)] + \mathbb{E}[\log(1 - D(G(z)))]$; implicit density, high-quality samples, no likelihood. See Ch. 6. Related: discriminator; minimax game; implicit density model.

Generator — the network in a GAN that maps noise to samples. Formally, $G : \mathcal{Z} \to \mathcal{X}$ with $G(\boldsymbol{z}) \sim p_G$; trained against D. See Ch. 6. Related: GAN; generator distribution; sampling.

Generative model — any model of $p(X)$ from which we can sample or estimate likelihoods. Formally, the course treats flows, VAEs, GANs, diffusion, and autoregressive models; generation *is* self-supervision because the data itself provides the targets. See Ch. 6–7. Related: density estimation; taxonomy of generative models; hidden structure.

Guidance — biasing diffusion samples toward a condition beyond what the conditional model does on its own; classifier-free guidance is the course's method of choice. See Ch. 7. Related: classifier-free guidance; conditioning.

Hash table / LSH bucket — the storage structure LSH points land in. Formally, applying an LSH h assigns each point a bucket; colliding points are candidates for close neighbors. See Ch. 2. Related: locality sensitive hashing; approximate nearest neighbor.

Head — a small trainable sub-network attached on top of a frozen backbone. Formally, a linear or low-capacity MLP predicting task labels from features; "linear head" when the head is one affine layer. See Ch. 8–12. Related: linear probing; projection head; predictor.

Hierarchical k-means — clustering the cluster centroids recursively to build a multi-level tree. Formally, at level t run $C_t = \text{kmeans}(C_{t-1}, k_t)$ with $C_0 = X$; used with resampling to balance web-scale corpora (Vo et al. for DINOv3). See Ch. 11. Related: data balancing; data curation; k-means.

Hutchinson trace estimator — replacing the trace of a huge matrix by an expectation over probe vectors. Formally, $\text{tr}(A) = \mathbb{E}_{\boldsymbol{\epsilon}}[\boldsymbol{\epsilon}^\top A \boldsymbol{\epsilon}]$ with centered, unit-variance $\boldsymbol{\epsilon}$; enables tractable log-likelihoods in continuous normalizing flows (Ch. 7; also Ch. 6 background). Related: change of variables; instantaneous change of variables; trace.

iBOT loss — a token-level DINO-style loss for masked image modeling. Formally, $\mathcal{L}_{\text{iBOT}} = -\sum_{i \in \mathcal{M}} p_{t,i} \log p_{s,i}$ over masked patch tokens where $p_{t,i}/p_{s,i}$ are teacher/student distributions over a codebook of visual tokens; used in DINOv2 alongside the class-token DINO loss. See Ch. 5, 11. Related: MLM; DINO; teacher–student.

Image-to-image translation — mapping images from one domain to another (day→night, sketch→photo). Paired: Pix2Pix uses supervised pairs; unpaired: CycleGAN uses cycle consistency; the taxonomy boundary between strictly supervised and semi-self-supervised. See Ch. 6, 12. Related: paired vs unpaired data; cycle consistency; conditional GAN.

Implicit density model — a generative model that can sample but cannot compute $p(\boldsymbol{x})$. Formally, GANs and diffusion under the Goodfellow taxonomy. See Ch. 6. Related: explicit density; GAN; diffusion model.

In-batch negative — every other sample in the minibatch serves as a negative. Formally, SimCLR's InfoNCE sums over the $2N - 2$ other augmented samples in the batch; the reason large batches are needed. See Ch. 9. Related: false negative; batch size; InfoNCE.

Invariance vs. covariance — the design choice whether features should ignore an augmentation (invariant) or track it (covariant). RotNet asks for covariance (discriminate rotation); PIRL asks for invariance (semantic identity); contrastive SSL usually wants invariance to nuisance but covariance to semantics. See Ch. 8. Related: augmentation; pretext task; PIRL.

Information bottleneck (IB) — the principle of keeping only the task-relevant information about $\boldsymbol{x}$ in $\boldsymbol{z}$. Formally, maximize $I(Z;Y)$ while minimizing $I(X;Z)$; an autoencoder's bottleneck is its heuristic instantiation. See Ch. 4. Related: mutual information; bottleneck; encoder.

Information gain / mutual information — see mutual information. In the course this is the object contrastive losses are argued to maximize a lower bound of (SimCLR slide; InfoMin analysis in Ch. 15). See Ch. 9, 15. Related: InfoNCE; data-processing inequality.

Inpainting — predicting the content of a masked image region directly in pixel space. Formally, pretext where part of $\boldsymbol{x}$ is removed and the network outputs the missing region; Context Encoders are the classic instance. See Ch. 8. Related: context encoder; MAE; masking.

Interpolation (latent) — moving between two data samples along a line in latent space to check smoothness. Formally, decode $\alpha \boldsymbol{z}_1 + (1-\alpha)\boldsymbol{z}_2$ for $\alpha \in [0,1]$; VAE/flow/GAN papers use it to demonstrate that the latent space is semantically continuous. See Ch. 4, 6. Related: latent space; VAE; disentanglement.

Invalid mask — a binary mask marking pixels that violate static scene assumptions (moving objects, occlusions). Formally, pixels where rigid-flow warping from frame A disagrees with frame B are excluded from photometric losses; geometric SSL needs such masks. See Ch. 14. Related: rigid flow; cross-modal consistency; forward–backward consistency.

Invertible layer — a layer whose input can be reconstructed exactly from its output. Formally, architectures chosen so the inverse map $g = f^{-1}$ exists with tractable cost (coupling, 1×1 conv / LU-parameterized); the constraint of normalizing flows. See Ch. 6. Related: normalizing flow; affine coupling; log-determinant.

Jacobian — the matrix of first derivatives of a vector-valued map. Formally, $J_g(\boldsymbol{x})_{ij} = \partial g_i / \partial x_j$; its determinant is the local volume distortion needed by the change of variables formula; triangular Jacobians have product-of-diagonal determinants. See Ch. 6, 7. Related: change of variables; invertible layer; affine coupling.

Jensen's inequality — the expectation of a concave function is less than or equal to the function evaluated at the expectation. Formally, for concave h, $\mathbb{E}[h(X)] \le h(\mathbb{E}[X])$; used once to derive the ELBO. See Ch. 4, Appendix A. Related: ELBO; cross-entropy.

k-means — clustering by alternating assignment and mean recomputation. Formally, the E/M iteration that minimizes $\sum_i \|\boldsymbol{x} - \boldsymbol{\mu}_{c_i}\|^2$; k-means++ seeding improves the worst-case bound. See Ch. 2–3, 11. Related: centroid; k-means++; mini-batch k-means.

k-means++ — a seeding scheme for k-means that picks new centroids proportionally to distance squared. Formally, D^2-sampling guarantees expected objective at most $8(\ln k + 2)$ from optimal; the course shows it produces better-spread centers. See Ch. 2. Related: k-means; seeding.

k-nearest neighbors (kNN) — predicting a label by taking a majority vote over the closest k labeled points. Formally, $\hat{y} = \text{mode}(y_{(1)}, \ldots, y_{(k)})$ for neighbors in some metric; an instant evaluation of feature quality. See Ch. 2–3, 8–11. Related: nearest neighbor; distance metric; evaluation protocol.

KD-tree — a binary tree that recursively splits space along axis-aligned planes to answer nearest-neighbor queries quickly. Formally, nodes store splitting values; queries traverse, then unwind, pruning half spaces whose distance to the query exceeds the current best; degrades in high dimension. See Ch. 2. Related: ANN search; nearest neighbor.

Kernel — a similarity function someone places around each point in KDE. Formally, any non-negative K with $\int K = 1$ (Gaussian, tophat, Epanechnikov, ...); choice matters less than bandwidth. See Ch. 6. Related: kernel density estimation; bandwidth.

Kernel density estimation (KDE) — estimating a continuous density by summing kernels around each observation. Formally, $\hat{f}_h(\boldsymbol{x}) = \frac{1}{nh} \sum_{i=1}^{n} K((\boldsymbol{x} - \boldsymbol{x}^{(i)})/h)$; the slide's sloth/rice-rat example, then a simple generative model for digits. See Ch. 6. Related: bandwidth; kernel; generative model.

Key (dictionary) — the target feature a query is matched to in contrastive-as-retrieval (MoCo's frames). Formally, the representation of an augmented view encoded with a momentum encoder and stored in a queue; contrastive learning becomes a dictionary look-up. See Ch. 9. Related: query; momentum encoder; queue.

KL divergence — the expected excess number of nats from coding p with a code optimized for q. Formally, $D_{\text{KL}}(p \,\|\, q) = \mathbb{E}_p[\log(p/q)] \geq 0$ with equality iff $p = q$ a.e.; asymmetric and central to VAEs and self-distillation. See Appendix A, Ch. 4, 7, 11. Related: cross-entropy; entropy; ELBO.

kNN evaluation — scoring a representation by running kNN classification on frozen features without further training. Formally, fit the kNN on train features and report accuracy on the validation set; DINO gets 74.5% top-1 with a ViT-S/16. See Ch. 8, 10–11. Related: evaluation protocol; linear probing; feature extractor.

KoLeo regularizer — a Kozachenko–Leonenko differential-entropy estimator that pushes embeddings to spread out. Formally, $\mathcal{L}_{\text{KoLeo}} = -\frac{1}{n} \sum_{i=1}^{n} \log d_{n,i}$ with $d_{n,i}$ the distance of the i-th sample to its nearest neighbor in the batch; used in DINOv2 against feature collapse. See Ch. 11. Related: collapse; uniformity; differential entropy.

Latent diffusion — running diffusion in the latent space of a pretrained autoencoder instead of pixel space. Formally, first train a VAE (with a small KL weight and often an adversarial term) to map $\boldsymbol{x}$ to a spatial latent $\boldsymbol{z}$ of size $H/D \times W/D \times C$, then train diffusion

on $\boldsymbol{z}$; the composition used by stable diffusion. See Ch. 7. Related: VAE; latent space; diffusion model.

Latent space / latent variable — the internal coordinate system $\mathcal{Z}$ in which a model stores compressed information, and the corresponding random variable $\boldsymbol{z}$ in probabilistic models. Formally, $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ for encoders or $\boldsymbol{z} \sim p(\boldsymbol{z})$ for priors; its geometry is what contrastive/flow/GAN methods shape. See Ch. 4, 6–13. Related: embedding; representation; sampling.

Layerwise pretraining — training one shallow network at a time and freezing it before adding the next. Formally, stack denoising autoencoders layer by layer where hidden layers are trained on $\|\boldsymbol{z} - \hat{\boldsymbol{z}}\|^2$; historically needed before BatchNorm/ReLU made deep training feasible. See Ch. 4. Related: denoising autoencoder; stacked AE.

Linear probing — classifying with a linear layer on frozen features to test representation quality. Formally, train only $W \in \mathbb{R}^{K \times d}$ on $\boldsymbol{z} = f(\boldsymbol{x})$ while $\boldsymbol{\theta}$ is fixed; the most common evaluation, sensitive to whether the pretraining left information linearly readable. See Ch. 3, 5, 8–11. Related: head; kNN evaluation; fine-tuning.

Locality sensitive hashing (LSH) — a hash family where close points collide with higher probability than far ones. Formally, h is (r, c, p_1, p_2)-sensitive if $d(a, b) \le r \Rightarrow p[h(a) = h(b)] \ge p_1$ and $d(a, b) \ge cr \Rightarrow p[h(a) = h(b)] \le p_2$; combine L hash tables to approach exact search. See Ch. 2. Related: hash table; approximate nearest neighbor; curse of dimensionality.

Log-determinant — the scalar $\log|\det J|$ that appears in the change-of-variables log-likelihood. Formally, accumulated as a sum across flow layers when Jacobians are triangular or parameterized with LU; must remain finite and cheap to compute for flows to be practical. See Ch. 6. Related: Jacobian; change of variables; affine coupling.

Log trick — replacing a product of probabilities by a sum of log-probabilities for stability. Formally, $\arg\max_W \prod_i p(\boldsymbol{x}^{(i)}) = \arg\max_W \sum_i \log p(\boldsymbol{x}^{(i)})$; converts maximum likelihood into the sums used in autoregressive training. See Ch. 7. Related: maximum likelihood; chain rule.

Loss ladder — the course's list of contrastive losses from triplet to clustering: Triplet, N-pair, NCE, InfoNCE, Cluster loss. Informal organizing device; each entry adds negatives or structure. See Ch. 9. Related: triplet loss; N-pair loss; InfoNCE; cluster loss.

Mapping network — the first stage of StyleGAN that turns the noise code into a disentangled style. Formally, a stack of 8 fully connected layers mapping $\boldsymbol{z} \in \mathcal{Z}$ to $\boldsymbol{w} \in \mathcal{W}$, whose output is broadcast into per-layer AdaIN styles. See Ch. 6. Related: style vector; AdaIN; synthesis network.

Margin — the minimum gap between positive and negative distances in ranking losses. Formally, the ε in $\max(0, \|f(\boldsymbol{x}) - f(\boldsymbol{x}^+)\|^2 - \|f(\boldsymbol{x}) - f(\boldsymbol{x}^-)\|^2 + \varepsilon)$; without it ($\varepsilon = 0$) collapsing every embedding to one point gives zero loss; the margin forbids that by demanding a real gap between positive and negative distances. See Ch. 9. Related: triplet loss; contrastive learning.

Mask token — a learned placeholder inserted where data was hidden. Formally, a shared, trainable embedding placed at masked positions so that the decoder still knows *where* to

produce reconstructions (MAE; BERT's [MASK]). See Ch. 5, 11. Related: masking ratio; visible patch; positional embedding.

Masked autoencoder (MAE) — an asymmetric autoencoder that encodes only the visible 25% of patches and decodes the full reconstruction. Formally, a ViT encoder on visible patches, mask tokens inserted at positions, a shallow decoder, pixel MSE loss on masked positions; high masking ratios are best for linear probing. See Ch. 5. Related: masked patch; asymmetric encoder; visible patch.

Masked language modeling (MLM) — predicting randomly hidden tokens from bidirectional context. Formally, BERT's objective: mask 15% of tokens, predict them, loss is token-level cross-entropy; the NLP ancestor of masked image modeling. See Ch. 11. Related: [MASK] token; bidirectional encoding; BERT.

Maximum likelihood estimation (MLE) — fitting parameters by maximizing the probability of the observed data. Formally, $\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_i p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$; the implicit objective behind cross-entropy everywhere in the book. See Ch. 7, Appendix A. Related: log trick; evidence; cross-entropy.

Meaningful distance metric — the informal property that nearby feature vectors correspond to semantically similar inputs. Formally the goal of representation learning in Ch. 8; the test is whether distance in $\mathcal{Z}$ correlates with task-relevant distance. Related: feature space; embedding; abstraction.

Memorization — learning to output training samples rather than structure. Formally, overfitting in a generative or autoencoder setting so $q(\boldsymbol{x})$ concentrates on the empirical data points; guarded against by bottlenecks, noise, or larger datasets. See Ch. 4, 6. Related: codebook; overfitting; bandwidth.

Memory bank — a rolling store of previously computed features used as negatives. Formally, a cache of representations $\mathcal{M}$ updated each step; PIRL's way to get large K without large batches. See Ch. 8–9. Related: queue; momentum encoder; in-batch negative.

Mini-batch k-means — an online variant of k-means that uses small random subsets per step. Formally, pick a random batch, assign, update counts with a running average; scalable to web-scale data. See Ch. 2. Related: k-means; online clustering.

Minimax game — the adversarial optimization between generator and discriminator. Formally, $\min_G \max_D V(D, G)$ with $V = \mathbb{E}[\log D(x)] + \mathbb{E}[\log(1 - D(G(z)))]$; under idealized capacity its Nash equilibrium is $p_G = p_{\text{data}}$. See Ch. 6. Related: GAN; discriminator; generator.

MTA loss (multi-teacher alignment) — distillation that matches the student's channel-averaged feature-map attention to the teachers'. Formally, a KL divergence between ℓ_2-normalized spatial attention maps of the student and of the combined teachers (the product of the RGB, thermal and depth teachers' maps), at several feature-pyramid levels; complements the detection focal loss in MM-DistillNet. See Ch. 14. Related: cross-modal distillation; focal loss; attention map.

Multi-crop — augmenting each image with two global and several local crops so the model aligns local-to-global predictions. Formally, a set of low-resolution crops (96×96 in DINO, 98×98 in DINOv2) + two 224×224 crops whose representations are matched by cross-entropy; the detail that makes DINO efficient and effective. See Ch. 11. Related: global crop; local crop; self-distillation; student–teacher.

Multimodal distillation — a general term for a student on one modality learning from teachers on another; concretely MM-DistillNet (audio from RGB/thermal/depth) is the running example. See Ch. 12, 14. Related: cross-modal distillation; MTA loss; distillation.

Mutual information (MI) — the reduction in uncertainty about one variable from observing another. Formally $I(X;Z) = D_{\mathrm{KL}}(p(x,z) \,\|\, p(x)p(z))$; InfoNCE maximizes a lower bound on $I(\boldsymbol{f}(\boldsymbol{x}); \boldsymbol{f}(\boldsymbol{x}^+))$. See Ch. 4, 9, 15. Related: InfoNCE; variational bound; entropy.

N-pair loss — a generalization of triplet loss to multiple negatives per pair. Formally, $\log(1 + \sum_{i=1}^{N-1} \exp(f(\boldsymbol{x})^\top f(\boldsymbol{x}_i^-) - f(\boldsymbol{x})^\top f(\boldsymbol{x}^+)))$; every step repels $N-1$ negatives at once. See Ch. 9. Related: triplet loss; InfoNCE; margin.

Natural basis for autoencoders — not a glossary-worthy term on its own; see bottleneck; codebook; representation.

Negative example / negative sample — any sample whose representation should be far from the anchor's. Formally, the terms in the denominator of InfoNCE or the $\boldsymbol{x}^-$ of a triplet; their number/quality controls the difficulty and the false-negative risk. See Ch. 9. Related: anchor; in-batch negative; false negative.

Noise-contrastive estimation (NCE) — training by discriminating data samples from a noise distribution. Formally, a logistic regression between real and noise samples gives a tractable estimator for unnormalized models; in the course re-appears as PIRL's loss. See Ch. 8–9. Related: InfoNCE; negative example.

Noisy Student — a self-training pipeline where the student is trained with added noise (augmentations, dropout, stochastic depth) on pseudo-labels from a teacher, then becomes the next teacher. Iterated, it surpasses the original SoTA. See Ch. 12. Related: self-training; pseudo-labels; EMA teacher-free distillation.

Normalizing flow — an invertible generative model obtained by composing simple bijections. Formally, $\boldsymbol{x} = f_T \circ \cdots \circ f_1(\boldsymbol{z})$, $\log p_X(\boldsymbol{x}) = \log p_Z(g(\boldsymbol{x})) + \sum_t \log |\det J_{g_t}|$; requires invertibility and tractable Jacobians. See Ch. 6, 7. Related: change of variables; affine coupling; invertible layer.

NSP (next sentence prediction) — BERT's second pretext task, predicting whether sentence B follows sentence A. Formally, binary classification from the [CLS] output; largely deprecated by later work but historical context for bidirectional pretraining. See Ch. 11. Related: MLM; [CLS] token.

OPTICS — a density-based clustering algorithm related to DBSCAN that outputs an ordering of points instead of a flat partition. Formally, produces a reachability plot whose

valleys are clusters and works with less memory than building the full $N \times N$ affinity matrix. See Ch. 2. Related: DBSCAN; clustering.

Paired vs. unpaired data — the distinction between having matched input–output examples (sketch+photo) and having two unrelated collections. Paired data crosses into supervised/semi-supervised territory; unpaired needs cycle consistency or other unsupervised bridges. The course treats collecting pairs "for free" (super-resolution, augmentations) as still SSL. See Ch. 12. Related: CycleGAN; Pix2Pix; conditional GAN.

Patch — a small fixed spatial slice of an image, the atom of ViT tokenization. Formally, an image $\mathbb{R}^{H\times W\times C}$ is split into $P \times P$ patches that are linearly embedded and treated as tokens by the transformer; MAE masks individual patches. See Ch. 5, 11. Related: visible patch; mask token; ViT.

PCA (principal component analysis) — orthogonal projection onto the top-variance directions of the data. Formally, eigen-decompose the covariance matrix and keep the top k eigenvectors; used before clustering and for feature visualization. See Ch. 2–3, 8, 11. Related: dimensionality reduction; embedding; t-SNE.

Perplexity — exp of average negative log-likelihood, the standard autoregressive quality metric. Mentioned only in the autoregressive interlude (Ch. 7); lower is better. Related: autoregressive model; maximum likelihood.

Planning loss / test-time planning — optimizing a sequence of actions so that predicted future latents match a goal latent. Formally, roll out $\hat{\boldsymbol{z}}_T$ from the world model given candidate actions and minimize $\|\hat{\boldsymbol{z}}_T - \boldsymbol{z}_g\|^2$ over the actions; DINO-WM plans in a frozen DINOv2 feature space. See Ch. 13. Related: world model; latent space; DINO-WM.

Pose estimation — predicting the relative motion between two frames, usually rotation+translation. In DF-Net, one of the three coupled networks whose output synthesizes rigid flow; in WVN, robot proprioception gives the pose used to reproject path segments into the image. See Ch. 14. Related: cross-modal consistency; rigid flow; proprioception.

Positive pair — two views of the same underlying sample that are trained to be similar. Formally, $(\boldsymbol{x}, \boldsymbol{x}^+)$ with $\boldsymbol{x}^+ = t(\boldsymbol{x})$ for $t \sim \mathcal{T}$ (or the same video frame/text span); the anchor–positive distance is minimized. See Ch. 8–12. Related: augmentation; anchor; contrastive learning.

Pre-registration / protocol — item on the researcher's checklist: fix the evaluation protocol before training; see Appendix C. Related: evaluation protocol.

Pretext-invariant representation learning (PIRL) — contrastive learning where positives are the same image under a destructive geometric pretext transform. Formally, the model is trained to align $f(I)$ with $f(I^t)$ against a memory bank of negatives; contrast with RotNet's covariance. See Ch. 8. Related: invariance; memory bank; NCE.

Pretext task — a hand-made classification/prediction problem whose solution requires understanding the content. Formally, given $\boldsymbol{x}' = t(\boldsymbol{x})$, predict t or some property of t (rotation,

jigsaw permutation, relative patch position, color channels); the label is computed from the data itself. See Ch. 8. Related: pretext vs downstream; rotation prediction; jigsaw.

Principal components — the orthogonal eigenvectors of the data covariance, ordered by explained variance. See PCA. Related: PCA.

Projection head — a small MLP appended during training whose output is used for the contrastive loss, then thrown away. Formally, SimCLR learns $\boldsymbol{z} = g(\boldsymbol{h})$, applies InfoNCE to $\boldsymbol{z}$, and evaluates $\boldsymbol{h}$ with the head discarded; the information-preserving story is that the head absorbs the augmentation-invariance pressure. See Ch. 9. Related: representation; InfoNCE; SimCLR.

Proprioception — a robot's internal sensing of its own motion (odometry, joint states). In WVN it supplies measured traversability; in ReLMM it enables self-reset. See Ch. 14. Related: sensory policy; system-level SSL.

Prototype — the class-like vector a DINO-style head assigns its output to. Formally, one of the K columns of the last projection layer whose softmax score defines the sharpened teacher target; also a single k-means centroid in clustering-based methods. See Ch. 9–11. Related: center; cluster; SwAV.

Pseudo-label — a target generated by the model itself or by an aggregator, not by a human. Formally, any label computed without external annotation (argmax of teacher, cluster id, rendered segmentation from a 3D map); concise way to reuse supervised losses in SSL. See Ch. 3, 9, 12, 14. Related: self-training; cluster loss; system-level pseudo-labels.

Query (dictionary) — the current sample whose representation must match its key in MoCo's view of contrastive learning. Formally, the representation from the online encoder, matched against queued momentum encoder keys; selecting the right key out of a queue is a cross-entropy. See Ch. 9. Related: key; queue; momentum encoder.

Queue — a FIFO cache of past keys in MoCo. Formally, decouples the number of negatives from batch size; enqueue current minibatch keys, dequeue oldest. See Ch. 9. Related: memory bank; momentum encoder; in-batch negative.

Reconstruction loss — penalizing the difference between input and decoder output. Formally, $\mathcal{L}_{\text{rec}} = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|_2^2$, which equals Gaussian negative log-likelihood up to a constant; the backbone of AEs, VAEs, DAEs, MAE. See Ch. 4–5, 7. Related: reconstruction target; expected log-likelihood; MAE.

Reconstruction target — what the decoder is asked to predict: pixels (MAE), normalized pixels, or features (dae, LDM latent). Choosing the target chooses what the model keeps. See Ch. 5, 7. Related: reconstruction loss; decoder; masking.

Rectified flow — the deterministic ODE transport between noise and data that a flow-matching model learns. Formally, sample $\boldsymbol{z} \sim \mathcal{N}(0, I)$, $\boldsymbol{x} \sim p_{\text{data}}$, $t \sim U[0,1]$, $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$, $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$, train $\mathcal{L} = \|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{v}\|^2$; "rectified" = paths are straightened so they can be integrated with few Euler steps. See Ch. 7. Related: flow matching; velocity field; transport map.

Redundancy reduction — the objective family pushing different embedding dimensions to decorrelate. Formally, Barlow Twins moves the cross-correlation matrix toward identity: diagonal $\to 1$ (invariance), off-diagonal $\to 0$ (decorrelation); removes the need for negatives. See Ch. 10. Related: cross-correlation matrix; invariance term; Barlow Twins.

Relative depth / relative pose — geometric quantities tying two frames; see cross-modal consistency (Ch. 14). Related: pose estimation; rigid flow.

Representation — synonymous with embedding; the vector the encoder returns. This book uses both terms interchangeably. See Ch. 1–14. Related: embedding; feature extractor; latent space.

Representation collapse — see collapse. Related: collapse.

Resampling (hierarchical) — reweighting samples inside the hierarchical k-means tree so each subtree contributes roughly equally. Formally the step that flattens long-tailed raw distributions (Vo et al.); part of the DINOv3 data engine. See Ch. 11. Related: hierarchical k-means; data balancing.

Retrieval-based curation — selecting the uncurated crawled samples that are closest to curated seed examples. Formally, embed the uncurated pool, retrieve top-k nearest neighbors for each seed image, union the results into the augmented curated dataset; step 3 of the DINOv2 pipeline. See Ch. 11. Related: data engine; seed dataset; curated vs crawl data.

Sampling (from a generative model) — producing synthetic data points. Flows invert the network; VAE/GAN start from a prior sample; diffusion iteratively denoise. See Ch. 6–7. Related: generator; latent space; diffusion model.

Score function — $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$, the gradient of the log density; denoising diffusion models—through Tweedie's identity—are score estimators. See Ch. 7. Related: score matching; denoising.

Score matching — training a model's score to match the data score without computing $\log p$ explicitly. Hyvärinen's original formulation; diffusion is its modern, noise-conditional instantiation. See Ch. 7. Related: score function; denoising.

Seed dataset — the small, trusted curated collection used to retrieve related data from the crawl. Formally, the "queries" of the DINOv2 curation pipeline (ImageNet-22k, Google Landmarks, splits of ImageNet-1k). See Ch. 11. Related: retrieval-based curation; data engine.

Self-distillation — training a network on the outputs of an earlier or averaged copy of itself. Formally, DINO: student predicts teacher's distribution, teacher = EMA of student; explicitly "distillation with no labels". See Ch. 10–12. Related: distillation; EMA teacher; student–teacher training.

Self-supervised learning (SSL) — learning from data where the supervision is derived from the data structure itself. No manual labels; signals come from augmentations, geometry, time, modalities, physics, or clustering. The subject of the book. Related: pseudo-labels; system-level SSL; generative model.

Self-training — using the current model's predictions as targets for the next training round. Pseudo-labeling, DeepCluster, and Noisy Student are instances; risks confirmation bias, mitigated by noise (Noisy Student) and ensembling. See Ch. 12. Related: pseudo-labels; Noisy Student; DeepCluster.

Semantic correspondence — matching points across images by meaning, not appearance. DIFT's emergent behavior: queries on one image find the semantically corresponding point on another, across instances, categories, and domains. See Ch. 7, 13. Related: DIFT; attention map; emergent behavior.

Sharpening — dividing the teacher's logits by a low temperature before softmax. In DINO, $\tau_t \approx 0.04-0.07$ vs $\tau_s \approx 0.1$; exaggerates the top prototype and prevents uniform predictions. See Ch. 11. Related: temperature; centering; prototype.

Siamese network — two branches of the same network processing two inputs, compared by a similarity loss. Historically for signature verification and tracking; modern SSL turns one branch into a moving target (stop-grad or EMA). See Ch. 9–10. Related: stop-gradient; twins; predictor head.

Stable diffusion — informal name for the latent diffusion family deployed by Rombach et al.; the course says latent diffusion = composition of VAE + GAN + flow/diffusion (Ch. 7). Related: latent diffusion.

Stacked autoencoder — a deep autoencoder trained layer-by-layer and then unrolled. Formally, first individual autoencoders with $\|\boldsymbol{z}-\hat{\boldsymbol{z}}\|^2$ are fit, then stacked into a deep encoder–decoder; historical solution to the vanishing-gradient problem. See Ch. 4. Related: layerwise pretraining; denoising autoencoder.

Stop-gradient — treating a tensor as a constant in the backward pass. Operationally $\boldsymbol{z} \leftarrow \mathrm{sg}(\boldsymbol{z})$ before computing the similarity term; without it Siamese losses collapse to constant outputs. See Ch. 9–11. Related: collapse; predictor head; momentum encoder.

Student–teacher training — matching the predictions of a student to those of a teacher. Forms: supervised distillation, self- distillation with EMA teacher, multi-teacher distillation (DUNE), cross-modal (MM-DistillNet). See Ch. 10–14. Related: distillation; self-distillation; DINO; distillation taxonomy.

Style vector — the intermediate latent $\boldsymbol{w}$ that controls StyleGAN's per-resolution appearance knobs. Formally, output of the mapping network, broadcast into per-layer affine scale/shift for AdaIN. See Ch. 6. Related: AdaIN; mapping network; disentanglement.

Super-resolution (as an SSL task) — predicting the high- resolution image from a low-resolution one; the degradation is synthetic, so the pair is "free". Formally used as a conditional flow example and as the classic case where paired data exists without humans labeling. See Ch. 6, 12. Related: conditional normalizing flow; paired data.

Synthesis network — StyleGAN's generator that grows a learned constant through conv+upsample blocks conditioned by AdaIN styles. See Ch. 6. Related: mapping network; AdaIN; style vector.

System-level pseudo-labels — pseudo-labels generated from the robot's integrated state over time: rendered 3D map, reprojected path, cross-view sensor occlusion checks. Their defining feature is that they aggregate many views before becoming one training target. See Ch. 14. Related: pseudo-labels; system-level SSL; continual learning.

System-level SSL — the course's closing idea: a robotic system learns online from everything its own sensors and actuators record, without any external labeling. Five case studies cover map supervision, geometric consistency, proprioceptive supervision, cross-modal distillation, and autonomous RL. See Ch. 14. Related: system-level pseudo-labels; continual learning; physical-laws view.

t-SNE — nonlinear dimensionality reduction for visualization that preserves neighborhood structure. Formally, an embedding minimizing KL between Gaussian similarities in high-D and Student-t similarities in 2-D; caveat: qualitative only. See Ch. 2, 8. Related: PCA; feature visualization.

Taxonomy of generative models — Goodfellow's tree: explicit tractable density (autoregressive, flows), explicit approximate density (VAE), implicit direct (GAN), implicit iterative (diffusion). Repeated at the start and end of lectures 4–5; glue for the book's Ch. 6–7. Related: generative model; explicit density; implicit density.

Teacher projector (TP) — a per-teacher linear head used in multi-teacher distillation (DUNE) to map each teacher's feature shape to the student's. Formally, one learned linear projection chosen by which teacher produced the target; enables heterogeneous teachers. See Ch. 12. Related: DUNE; multi-teacher distillation.

Temperature τ — the scale inside $\mathrm{softmax}(\boldsymbol{s}/\tau)$. Controls distribution sharpness: low τ sharpens (DINO teacher), high τ smooths (distillation target), mid (InfoNCE) tunes contrastive difficulty. See Ch. 9, 11–12. Related: softmax; sharpening; InfoNCE.

Tracking (emergent) — following semantic keypoints through a video from features trained for colorization or diffusion. Vondrick et al.'s colorizer and DIFT both exhibit this zero-shot behavior. See Ch. 7–8. Related: semantic correspondence; emergent behavior; DIFT.

U-Net — the encoder–decoder with skip connections used as the denoiser in diffusion models. Formally, resolution-pyramid convolutions with long skip links between matching scales; DIFT reads its intermediate features. See Ch. 7. Related: diffusion model; DIFT; feature map.

Unary potential — the per-site cost in a CRF energy. Formally, $\psi_u(x_i)$ in $E(\boldsymbol{x})$; usually $-\log p(\mathrm{class}_i \mid \boldsymbol{x})$ from the segmentation network. See Ch. 3. Related: CRF energy; pairwise potential.

Universal encoder — a single student trained to mimic several specialist teachers. DUNE's goal: one encoder performs 2D semantics, 3D geometry, and human-pose tasks without per-task fine-tuning. See Ch. 12. Related: DUNE; multi-teacher distillation.

Unpaired data — see paired vs. unpaired data. Related: paired vs. unpaired data.

VAE (variational autoencoder) — an autoencoder whose encoder outputs a distribution over latents and whose loss includes a KL regularizer toward a prior. Formally, ELBO = $\mathbb{E}_q[\log p(\boldsymbol{x} \mid \boldsymbol{z})] - D_{\mathrm{KL}}(q(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p(\boldsymbol{z}))$; the probabilistic backbone bridging autoencoders and flows. See Ch. 4. Related: ELBO; reparameterization trick; encoder; decoder.

Variance regularization — pushing per-dimension embedding variance away from zero against collapse. VICReg's main anti-collapse term, contrasted with Barlow Twins' redundancy reduction. See Ch. 10. Related: collapse; redundancy reduction; VICReg.

Velocity field — the vector field $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ that flow-matching models predict. Formally, trained against $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$ on linear interpolations; integrating the field gives samples. See Ch. 7. Related: rectified flow; Euler integration.

View — one augmented copy of a sample fed to one branch of a Siamese/contrastive pipeline. Two views make a positive pair; in multi-crop there are many "local views". See Ch. 8–12. Related: positive pair; augmentation; multi-crop.

Visible patch — the unmasked patches a MAE encoder actually sees. Only these go through the transformer, which is why the encoder can afford to be very wide. See Ch. 5. Related: masking ratio; mask token; MAE.

Weakly-supervised learning — training on labels that are cheap-but-noisy: hashtags (SWAG), alt-text (CLIP, WIT-400M), captions. DINOv2 matches their performance without any labels — the lecture's moment where unsupervised catches up with weak supervision. See Ch. 11. Related: data curation; CLIP.

World model — a learned simulator of observation evolution. Autoregressive: $\boldsymbol{x}_{t+1} \sim p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_{0:t})$; action- conditioned for planning; DINO-WM runs in frozen DINOv2 feature space. Whether it is "self-supervised" is a taxonomic debate in Ch. 13. Related: action-conditioning; planning loss; DINO-WM.

$\boldsymbol{x}$, $\boldsymbol{z}$ — the book's standing notation for a raw input and for its latent representation (see the Notation table in the front matter). Related: embedding; representation.

Zero-shot planning — deciding actions purely by rolling out a frozen world model toward a goal encoding, without further training. DINO-WM's headline capability. See Ch. 13. Related: world model; planning loss; DINO-WM.

CHAPTER C

Cheat Sheets

Five reference cards: (1) key equations, (2) the methods master table, (3) a one-line index of every paper cited in this book, (4) a terminology card, and (5) a checklist for reading or designing an SSL method.

C.1 Sheet 1 — Key Equations

Distances and similarities.

$$d_{\text{Euclid}}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{u} - \boldsymbol{v}\|_2, \qquad \text{sim}(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u}^\top \boldsymbol{v}}{\|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2} = \cos\theta, \qquad \text{(Ch. 2, 9)}.$$

For unit-normalized vectors, $\|\boldsymbol{u} - \boldsymbol{v}\|_2^2 = 2 - 2\,\text{sim}(\boldsymbol{u}, \boldsymbol{v})$.

KL divergence and ELBO.

$$D_{\text{KL}}(p \,\|\, q) = \mathbb{E}_{x \sim p}\Big[\log \tfrac{p(x)}{q(x)}\Big] \geq 0,$$

$$\log p(\boldsymbol{x}) \;\geq\; \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \,|\, \boldsymbol{z})\big] - D_{\text{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \,|\, \boldsymbol{x}) \,\|\, p(\boldsymbol{z})\big) \quad \text{(Ch. 4)}.$$

Contrastive family.

$$\mathcal{L}_{\text{triplet}} = \max\big(0,\, \|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{x}^+)\|_2^2 - \|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{x}^-)\|_2^2 + \epsilon\big) \quad \text{(Ch. 9)}.$$

$$\ell(i, j) = -\log \frac{\exp\big(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau\big)}{\sum_{k=1}^{2N} \mathbb{1}[k \neq i] \, \exp\big(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau\big)} \quad \text{(SimCLR / NT-Xent, Ch. 9)}.$$

Interpretation: $I(\boldsymbol{z}_i; \boldsymbol{z}_j) \geq \log(2N-1) - \ell(i, j)$ — the InfoNCE bound with $2N-1$ candidates (van den Oord et al., 2018b; Poole et al., 2019).

MoCo dictionary view.

$$\mathcal{L}_{\text{MoCo}} = -\log \frac{\exp\left(\boldsymbol{q}^\top \boldsymbol{k}^+/\tau\right)}{\exp\left(\boldsymbol{q}^\top \boldsymbol{k}^+/\tau\right) + \sum_{\boldsymbol{k}^- \in \text{queue}} \exp\left(\boldsymbol{q}^\top \boldsymbol{k}^-/\tau\right)},$$

$$\boldsymbol{\theta}_k \leftarrow m\,\boldsymbol{\theta}_k + (1-m)\,\boldsymbol{\theta}_q \quad \text{(Ch. 9)}.$$

Barlow Twins (no negatives at all).

$$\mathcal{L}_{\text{BT}} = \underbrace{\sum_i (1-\mathcal{C}_{ii})^2}_{\text{invariance}} + \lambda \underbrace{\sum_i \sum_{j \neq i} \mathcal{C}_{ij}^2}_{\text{redundancy reduction}},$$

$$\mathcal{C}_{ij} = \frac{\sum_b z^A_{b,i}\, z^B_{b,j}}{\sqrt{\sum_b (z^A_{b,i})^2}\,\sqrt{\sum_b (z^B_{b,j})^2}} \quad \text{(Ch. 10)},$$

with each column first mean-centred over the batch (so $\mathcal{C}_{ij}$ is a Pearson correlation).

SimSiam symmetric loss (stop-gradient).

$$\mathcal{L} = \tfrac{1}{2} D(p_1, \text{sg}(z_2)) + \tfrac{1}{2} D(p_2, \text{sg}(z_1)),$$

$$D(p, z) = -\frac{p}{\|p\|_2} \cdot \frac{z}{\|z\|_2} \quad \text{(Ch. 10)}.$$

DINO / self-distillation.

$$\mathcal{L}_{\text{DINO}} = -\sum_k p_t(k)\, \log p_s(k),$$

$$p_s = \text{softmax}\big(g_{\boldsymbol{\theta}_s}(\boldsymbol{x})/\tau_s\big), \quad p_t = \text{softmax}\big((g_{\boldsymbol{\theta}_t}(\boldsymbol{x}) - c)/\tau_t\big), \qquad \text{(Ch. 11)}.$$

Teacher: $\boldsymbol{\theta}_t \leftarrow m\,\boldsymbol{\theta}_t + (1-m)\,\boldsymbol{\theta}_s$; center: $c \leftarrow m\,c + (1-m)\,\frac{1}{B}\sum_i g_{\boldsymbol{\theta}_t}(\boldsymbol{x}_i)$. KoLeo regularizer (spread): $\mathcal{L}_{\text{KoLeo}} = -\frac{1}{n}\sum_{i=1}^n \log d_{n,i}$, where $d_{n,i}$ is the distance of $\boldsymbol{z}_i$ to its nearest in-batch neighbor.

Generative modeling toolbox.

Change of variables: $\log p_X(\boldsymbol{x}) = \log p_Z(g(\boldsymbol{x})) + \log\left|\det J_g(\boldsymbol{x})\right|$ (Ch. 6).

GAN minimax: $\min_G \max_D \; \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\big[\log\left(1 - D(G(\boldsymbol{z}))\right)\big]$ (Ch. 6).

Rectified flow: $\boldsymbol{x}_t = (1-t)\,\boldsymbol{x} + t\,\boldsymbol{z}, \quad \boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}, \quad \mathcal{L} = \left\|f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) - \boldsymbol{v}\right\|_2^2$ (Ch. 7).

CFG sampling: $\boldsymbol{v}^{\text{cfg}} = (1+w)\,\boldsymbol{v}^y - w\,\boldsymbol{v}^{\emptyset}$ (Ch. 7).

MAE.

$$\mathcal{L}_{\text{MAE}} = \sum_{i \in \mathcal{M}} \left\| \boldsymbol{x}_i - D(E(\boldsymbol{x}_{\text{visible}}))_i \right\|_2^2, \qquad \mathcal{M} = \text{masked patch indices (75\%)} \quad \text{(Ch. 5)}.$$

DUNE distillation.

$$\mathcal{L}_{\text{DUNE}} = \sum_{\text{teacher } j} \left\| \text{TP}_j(\boldsymbol{z}_{\text{student}}) - \boldsymbol{z}_j \right\|_2^2 \quad \text{(Ch. 12)},$$

conceptually: one projector TP_j per teacher, applied to the student's features.

WVN novelty signal.

$$\mathcal{L}_{\text{reco}} = \sum_i \delta_i \left\| \boldsymbol{f}_i - \hat{\boldsymbol{f}}_i \right\|^2, \qquad \delta_i = \begin{cases} 1 & \text{if segment } i \text{ is annotated} \\ 0 & \text{otherwise} \end{cases} \quad \text{(Ch. 14)}.$$

ReLMM exploration score (equation on the slide).

$$\tilde{G}(o_g, a_g) = \alpha\, \mathbb{E}\big[G^i(o_g, a_g)\big] + \beta\, \sigma\big(G^i(o_g, a_g)\big) \quad \text{(Ch. 14)}.$$

C.2 Sheet 2 — Methods Comparison Master Table

Method (year)	Family	Loss recipe	Prevents collapse by	Ch.
KDE	generative	kernel smoothing & sample	— (no encoder)	6
Autoencoder	reconstruct	$\|\boldsymbol{x} - D(E(\boldsymbol{x}))\|_2^2$	bottleneck size	4
DAE / stacked DAE	reconstruct	denoise & reconstruct	noise + bottleneck	4, 7
VAE	approx. density	ELBO = recon − KL	KL to prior	4
Real NVP / Glow	tractable density	exact $\log p$ via flows	invertibility constraint	6
GAN / StyleGAN / Pix2Pix	implicit density	minimax game	discriminator pressure	6
DDPM / score SDE	implicit iterative	denoise at each t / match score	noise schedule	7
Rectified Flow / FM	flow matching	regress velocity $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$	geometry of transport	7

Method (year)	Family	Loss recipe	Prevents collapse by	Ch.
Autoregressive (PixelRNN, GPT)	tractable density	chain-rule MLE	causal masking	7, 11
RotNet	pretext predict	4-way rotation classification	task diversity	8
Jigsaw / patch-location	pretext predict	permutation / position	permutation set size	8
Context Encoders	reconstruct	inpaint masked region	adversarial + L2	8
Colorization (static, video)	data-transform	predict ab channels	color structure	8
PIRL	contrastive	NCE + memory bank	negatives from bank	8–9
SimCLR	contrastive	NT-Xent / InfoNCE	$2N-2$ in-batch negatives	9
MoCo (v1–v3)	contrastive	InfoNCE over queue	EMA key encoder + queue	9
BYOL	negative-free	predictor to EMA target	EMA target (+predictor)	10
SimSiam	negative-free	symmetric stop-grad cosine	stop-gradient (+predictor)	10
Barlow Twins	negative-free	cross-correlation $\to I$	decorrelation penalty	10
VICReg	negative-free	var. + invar. + cov.	explicit variance floor	10
SwAV	clustering	contrast cluster assignments	Sinkhorn-balanced prototypes	10
DeepCluster	clustering	cluster $\to$ classify pseudo-label	re-cluster each epoch	3
DINO / DINOv2 / DINOv3	self-distillation	CE student vs. EMA teacher	EMA teacher + centering + sharpening	11
MAE	masked modeling	pixel MSE on masked patches	heavy masking + small decoder	5
BERT	masked modeling	token CE on [MASK]	bidirectional context	11
BEiT / iBOT	masked modeling	visual-token CE	tokenizer as teacher	5, 11
CLIP (weakly-sup.)	contrastive	image–text InfoNCE	text as labels	11
Pseudo-labels / Noisy Student	self-training	CE on argmax of teacher	noise on student; iterate	12
Classic KD (Hinton)	distillation	CE on soft targets, $T > 1$	supervised teacher	12

Method (year)	Family	Loss recipe	Prevents collapse by	Ch.
DUNE	multi-teacher distill	project each teacher, match	supervision from teachers	12
CycleGAN	implicit density	GAN + cycle consistency	cycle term pins generator	12
DINO-WM	world model	predict next DINO latent	frozen encoder	13
Frey et al. 2022	system-level	CE on rendered map labels	3D fusion consistency	14
DF-Net	system-level	photometric + cross-task	rigid-flow masks	14
WVN	system-level	traversability + reco	ground-truth through feet	14
MM-DistillNet	cross-modal distill	focal on NMS + MTA	teacher agreement	14
ReLMM	autonomous RL	RL on grasp reward	boots itself via resets	14

C.3 Sheet 3 — Papers Index (every entry in `references.bib`)

All 141 BibTeX entries, sorted by year. Each line: citation key; one-line short title. The full reference is in the bibliography.

1956 `kmeans1957` — earliest Lloyd-style partitioning scheme (Steinhaus; k-means ancestry).
1967 `macqueen1967some` — MacQueen's *Some Methods for Classification*, the original k-means formulation.
1975 `bentley1975kdtree` — the KD-tree.
1982 `lloyd1982least` — Lloyd's least-squares quantization (k-means; IEEE IT).
1990 `hutchinson1990stochastic` — stochastic trace estimator.
1992 `becker1992imax` — Becker & Hinton's Imax: mutual information between patches (Nature).
1993 `bromley1993siamese` — Siamese networks for signature verification.
1996 `ester1996dbscan` — DBSCAN density clustering.
1998 `indyk1998lsh` — approximate nearest neighbors via LSH (theory).
1998 `lecun1998gradient` — gradient-based learning for document recognition (LeNet).
1999 `ankerst1999optics` — OPTICS ordering-based clustering.
1999 `gionis1999lsh` — LSH for high-dimensional similarity search (VLDB).
2000 `tishby2000ib` — the information-bottleneck method.
2001 `lafferty2001crf` — conditional random fields.
2002 `jolliffe2002pca` — Jolliffe's PCA textbook.
2003 `sang2003conll` — CoNLL-2003 named-entity recognition benchmark.
2005 `chopra2005siamese` — discriminative similarity metric for faces (Siamese).

2005 `hyvarinen2005scorematching` — score matching (Hyvärinen).
2007 `arthur2007kmeanspp` — k-means++ seeding.
2008 `vandermaaten2008tsne` — t-SNE visualization.
2008 `vincent2008extracting` — denoising autoencoders.
2009 `deng2009imagenet` — ImageNet.
2010 `everingham2010pascal` — PASCAL VOC benchmark.
2010 `gutmann2010nce` — noise-contrastive estimation.
2010 `sculley2010webscale` — web-scale mini-batch k-means.
2010 `vincent2010stacked` — stacked denoising autoencoders.
2013 `lee2013pseudo` — pseudo-labels for semi-supervised deep learning.
2014 `goodfellow2014gan` — GANs.
2014 `kingma2014vae` — VAE: Auto-Encoding Variational Bayes.
2014 `lin2014coco` — MS COCO dataset.
2014 `mirza2014cgan` — conditional GAN.
2014 `rezende2014stochastic` — stochastic backpropagation / deep generative models.
2014 `zhou2014places` — Places database.
2015 `dinh2015nice` — NICE: first modern coupling flow.
2015 `doersch2015context` — relative patch-position pretext.
2015 `germain2015made` — MADE: masked autoencoder for distribution estimation.
2015 `hinton2015distilling` — knowledge distillation.
2015 `ioffe2015batchnorm` — BatchNorm.
2015 `rezende2015variational` — variational inference with normalizing flows.
2015 `ronneberger2015unet` — U-Net.
2015 `schroff2015facenet` — FaceNet: triplet loss at scale.
2016 `doersch2016tutorial` — VAE tutorial (Doersch).
2016 `kingma2016iaf` — IAF: inverse autoregressive flow.
2016 `noroozi2016jigsaw` — jigsaw pretext task.
2016 `oord2016pixelrnn` — PixelRNN autoregressive images.
2016 `pathak2016context` — Context Encoders inpainting.
2016 `radford2016dcgan` — DCGAN: convolutional GAN recipe.
2016 `reed2016text` — text-to-image GAN (GAN-CLS).
2016 `salimans2016improved` — improved GAN training (minibatch discrimination, Inception Score).
2016 `sohn2016npair` — N-pair loss.
2016 `theis2016note` — a note on the evaluation of generative models.
2016 `zhang2016colorization` — colorization pretext.
2017 `arjovsky2017wgan` — Wasserstein GAN.
2017 `devries2017cbn` — conditional batch normalization from language.
2017 `dinh2017realnvp` — Real NVP.
2017 `dumoulin2017cin` — conditional instance normalization for style.
2017 `gulrajani2017wgangp` — WGAN-GP: gradient penalty.
2017 `heusel2017ttur` — two time-scale update rule; FID.
2017 `isola2017pix2pix` — Pix2Pix paired translation.
2017 `odena2017acgan` — AC-GAN: auxiliary classifier GAN.
2017 `papamakarios2017maf` — MAF: masked autoregressive flow.
2017 `vaswani2017attention` — the Transformer.
2017 `zhang2017splitbrain` — split-brain autoencoders.
2017 `zhou2017ade20k` — ADE20K scene-parsing dataset.
2017 `zhu2017cyclegan` — CycleGAN unpaired translation.
2018 `caron2018deepcluster` — DeepCluster.
2018 `chen2018neuralode` — Neural ODEs.

2018 `gidaris2018rotnet` — RotNet rotation prediction.
2018 `huang2018naf` — neural autoregressive flows.
2018 `karras2018progan` — progressive growing of GANs.
2018 `kingma2018glow` — Glow: invertible 1×1 conv.
2018 `mescheder2018converge` — which GAN training methods converge (R1, Dirac-GAN).
2018 `miyato2018projection` — projection discriminator for conditional GANs.
2018 `miyato2018spectral` — spectral normalization for GANs.
2018 `oord2018cpc` — Contrastive Predictive Coding.
2018 `oord2018parallelwavenet` — Parallel WaveNet: probability density distillation.
2018 `peters2018elmo` — ELMo contextual word embeddings.
2018 `radford2018gpt` — GPT-1 generative pretraining.
2018 `vondrick2018tracking` — video colorization yields tracking.
2018 `zellers2018swag` — SWAG commonsense dataset.
2018 `zou2018dfnet` — DF-Net depth/flow cross-consistency.
2019 `brock2019biggan` — BigGAN: large-scale class-conditional GAN.
2019 `devlin2019bert` — BERT.
2019 `durkan2019nsf` — neural spline flows.
2019 `grathwohl2019ffjord` — FFJORD continuous flows.
2019 `karras2019stylegan` — StyleGAN.
2019 `kingma2019introduction` — Kingma & Welling's VAE monograph.
2019 `nalisnick2019ood` — deep generative models and out-of-distribution likelihoods.
2019 `poole2019variational` — variational bounds of MI.
2019 `sablayrolles2019koleo` — KoLeo spreading regularizer.
2019 `winkler2019learning` — conditional normalizing flows for super-resolution.
2020 `caron2020swav` — SwAV: contrast cluster assignments.
2020 `chen2020mocov2` — MoCo v2 (improved baselines).
2020 `chen2020simclr` — SimCLR.
2020 `chen2020simclrv2` — SimCLRv2 semi-supervised.
2020 `grill2020byol` — BYOL: no negatives.
2020 `he2020moco` — MoCo.
2020 `ho2020ddpm` — DDPM.
2020 `karras2020ada` — ADA: adaptive discriminator augmentation.
2020 `karras2020stylegan2` — StyleGAN2.
2020 `lugmayr2020srflow` — SRFlow: flow-based super-resolution.
2020 `misra2020pirl` — PIRL.
2020 `tian2020infomin` — InfoMin views study.
2020 `xie2020noisy` — Noisy Student.
2020 `zhao2020diffaugment` — DiffAugment: differentiable augmentation for GANs.
2021 `caron2021dino` — DINO.
2021 `chen2021mocov3` — MoCo v3 (ViT contrastive).
2021 `chen2021simsiam` — SimSiam.
2021 `dhariwal2021beatgans` — diffusion beats GANs (classifier guidance).
2021 `dosovitskiy2021vit` — ViT.
2021 `karras2021stylegan3` — StyleGAN3: alias-free generation.
2021 `papamakarios2021nfreview` — normalizing flows review.
2021 `radford2021clip` — CLIP.
2021 `song2021ddim` — DDIM deterministic sampling.
2021 `song2021sde` — score-based SDE framework.
2021 `valverde2021mmdistill` — MM-DistillNet audio-from-multimodal detection.
2021 `zbontar2021barlow` — Barlow Twins.
2022 `bao2022beit` — BEiT.

2022 `bardes2022vicreg` — VICReg.
2022 `frey2022continual` — continual 2D-3D adaptation with map pseudo-labels.
2022 `hamilton2022stego` — STEGO unsupervised segmentation.
2022 `he2022mae` — MAE.
2022 `ho2022classifierfree` — classifier-free guidance.
2022 `rombach2022ldm` — latent diffusion.
2022 `sun2022relmm` — ReLMM autonomous real-world RL.
2022 `zhou2022ibot` — iBOT.
2023 `brooks2023instructpix2pix` — InstructPix2Pix instruction-based editing.
2023 `chi2024diffusionpolicy` — Diffusion Policy.
2023 `lipman2023flowmatching` — Flow Matching.
2023 `liu2023rectifiedflow` — Rectified Flow.
2023 `mattamala2024wild` — WVN traversability in the wild.
2023 `tang2023dift` — DIFT: correspondence from diffusion features.
2024 `gemini2024report` — Gemini 1.5 report (distillation at scale).
2024 `leroy2024mast3r` — MASt3R 3D-aware matching.
2024 `oquab2024dinov2` — DINOv2.
2024 `ranzinger2024amradio` — AM-RADIO agglomerative foundation encoder.
2024 `vo2024automatic` — automatic data curation (hierarchical k-means; DINOv3 data engine).
2024 `zhou2025dinowm` — DINO-WM zero-shot planning in DINO space.
2024 `zhu2024snislam` — SNI-SLAM (mapping animation in slides).
2025 `sariyildiz2025dune` — DUNE universal encoder.
2025 `simeoni2025dinov3` — DINOv3.

C.4 Sheet 4 — Terminology Card

Twenty-five terms to have at your fingertips; each is defined fully in the glossary (Appendix B).

Term	Remember it as
Pretext task	label generated from $\boldsymbol{x}$ itself (rotation, jigsaw, inpainting)
Downstream task	the labeled task you evaluate on after pretraining
Linear probing	freeze encoder; train one linear head
Fine-tuning	unfreeze everything, small learning rate
Positive / negative pair	same sample vs. different sample in contrastive loss
In-batch negatives	batch mates used as negatives
Memory bank / queue	rolling store of features reused as negatives
Projection head	MLP whose output is used for loss, then discarded
EMA / momentum encoder	$\boldsymbol{\theta}_t \leftarrow m\boldsymbol{\theta}_t + (1-m)\boldsymbol{\theta}_s$
Stop-gradient sg	forward identity, backward zero
Collapse	all inputs map to the same vector
Centering & sharpening	DINO: subtract mean, then divide by low τ
Prototype	DINO column / SwAV cluster vector
Bottleneck	narrow layer that forces compression
ELBO	reconstruction − KL; lower bound on $\log p(\boldsymbol{x})$
Change of variables	volume-correction through $\log \lvert \det J \rvert$
Affine coupling layer	invertible layer with triangular Jacobian
Score function	$\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$, estimated via denoising
Velocity field	vector field that rectified flow learns to integrate
Mutual information bound	InfoNCE lower-bounds I up to $\log N$
Pseudo-labels	model-generated targets (argmax / cluster id / rendered map)
Self-distillation	teacher = EMA of the same student
World model	learned predictor of the next observation (maybe given actions)
Planning loss	distance between imagined future and goal latent
System-level SSL	learning from everything the robot produces, zero external labels

C.5 Sheet 5 — Researcher's Checklist

Questions to ask when reading an SSL paper, and again when designing your own method. The chapter numbers point to where this book develops the concern.

A. When reading someone else's method.

1. *Where does the supervision actually come from?* Identify the exact object that plays the role of a label: another view, another modality, another time step, a cluster, a rendered map, the EMA teacher. (Ch. 1–2.)
2. *What prevents collapse?* Negatives (contrastive), target EMA (BYOL-type), stop-gradient (SimSiam), variance/cross-correlation penalty (Barlow Twins/VICReg), centering + sharpening (DINO), or non-triviality by construction (generative, geometric). If the answer is "the architecture is asymmetric", verify empirically. (Ch. 9–11.)
3. *What does the loss optimize, in information-theoretic terms?* Is it an MI lower bound (InfoNCE), a reconstruction likelihood (MAE, VAE), a distributional matching (GAN), a deterministic transport (flow matching)? Each family makes different assumptions. (Ch. 4–9.)

4. *What is the evaluation protocol?* Linear probe, kNN, fine-tune, few-shot, dense prediction, low-data transfer. A method that wins at linear probing may lose at detection. Never average protocols. (Ch. 3, 5, 8, 10.)
5. *How does it fail?* Look for failure-mode sections. Typical: collapse, false negatives, shortcut learning from augmentations, confirmation bias in pseudo-labels, mode dropping in GANs, blurred reconstructions in VAEs, catastrophic teacher–student drift. (Distributed throughout; see the `failuremodes` boxes.)
6. *What is new vs. what is scale?* Separate algorithmic novelty (a new loss, a new architecture) from scaling effects (more data, more compute, a better data engine). Model your expected gain before running. (Ch. 11.)

B. When designing your own method.

1. *What structure in the data gives you signal for free?* Pick the invariance or symmetry you are willing to bake in. ("Anything better than chance can be exploited.")
2. *Write down the objective precisely.* The loss should be a number that goes down only when the representation improves, not a program with heuristic scaling terms.
3. *Diagnose collapse before scaling.* Train a tiny version. Measure variance of each embedding dimension, linear-probe after 1 epoch, watch for degenerate solutions.
4. *Choose an evaluation protocol up front and pre-register it* (informally at least: one table, frozen settings). If you iterate on the protocol you are overfitting to the benchmark.
5. *Check compute budget:* contrastive needs large batch (SimCLR) or memory engineering (MoCo queue); diffusion needs many steps; distillation needs two forward passes; reconstruction needs a decoder. Pick the one that fits your hardware. (Ch. 7–12.)
6. *Plan for ablations*: augmentation set, temperature, projection head capacity, EMA coefficient, masking ratio. If the method works only with a narrow slice, it probably does not work.
7. *Look at what the features actually encode*: PCA/t-SNE of embeddings, attention maps, correspondence probes (DIFT- style), first-layer filters. Direct inspection complements downstream numbers. (Ch. 8, 10.)
8. *Know what would falsify your explanation.* If the claim is "invariance drives it", an experiment that removes a source of invariance must hurt. If it does not, update the story. (Ch. 15.)

CHAPTER D

Slide Coverage Index

This appendix accounts for every slide in the eight lecture decks (*analysis/0X-*.md* is the authoritative slide-by-slide inventory this table compresses; lecture 01 is organizational and contains no technical content).

Numbering conventions differ between decks. Three conventions are in use: (i) *file index* — the slide-N key in the analysis inventories, which matches the extracted image/PDF page number; (ii) *footer page* — the number printed on each slide by the deck itself; (iii) *build frame* — decks that animate one logical slide over several PDF pages (05, 06) end up with a file index larger than the printed page count. The tables below always index by *file index*; a caption note states where the two conventions diverge. Across all eight decks the total is $34 + 39 + 43 + 69 + 73 + 55 + 41 + 48 = 402$ slides.

Lecture 02 — Classical Methods (34 slides)

File index and footer number coincide after the title slide (file 2 = footer page "1", a fixed -1 offset that does not produce ambiguity); the analysis uses the file index.

Slides (file index)	Topic	Book location
1–2	Title, agenda (NN / clustering / hybrids)	— (structure only)
3–5	kNN; descriptor sets; Euclidean vs. cosine distance; hypersphere	Ch. 2, §distances & metrics
6–9	KD-trees: insertion, query with path unwinding, worked example	Ch. 2, §nearest-neighbor search
10–12	LSH definition, (r, c, p_1, p_2)-sensitivity, L tables, ANN benchmarks	Ch. 2, §approximate NN
13–14	Unsupervised classification framing, "no free lunch" of clustering	Ch. 2–3
15–17	k-means Lloyd/E–M; k-means++; mini-batch k-means	Ch. 2, §clustering

Slides (file index)	Topic	Book location
18–21	DBSCAN, k-means vs. DBSCAN trade-offs, OPTICS, PCA/subsampling	Ch. 2, §density clustering
22–25	DeepCluster: iterative clustering + backprop, pseudo-labels, "better than chance can be exploited as a learning signal"	Ch. 3
26–29	STEGO observation, low-dim projection, similarity enhancement, cluster at inference	Ch. 3
30–32	CRF, unary/pairwise potentials; STEGO as a CRF-like energy	Ch. 3
33	STEGO architecture recap (no new content)	Ch. 3
34	Summary	—

Lecture 03 — Autoencoder (39 slides)

Slides (file index)	Topic	Book location
1	Title	—
2–3	AE general concept $\boldsymbol{x} \to \boldsymbol{z} \to \hat{\boldsymbol{x}}$; hyperparameters (capacity, regularization)	Ch. 4
4–5	Information bottleneck $I(X;T)$ vs. $I(T;Y)$	Ch. 4
6	Codebook / memorization failure	Ch. 4 (Failure Modes)
7–9	L2 reconstruction loss; stacked AE; BatchNorm remark	Ch. 4
10–12	VAE setup: prior, encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}\|\boldsymbol{x})$, decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}\|\boldsymbol{z})$	Ch. 4
13–14	Jensen's inequality, ELBO derivation from scratch	Ch. 4; Appendix A
15	VAE recap (repeat of 11)	Ch. 4
16–18	Gaussian decoder → L2; reparameterization trick; practical ELBO objective	Ch. 4
19–20	Frey-face & MNIST latent manifolds	Ch. 4
21–22	VAE as generative model; AI-art motivation	Chs. 4, 6

Slides (file index)	Topic	Book location
23–24	MAE paper splash; masking illustration	Ch. 5
25–31	Random masking; encode visible patches; mask tokens; asymmetric decoder	Ch. 5
32–35	MAE quiz/reveal slides; masking-ratio sweep (75/85/95%)	Ch. 5
36	MAE results: ViT-L linear/fine-tune numbers, COCO, ADE20K	Ch. 5
37–39	MAE as representation learner; linear probing vs. fine-tuning; masking-ratio curves	Ch. 5, 8

Lecture 04 — Generative Models (43 slides)

Slides (file index)	Topic	Book location
1	Title	—
2–3	Modeling $p(X)$; generative modeling *is* SSL (bridge from MAE)	Ch. 6 opening
4–8	KDE: species maps, kernels, Gaussian KDE, KDE as a generative model of digits	Ch. 6
9–13	Normalizing flows: bijective as VAE-alternative; change of variables; composition; triangular-Jacobian desiderata	Ch. 6
14–17	Real NVP: affine coupling, multi-scale factorization, samples	Ch. 6
18–20	Glow: invertible 1×1 conv, LU decomposition, non-invertibility danger, samples w/ temperature	Ch. 6
21–23	Conditional normalizing flows (notation flip: data y, condition x); super-resolution	Ch. 6
24	NF summary (strengths, restrictions)	Ch. 6
25–29	GAN build sequence: generator, discriminator, minimax, fixed-G/ fixed-D interpretation	Ch. 6

Slides (file index)	Topic	Book location
30–35	StyleGAN: mapping network $\boldsymbol{z} \to \boldsymbol{w}$, AdaIN, noise, style mixing, disentanglement; sample grid	Ch. 6
36–39	Pix2Pix: conditional GAN, conditioned D, L1 term; instructor warning that paired data is not strictly SSL	Ch. 6; caveat Ch. 12
40–42	GAN pros/cons; modeling ambiguity $P(x\|y)$ motivation	Ch. 6–7
43	Goodfellow taxonomy of generative models (closes Act I)	Ch. 6; Ch. 7 reopens

Lecture 05 — Generative Models & Denoising (69 slides)

The deck animates almost every logical slide over several PDF pages. The analysis inventory therefore uses the *file index* 1–69; the printed footer page numbers lag the file index by one (file 3 ↔ footer "2"). Both are noted where useful.

Slides (file index)	Topic	Book location
1–2	Title; exam reminder (admin)	—
3–6	Taxonomy recap; KDE; NF; GAN (bridge from lecture 04)	Ch. 6–7 bridge
7–11	Denoising intro animation; denoising AE; stacked DAE w/ layerwise pretraining	Ch. 7 Act I
12–16	Diffusion as learned gradual denoising (animated build)	Ch. 7
17	Duplicate build frame of 16	—
18	Concept comparison: VAE vs. NF vs. diffusion unrolled	Ch. 7
19–23	Rectified flow: setup, sampling of $(\boldsymbol{x}, \boldsymbol{z}, t)$, $\boldsymbol{x}_t = (1-t)\boldsymbol{x} + t\boldsymbol{z}$, $\boldsymbol{v} = \boldsymbol{z} - \boldsymbol{x}$, 5-line training code	Ch. 7 Act II
24–33	Euler sampling animation, T=50, sample code	Ch. 7
34–35	Rectified-flow summary; "what is rectified about it" (straight vs. curved transport paths)	Ch. 7

Slides (file index)	Topic	Book location
36–37	Continuous NF: instantaneous change of variables, $\log p(\boldsymbol{x}) = \log p(\boldsymbol{z}_0) - \int_0^T \operatorname{tr}[\partial f/\partial \boldsymbol{z}_t]\, dt$, Hutchinson trace trick	Ch. 7; Appendix A
38–39	Conditional rectified flow (code: condition passed in both train and sample)	Ch. 7
40–45	Classifier-free guidance: null condition, velocity extrapolation $\boldsymbol{v}^{\text{cfg}}$, doubled-cost caveat, full sample code	Ch. 7
46–52	Latent diffusion: VAE encoder–decoder stage, diffusion on latents, inference, VAE+GAN autoencoder composition, "VAE+GAN+flow+diffusion" banner	Ch. 7
53–58	Autoregressive interlude: MLE, log trick, chain rule, scanline subpixels, tile proposal	Ch. 7
59–64	DIFT: paper, cat/duck abstraction, U-Net diffusion features, cosine-similarity correspondence, cross-instance/category/domain results, downstream pose	Ch. 7
65–67	Diffusion Policy: visuomotor policy as action diffusion; warn "supervised imitation, not SSL"	Ch. 7 caveat; Ch. 13
68–69	Taxonomy loop-close: flow matching joins the explicit side	Ch. 7 closing

Lecture 06 — Pretext Tasks & Augmentations (73 slides)

Animation builds mean several file indices per logical slide; the *file index* (1–73) is used throughout.

Slides (file index)	Topic	Book location
1–9	Bridge from generative; diffusion features & interpolations as features; MAE teaser; feature extraction goal: abstraction with a meaningful metric	Ch. 8 opening

Slides (file index)	**Topic**	**Book location**
10–12	Augmentations catalog (image); time-series and text/NLP augmentation taxonomy	Ch. 8
13–14	Reconstruction-based SSL schema; Context Encoders inpainting	Ch. 8
15–17	Pretext schema; patch-relative-position; jigsaw (permutation set, shared towers)	Ch. 8
18–22	Colorization (L,ab); video colorization; emergent tracking of masks and keypoints	Ch. 8
23	Temporal-order tasks (shuffle verification, odd-one-out)	Ch. 8
24–28	RotNet worked example: paper, "visual commonsense", 4-way classification, filter equivalence, transfer table	Ch. 8
29–33	Evaluation taxonomy: direct inspection vs. downstream; linear probe vs. fine-tuning; kNN "low capacity in head"; heads; transfer-learning mapping	Ch. 8
34–37	Quiz slides (attention maps; transfer table; filters; keypoint tracking)	Ch. 8 exercises
38	Invariance vs. covariance	Ch. 8
39–43	PIRL: paper, two-encoder pretext-invariant alignment, memory bank, NCE formula, linear-probe transfer table	Ch. 8–9
44–47	Contrastive schema; Becker–Hinton & Siamese history, audio positive-pair example; loss ladder intro	Ch. 9
48–52	Triplet $\rightarrow$ N-pair $\rightarrow$ NCE $\rightarrow$ InfoNCE with formulas	Ch. 9
53	Cluster (pseudo-label) loss recalled	Ch. 3, 9
54–62	SimCLR: paper; augmentation families; Algorithm 1 line-by-line (4 build slides); few-label results; why a projection head; large-batch MI bound ($\log N$ shift)	Ch. 9
63–71	MoCo: dictionary look-up, queue, momentum/EMA encoder, PyTorch pseudocode walk (build slides)	Ch. 9

Slides (file index)	Topic	Book location
72	Barlow Twins preview (cross-correlation toward identity)	Ch. 10
73	Three-paradigm comparison (reconstruct / predict / agree) closing	Ch. 8–10 summary

Lecture 07 — Representation Learning (55 slides)

The deck contains an explicit numbering note: *slide images are numbered 1–55; internal footer numbers lag the image/file index by one because the title slide is unnumbered.* All citations here use the image/file index.

Slides (file index)	Topic	Book location
1	Title	—
2–5	Housekeeping; three worked exam examples (DBSCAN MCQ, clustering case study, deep-clustering code)	Ch. 2–3 review; Ch. 17
6–10	Feature space with meaningful metric; invariance/co-variance as purpose; three SSL families side-by-side; evaluation taxonomy	Ch. 10 opening, Ch. 8 recap
11–17	SimSiam: collapse phenomenon, loss-vs-kNN curves, stop-grad + predictor, pseudocode line-by-line	Ch. 10
18–23	BERT interlude: masked language modeling (15%), [CLS]/[SEP]/[MASK], NSP, bidirectional vs. GPT, frozen vs fine-tuned features	Ch. 11 “masked token modeling beyond vision”
24–32	DINO: self-distillation, centering + sharpening, EMA teacher, multi-crop, pseudocode	Ch. 11
33–36	kNN evaluation on frozen features; ImageNet linear/kNN comparison table	Ch. 10–11
37	Patch-level representations; emergent attention maps from [CLS]-query	Ch. 11
38–41	DINOv2 losses: DINO on [CLS], iBOT on masked patches, KoLeo; loss stack built slide-by-slide	Ch. 11

Slides (file index)	Topic	Book location
42–44	DINOv2 comparisons: self-supervised vs. weakly supervised (CLIP, SWAG, OpenCLIP, EVA-CLIP match)	Ch. 11
45–51	DINOv3; scale (1.3 M → 142 M → 1689 M); data sources; four-step DINOv2 curation pipeline (dedup, NSFW, retrieval)	Ch. 11
52–54	Hierarchical k-means with resampling (Vo et al. Algorithm 1), balancing equation, hierarchical sampling, "data engine"	Ch. 11
55	Summary	—

Lecture 08 — Mix: Self-Training, Cycle Consistency, World Models (41 slides)

Footer numbers lag the file index by one (title slide unnumbered, footer ranges 2–40).

Slides (file index)	Topic	Book location
1–2	Title; Haribo framing (mixed assortment)	—
3	Overcoming data limitations; multi-crop assumption; transfer axes	Ch. 12 opening
4–7	Student–teacher framework; pseudo-labels (argmax + CE, Lee 2013 numbers)	Ch. 12
8–10	DeepCluster as teacher; Noisy Student pipeline; iterative self-training surpassing teacher	Ch. 3, 12
11–13	Classic distillation (Hinton): soft targets carry information argmax hides; teachers can be ensembles or EMA	Ch. 12
14–16	DINO = "distillation with no labels"; distillation for compression; Gemini 1.5 Flash example	Ch. 11 recap, Ch. 12

Slides (file index)	Topic	Book location
17–22	DUNE: heterogeneous 2D/3D teachers, teacher projector heads, no per-teacher filtering, student surpassing teachers	Ch. 12
23	Haribo transition (no content)	—
24–26	Paired vs. unpaired data; when mined pairs still count as SSL	Ch. 12 taxonomy
27	GAN recap (minimax, fixed-G/fixed-D)	Ch. 6 callback
28–31	CycleGAN: two intertwined GANs, cycle consistency loss, results	Ch. 12
32	Haribo transition (no content)	—
33–34	Abstraction in feature space (t-SNE-style embedding projector); DIFT emergent correspondence callback	Ch. 13 opening (and Ch. 7)
35–37	World models: autoregressive as simplest case; action conditioning; "is it SSL?" debate	Ch. 13
38–41	DINO-WM: world model on frozen DINOv2 features, planning by test-time action optimization, push-T rollout animation	Ch. 13

Lecture 09 — System-Level SSL (48 slides)

The analysis notes that *slide-file numbers and footer page numbers drift after build slides*: file indices run 1–48, footer pages 1–47 (reused at slide 48). The table below uses the file index; where the caption for a system is referenced in the chapter text, both are given.

Slides (file index)	Topic	Book location
1–3	Title; exam logistics (no derivations; choose/implement losses)	—
4–6	Paired vs. unpaired data recap; sensor rigs; system-level SSL definition	Ch. 14 opening
7–14	Frey 2022: 2D-3D map accumulation, rendering pseudo-labels, continual learning with replay, raw-teacher/pseudo/adapted panels, uncertainty RViz, curiosity planner results	Ch. 14, case study 1

Slides (file index)	**Topic**	**Book location**
15–23	DF-Net: cross-task consistency; optical-flow geometry; flow from depth+pose; invalid masks; forward-backward consistency; architecture and losses; results	Ch. 14, case study 2
24–29	WVN: traversability from proprioception; DINO-feature AE; reconstruction-error novelty; decision-aware loss $(1 - c(f_n))$; results	Ch. 14, case study 3
30–35	MM-DistillNet: audio student from RGB/thermal/depth teachers; NMS pseudo-ground-truth, focal + MTA losses; demo	Ch. 14, case study 4
36–41	ReLMM: grasp algorithm with ensemble uncertainty exploration; autonomous reset/pseudo-reset; navigation algorithm (SAC, relabel), training montage	Ch. 14, case study 5
42–47	Summary: system-level definition reprise; paired modalities / physical laws / uncertainty as the three axes	Ch. 14, closing taxonomy
48	Administrative outro (guest lecture, master theses)	—

CHAPTER E

Study-Session Transcripts (Curated)

This appendix collects, in lightly copyedited form, the exam-preparation study sessions the author held with an AI mentor while working through the course slides. Nothing here replaces the chapters; its role is to preserve the step-by-step derivations, analogies, and worked numbers exactly as they were argued, including the questions that forced each clarification. Compiled from the sessions for Lectures 02–08; Lecture 09 (system-level SSL) was not part of those sessions — no study-session transcript for that deck exists, so no section for it appears here.

Each section opens with a one-paragraph editorial preface naming the companion chapters, then reproduces the session's substantive exchanges as Student–Mentor dialogue with the derivations kept in full, and closes with three revision questions the student implicitly asked, mapped to the exercise sets of Chapter 29.

E.1 Lecture 02: classical methods — metrics, indexing, clustering, DeepCluster, STEGO

Companion chapters: Chapter 2 (foundations/tools) and Chapter 3 (classical–SSL hybrids). The session worked through the 34-slide deck in six topic blocks: distance metrics and the high-dimensional curse; KD-trees and LSH; k-means and DBSCAN; DeepCluster; STEGO; and conditional random fields.

Student. The lecture opens with k-nearest neighbors. I keep seeing "k-NN evaluation" in self-supervised papers — but k-NN is a classifier that needs labels, so how can an unsupervised method use it? And is it even a model?

Mentor. Read the algorithm more carefully than the name suggests. k-NN stores the entire training set; for a query $\boldsymbol{x}$ it computes the distance to every stored point, selects the k closest, and takes a majority vote. Two properties follow. First, it is *non-parametric*: no weights are learned and no objective is optimized — "training" consists of storing the dataset. Second, and this is the exam point, the *search* over neighbors is purely distance-based and requires no labels at all; labels enter only at the final voting step. The neighbor search therefore works unchanged in unsupervised settings such as retrieval.

This decoupling is exactly how SSL papers use it. During pretraining the backbone is frozen, representations of the training set are extracted, and a k-NN classifier is run on top — the so-called *k-NN monitor*. If k-NN accuracy rises over epochs, the unsupervised representations are separating into semantic classes without the network ever seeing a label. In the lecture's words, k-NN is the standard *non-parametric evaluation protocol* of SSL.

This resolves Quiz Q1 directly: k-NN is best described as a similarity-based method usable with or without labels. The distractors fail as follows. "Inherently supervised" fails because the retrieval step ignores labels. "Parametric model trained by optimization" fails because there are no learned parameters. "Learns cluster centroids" confuses k-NN with k-means: k-NN keeps every data point rather than abstracting them into centroids (MacQueen, 1967).

Student. So everything hinges on the distance metric. The lecture gives Euclidean and cosine — how do they differ in practice, and which one should I reach for?

Mentor. For vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^D$:

$$d_{\text{Eucl}}(\boldsymbol{a}, \boldsymbol{b}) = \|\boldsymbol{a} - \boldsymbol{b}\|_2 = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2},$$

$$\text{sim}_{\cos}(\boldsymbol{a}, \boldsymbol{b}) = \cos\theta = \frac{\boldsymbol{a}^\top \boldsymbol{b}}{\|\boldsymbol{a}\|_2 \, \|\boldsymbol{b}\|_2}, \qquad d_{\cos}(\boldsymbol{a}, \boldsymbol{b}) = 1 - \text{sim}_{\cos}(\boldsymbol{a}, \boldsymbol{b}).$$

(A notation slip on the lecture slide writes "cosine distance" for $\boldsymbol{a}^\top \boldsymbol{b}/(\|\boldsymbol{a}\|\|\boldsymbol{b}\|)$; that formula is cosine *similarity*, ranging in $[-1, 1]$. Cosine distance is one minus it. The exam probes exactly this trap.)

The operative distinction is magnitude. Cosine similarity is invariant to positive rescaling of either argument, since $\text{sim}_{\cos}(k\boldsymbol{a}, \boldsymbol{b}) = (k\, \boldsymbol{a}^\top \boldsymbol{b})/(k\|\boldsymbol{a}\| \, \|\boldsymbol{b}\|) = \text{sim}_{\cos}(\boldsymbol{a}, \boldsymbol{b})$ for any $k > 0$; it measures the direction, not the portion size. Euclidean distance is the opposite: it sums squared coordinate differences without any normalization, so a single dimension with a large numerical range dominates. If coordinate k is rescaled by a factor S_k, then $d^2_{\text{scaled}}(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i \neq k} (a_i - b_i)^2 + S_k^2 (a_k - b_k)^2 \approx S_k^2 (a_k - b_k)^2$ as $S_k \to \infty$: one arbitrarily rescaled feature swallows all others. Hence Quiz Q3 (most scale-sensitive metric: Euclidean) and Quiz Q4 (cosine preferred *when direction matters more than magnitude*). The intuition: a 10-page essay and a 1000-page encyclopedia, both about space exploration, are far apart in Euclidean word-count space because of sheer volume, yet nearly coincident in angle because they share the proportions of "rocket", "orbit", "galaxy". Cosine reads the recipe; Euclidean reads the portion.

Consequently Euclidean distance demands feature normalization (z-scoring or min–max) before use, whereas Hamming and Jaccard, operating on discrete coordinates and set overlaps, are untouched by continuous scaling.

Student. The lecture then normalizes everything onto a unit hypersphere. What is actually gained — and is Euclidean distance then equivalent to cosine?

Mentor. On the unit hypersphere the two metrics coincide up to a monotone affine map, which is the cleanest small derivation of the whole lecture. Take $\|\boldsymbol{a}\|_2 = \|\boldsymbol{b}\|_2 = 1$ and expand:

$$\|\boldsymbol{a}-\boldsymbol{b}\|_2^2 = (\boldsymbol{a}-\boldsymbol{b})^\top(\boldsymbol{a}-\boldsymbol{b}) = \boldsymbol{a}^\top\boldsymbol{a} - 2\boldsymbol{a}^\top\boldsymbol{b} + \boldsymbol{b}^\top\boldsymbol{b} = 1 - 2(\boldsymbol{a}^\top\boldsymbol{b}) + 1 = 2\big(1 - \mathrm{sim}_{\cos}(\boldsymbol{a},\boldsymbol{b})\big).$$

Maximizing cosine similarity and minimizing squared Euclidean distance are the same optimization on S^{D-1}: $\mathrm{sim} \to 1$ corresponds to $d^2 \to 0$, and $\mathrm{sim} \to -1$ to $d^2 \to 4$. This is why DINO descriptors can be compared by cosine similarity regardless of how the geometry was trained (Caron et al., 2021).

There is also a collapse argument. If a network is trained naively to minimize Euclidean distance between positive pairs, a trivial global optimum is $f(\boldsymbol{x}) = \mathbf{0}$ for every $\boldsymbol{x}$: all distances vanish and the loss is perfectly minimized. The zero vector cannot live on a unit hypersphere, so L_2-normalizing features and optimizing cosine similarity deletes this cheat path. And there is a numerical convenience: for normalized vectors, cosine similarity *is* the dot product, so an entire database comparison is a single matrix multiplication.

The rule of practice the session distilled: *evaluate with the metric the network was trained to optimize.* A model trained with temperature-scaled cosine loss organizes its geometry angularly; probing it with Euclidean distance on unnormalized features reads the space in the wrong coordinates.

Student. Why does all of this matter so much in deep learning specifically? What goes wrong with nearest neighbors in very high dimensions?

Mentor. Distance concentration — the geometric face of the curse of dimensionality. As D grows, the volume of $\mathbb{R}^D$ expands exponentially and points become sparse; under mild i.i.d. assumptions on the coordinates, all pairwise distances concentrate around a single average value, so the contrast between nearest and farthest neighbor vanishes:

$$\lim_{D\to\infty} \frac{d_{\max} - d_{\min}}{d_{\min}} = 0.$$

When every point is roughly equidistant from the query, neighbor rankings are carried by noise, and a "nearest neighbor" result is semantically unstable. This is the substance of Quiz Q2.

Where does that leave the metric? On a hypersphere, angle still separates what Euclidean scale cannot, which is one more reason modern pipelines normalize. But caution: the flip of Quiz Q4's distractor is real — if magnitude carries meaning (physical velocities, counts, calibrated confidence), cosine is the wrong tool, because it declares $[3, 4]$ and $[30, 40]$ identical.

Student. Brute-force neighbor search over millions of embeddings is $O(ND)$ per query. The lecture introduces KD-trees as the classical fix — how do they achieve $O(\log N)$ search, and why does that break down?

Mentor. A KD-tree is a binary search tree over points that partitions space with axis-aligned hyperplanes, cycling through the coordinates (x, then y, then x again in 2D) (Bentley, 1975). Construction needs balance — split at medians — so depth stays $O(\log N)$.

Querying proceeds in two phases. *Downward:* branch on the splitting coordinate at each node until hitting a leaf; that leaf is a candidate, but the true nearest neighbor may lie just across any splitting plane crossed on the way down. *Backtrack and prune:* climbing back up, at each ancestor that splits dimension d at value v, check the distance from the query to the plane itself; prune the unexplored side if and only if

$$|q_d - v| \;\geq\; \text{best_dist},$$

because only a query hypersphere of radius best_dist that *crosses* the plane can contain a closer point on the far side.

A worked search makes the mechanism concrete. With the lecture's toy tree — root $(51, 75)$ splitting x, right child $(70, 70)$ splitting y, and the candidate chain $(50, 50), (55, 1), (60, 80)$ splitting x again — a query $q = (52, 52)$ goes right at the root ($52 \geq 51$; tentative best $d = \sqrt{(52-51)^2 + (52-75)^2} = \sqrt{530} \approx 23.02$), left at $(70, 70)$ ($52 < 70$; node distance $\sqrt{648} \approx 25.46$, no update), right at $(50, 50)$ ($52 \geq 50$; distance $\sqrt{8} \approx 2.82$, new best). On the way back up, the plane $x = 50$ is only $|52 - 50| = 2 < 2.82$ away, so the other branch must be explored; $(55, 1)$ and $(60, 80)$ give distances $\sqrt{2610}$ and $\sqrt{848}$, updating nothing. The same test at the root ($|52 - 51| = 1 < 2.82$) forces a visit to the root's left subtree as well; nothing there is closer, and the answer $(50, 50)$ stands. The example shows both why backtracking is mandatory and how pruning is what produces sublinear search.

In high dimensions the pruning inequality fails almost everywhere: the query hypersphere intersects essentially every splitting plane, every branch must be explored, and the search degrades toward $O(N)$ brute force. That KD-trees stop scaling is Quiz Q5. Distance concentration (the previous exchange) is why: when all points sit at similar distances, no cut separates them well.

Student. If exact indexes die in high dimensions, what is the approximate route — how does locality-sensitive hashing actually guarantee anything?

Mentor. Locality-sensitive hashing inverts the purpose of a hash function (Indyk and Motwani, 1998; Gionis et al., 1999). Standard hashing tries to avoid collisions; LSH deliberately makes *similar items* collide. A family of hash functions is called $(r,\ cr,\ p_1,\ p_2)$-sensitive if $d(x,y) \le r$ implies $\Pr[h(x) = h(y)] \ge p_1$ and $d(x,y) \ge cr$ implies $\Pr[h(x) = h(y)] \le p_2$, with $p_1 > p_2$ the defining gap. Note LSH families are metric-specific: there are different families for Euclidean distance, cosine similarity, Hamming, or Jaccard.

Neighboring points collide in one table only probabilistically, so LSH sharpens p_1/p_2 with two amplifiers. The *AND construction*: use one hash key that concatenates r independent hash values per table. Collision probabilities become p_1^r and p_2^r; raising both to a high power pushes the far-neighbor probability toward zero much faster than the near-neighbor one, suppressing false positives. The *OR construction*: keep ℓ independent tables altogether and return the union of buckets across tables. The probability that two close points collide in at least one table rises toward 1, recovering recall. With $p_1 = 0.8$, $p_2 = 0.4$, $r = 5$ hashes per key and ten tables, near neighbors collide with probability $1 - (1 - 0.8^5)^{10} \approx 0.98$ while far ones drop to $1 - (1 - 0.4^5)^{10} \approx 0.10$ — exactness is traded for speed in a controllable way.

The benchmark picture (ann-benchmarks.com) closes the block: tree-based exact methods (`kd`, `ckdtree`) track brute force on high-dimensional embeddings; modern approximate indexes — graph-based HNSW, quantization-based FAISS/ScaNN — sit high on a recall-versus-queries-per-second frontier. In current vector databases, multi-table LSH is largely supplanted by graph traversal (a "navigable web" of points), but the AND/OR amplification logic is the conceptual ancestor and is what the exam asks about.

Student. Moving to clustering: the lecture insists k-means and DBSCAN make "different geometric assumptions." What exactly does each assume, and where does each fail?

Mentor. k-means is an EM-style alternating minimization of within-cluster squared error (the distortion):

$$\min_{\{z_i\},\{\boldsymbol{\mu}_j\}} \sum_{i=1}^{N} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|_2^2,$$

cycling an E-step (assign each $\boldsymbol{x}_i$ to its nearest centroid) and an M-step (recompute each centroid as the mean of its assigned points) (MacQueen, 1967; Lloyd, 1982). Two structural facts: (i) each step never increases the objective, and the number of partitions is finite, so convergence to a local minimum is guaranteed — the global minimum is not; (ii) the decision regions are Voronoi cells, hence linear, convex boundaries, and the spherical-Euclidean objective bakes in the assumption that clusters are roughly equal-variance blobs. Hence the classic failures: on the "two moons" dataset k-means slices each crescent with a linear cut; with K larger than the number of semantic groups it cheerfully invents artificial sub-clusters; and it is exquisitely sensitive to initialization, which is why the community standard is k-means++ — choose the first centroid uniformly at random, then each subsequent centroid with probability proportional to its squared distance from the nearest already-chosen centroid (Arthur and Vassilvitskii, 2007) — and to scale, which is why mini-batch k-means (an online, SGD-like update on random subsets) is the variant used when N is in the millions. Quiz Q7–Q10 play out exactly these points: spherical assumption, crescent failure, over-clustering, initialization sensitivity.

DBSCAN abandons centroids for density (Ester et al., 1996). Two parameters: a neighborhood radius ε and a density threshold minPts. A point with at least minPts neighbors within ε is a *core* point; a non-core point within ε of a core point is a *border* point; everything else is *noise*. Clusters are maximal sets of density-connected core points plus their borders. Arbitrary shapes fall out of this: clusters can be rings, snakes, anything connected through dense tubes, and the number of clusters is discovered rather than prescribed. The failure modes are just as characteristic: a single global ε cannot serve clusters of different densities (one value either merges dense neighbors or labels sparser clusters as noise); with ε too small almost everything becomes noise because the minPts threshold is never met; its naive all-pairs neighborhood computation costs $O(N^2)$ memory; and in high dimensions, distance concentration makes any ε uninformative — which is why the lecture pairs it with OPTICS (an ordering-based variant that avoids fixing ε) (Ankerst et al., 1999) and with PCA as a preprocessing escape hatch.

For deep embeddings the conclusion is asymmetric: k-means (especially mini-batch) is the workhorse, because it is $O(NDK)$ streaming, PCA-safe, and indifferent to absolute scale after normalization; DBSCAN's $O(N^2)$ memory and ε fragility make it impractical at embedding scale.

Student. DeepCluster puts clustering *inside* the training loop: cluster the features, treat cluster ids as "pseudo-labels", train the network to predict them. Isn't that circular — and how does it possibly get started?

Mentor. The bootstrap works because convolutional architectures carry a prior even with random weights. The DeepCluster paper's striking observation: a small (MLP) classifier on top of a *frozen, randomly initialized* AlexNet already achieves about 12% top-1 on ImageNet, against a chance level of 0.1% (Caron et al., 2018). Random convolutions act as structured edge- and texture-detectors, grouping images with similar local statistics. The guiding sentence of the lecture — and a fair one-line summary of the philosophy of much of SSL — is: *anything better than chance can be exploited as a learning signal.*

The pipeline alternates once per epoch (Caron et al., 2018): extract features for the entire dataset with the encoder $f_{\boldsymbol{\theta}}$; reduce them with PCA (to 256 dimensions — close the loop from the indexing block: k-means needs a metric that behaves, and high-dimensional distances concentrate); cluster with k-means into assignments $z_i = y_i$, the pseudo-labels; then optimize an ordinary cross-entropy classification loss

$$\min_{\boldsymbol{\theta}, W} \; \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{\mathrm{CE}}\big(g_W(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)),\, y_i\big)$$

with a linear head g_W, and repeat. The analogy: a student categorizing a library in a language they cannot read. First pass groups books by cover color and thickness; you declare those groups to be "classes" and force the student to study features that reproduce them; each pass forces attention to deeper structure until the groupings align with genuine topics.

Two safeguards fight the obvious collapse. Mapping everything to one cluster achieves a trivially perfect cross-entropy (the classifier just always predicts the majority class), so DeepCluster (i) re-seeds centroids of empty clusters, and (ii) weights the classification loss inversely to cluster size, penalizing solutions that hoard points into a dominant bin.

Results (linear probe on frozen features): DeepCluster reaches roughly 38–41% on ImageNet depending on the AlexNet layer, peaking around conv3 rather than conv5: late layers overfit the pseudo-label partition boundary, so mid-level features transfer best. On Places the conv5 features reach about 34.7% — evidence of genuine cross-dataset generality rather than dataset-specific memorization (Caron et al., 2018).

Two quiz answers complete the block. Q6 (every cluster mixes many object categories): the culprit is poor features, not a bad K. k-means can only partition the space it is given; if the embeddings of dogs, cars, and trees overlap, no clustering algorithm can separate them — the diagnosis must point at the representation, not the algorithm. Q12 (the major risk of cluster-as-pseudo-label training): a self-reinforcing feedback loop. The network is both label generator and label consumer. If the clustering step makes a systematic error — say it groups animals by background colour — those assignments become targets, gradient descent reshapes the features to preserve them, and the next epoch's clustering sees an even more distorted space. Success requires the initial random prior to be systematically informative, exactly the 12%-of-AlexNet point.

Student. STEGO turns this from image-level to pixel-level: unsupervised semantic segmentation. What is it distilling, and why a frozen DINO?

Mentor. The starting observation is empirical (Hamilton et al., 2022): the dense patch features of self-supervised Vision Transformers such as DINO already carry exquisitely semantic *correspondences*. A pixel marked on the sky in one image has, in the frozen backbone's feature space, high cosine similarity to sky pixels in entirely different images; likewise grass-to-grass, rider-to-rider. The correspondence field already respects semantic boundaries; it is merely high-dimensional, noisy, and not clustered. STEGO's move is to train a lightweight convolutional *segmentation head* on top of the frozen backbone that distills this correspondence structure into a compact, low-dimensional space, whose clusters are then aligned to segments.

The whole system rests on the quiz answer to Q13: the goal of such methods is to *discover semantic structure without labels* — not supervised segmentation, not image classification. And the architecture matters because ViTs compute global attention relations in every layer, which literally contains the correspondence information STEGO reads out; CNNs, with their local receptive fields, do not pack it so accessibly.

Student. Walk me through STEGO's correspondence distillation loss — especially that bias b, which looks mysterious.

Mentor. For two images with frozen-backbone feature maps f and g, and learnable head outputs s and t for the same two images, write the pairwise cosine similarity between backbone features at pixel location (h, w) of the first image and (i, j) of the second:

$$F_{hwij} \;=\; \frac{f_{hw} \cdot g_{ij}}{\|f_{hw}\|_2 \, \|g_{ij}\|_2} \;\in [-1, 1], \qquad S_{hwij} \;=\; s_{hw} \cdot t_{ij},$$

and minimize the correspondence distillation loss

$$\mathcal{L}_{\text{corr}}(f, g) \;=\; -\sum_{h,w,i,j} (F_{hwij} - b) \cdot S_{hwij},$$

with a scalar bias hyperparameter b (Hamilton et al., 2022). Minimizing $\mathcal{L}_{\text{corr}}$ drives S_{hwij} up where $F_{hwij} > b$ and down where $F_{hwij} < b$: the loss forces the segmentation head's dot products to reproduce the frozen backbone's similarity structure.

The bias b is the anti-collapse switch, and the gradient computation shows exactly why. Since $\partial \mathcal{L}_{\text{corr}} / \partial S_{hwij} = -(F_{hwij} - b)$, gradient descent updates S_{hwij} in proportion to $F_{hwij} - b$:

Attractive regime ($F_{hwij} > b$): the update raises S_{hwij}, pulling s_{hw} and t_{ij} toward each other — pixels the backbone considers semantically alike are merged.

Repulsive regime ($F_{hwij} < b$): the update lowers S_{hwij}, pushing s_{hw} and t_{ij} apart — unrelated pixels are separated.

With $b = 0$ the forces are almost all attractive, because backbone similarities are mostly positive, and the segmentation head collapses to a constant embedding; b supplies uniform negative pressure that carves out the space. The exam sharpens this: set $b = 1.0$ and, since $F \leq 1$ with equality only on the diagonal, *every* off-diagonal term flips repulsive; no attractive cluster can form and the learned representation fragments into unusable chaos.

STEGO applies this loss across three correspondence sources — within an image, between k-NN-retrieved similar images, and between random pairs. Cross-view matching is precisely why Q14's answer is "it enforces consistency of semantic regions": identical semantic content is forced to the same low-dimensional region across illumination, viewpoint, and scale changes.

Student. Does STEGO cluster during training like DeepCluster — and where does the CRF fit?

Mentor. No clustering during training — that is the deliberate contrast with DeepCluster (Hamilton et al., 2022). DeepCluster must re-cluster the full dataset every epoch; STEGO keeps training continuous and cheap. The training pipeline: sample images likely to share classes (via k-NN on backbone features), extract features, project through the segmentation head, and distill correspondences. Inference is where discreteness arrives: cluster the projected features with k-means to assign each pixel a class, then apply a conditional random field to snap boundaries to image edges.
The CRF (Lafferty et al., 2001) treats segmentation as minimizing an energy over label assignments $\boldsymbol{x} = (x_1, \ldots, x_N)$:

$$E(\boldsymbol{x}) \;=\; \sum_i \psi_u(x_i) \;+\; \sum_{i<j} \psi_p(x_i, x_j),$$

where the unary potential $\psi_u(x_i) = -\log P(x_i \mid \text{image})$ keeps the network's confident predictions honest, and the pairwise potential $\psi_p(x_i, x_j) = \mu(x_i, x_j)\, k(f_i, f_j)$ penalizes label disagreement between pixels that are close in position *and* color. In the fully connected Gaussian form (Krähenbühl & Koltun 2011), the kernel is a sum of an appearance term (bilateral: position + RGB) and a smoothness term (position only):

$$k(f_i, f_j) = w^{(1)} \exp\!\Big(-\tfrac{\|p_i - p_j\|^2}{2\theta_\alpha^2} - \tfrac{\|I_i - I_j\|^2}{2\theta_\beta^2}\Big) + \; w^{(2)} \exp\!\Big(-\tfrac{\|p_i - p_j\|^2}{2\theta_\gamma^2}\Big).$$

The first kernel is the edge-preserver: where pixel colors differ sharply, $\exp(-\|I_i - I_j\|^2/2\theta_\beta^2) \to 0$, so no smoothing penalty is applied and the boundary survives — boundaries snap to high-frequency color edges. The second kernel is a speckle filter that flips isolated mislabelled pixels. As $\theta_\beta \to \infty$ the appearance kernel loses all color sensitivity and degrades to pure spatial smoothing — the classic "bleeding edges" failure. Q15's answer is now structural: the CRF is a spatial-consistency post-processor, not a feature generator, trainer, or clustering algorithm. And its blind spot is worth stating: it cannot recover a class the unary potential missed entirely — it sharpens roughly right answers, it does not invent them.

Three revision questions the student implicitly asked. These are what the session's pacing commands and follow-up questions meant in exercise form; they map to the Part V practice sets.

1. *(From the metric exchanges)* Given unit-L_2 vectors, prove that squared Euclidean distance and cosine similarity differ only by an affine monotone transformation, and explain why this justifies using a k-NN monitor interchangeably with either metric. → Ch. 3 exercises, metric and geometry set.
2. *(From the clustering exchanges)* On the "two moons" dataset, predict and justify the failure of k-means, and describe how DBSCAN with a single global ε fails when cluster

densities differ. → Ch. 3 exercises, clustering set (and Ch. 17's exemplar answer on DBSCAN).

3. *(From DeepCluster/STEGO)* Explain the representational-collapse risk in DeepCluster and identify the two mechanisms (empty-cluster re-seeding, loss reweighting) that prevent it; then explain why STEGO's correspondence bias b is essential and what happens at $b \to 1$. → Ch. 3 exercises, hybrid-methods set.

E.2 Lecture 03: autoencoders — bottlenecks, the ELBO, reparameterization, MAE

Companion chapters: Chapter 4 (autoencoders and variational autoencoders) and Chapter 5 (masked autoencoders). The session taught the 39-slide deck in five blocks: deterministic autoencoders and the bottleneck (slides 1–9); the VAE formulation and the ELBO (slides 10–16); reparameterization and VAE failure modes (slides 17–22); the MAE asymmetric architecture (slides 23–31); and MAE evaluation ablations (slides 32–39), followed by a condensed MAE revision sheet the student requested by name.

Student. An autoencoder sets the target equal to the input. What forces it to learn anything useful rather than copy — and why does the lecture insist that a one-dimensional bottleneck is not enough?

Mentor. Copy a thousand-page book with an unlimited photocopier and you finish the task without reading a sentence; that is an unconstrained autoencoder. Give a student ten pages of summary from which a classmate must rebuild the book, and the writing of those ten pages *is* representation learning: recurring themes are kept, typos and margins are discarded (Vincent et al., 2008). The three dials that prevent the photocopier are (i) the latent dimension, $\dim \boldsymbol{z} \ll \dim \boldsymbol{x}$; (ii) encoder/decoder capacity; (iii) regularization (weight decay, dropout, noise injection).

The information bottleneck view formalizes this: with $I(X;T) = H(X) - H(X \mid T)$ the mutual information between input and representation, a supervised bottleneck trades off $I(X;T)$ against $I(T;Y)$ for the task Y (Tishby et al., 2000). An autoencoder has no Y; the dataset itself is the target, so the bottleneck should retain only what is *explainable across the dataset* — patterns that recur and are therefore compressible — and discard instance-specific noise, which is unpredictable and therefore expensive to store. Note that "noise" here is capacity-relative: with enough capacity the model treats sensor noise as structure worth reconstructing.

The lecture's one-dimensional counterexample completes the argument. Even with $\dim \boldsymbol{z} = 1$, a high-capacity pair of networks can implement a *codebook*: map the chocolate lab to $z = 1.0$, the black lab to $z = 2.0$, the pomeranian to $z = 3.0$, and have the decoder memorize the three lookup rules. Reconstruction is perfect and nothing about snouts or fur was learned. Two limit behaviors settle the theory: with linear encoder and decoder, minimizing $\mathcal{L} = \|\boldsymbol{x} - W_d W_e \boldsymbol{x}\|_2^2$ over rank-d products recovers, by the Eckart–Young theorem, the span of the top principal components — a linear autoencoder *is* PCA (Jolliffe, 2002). With high capacity and no constraint, the global minimum is the identity function and the mutual information $I(X;Z) = H(X)$: no compression, no generalization. The slide's exam question follows directly: a small-dataset run with near-zero training loss and poor validation reconstruction is memorizing; the remedies are stronger bottleneck pressure, regularization, input corruption, or augmentation — never more capacity.

Student. Why did the lecture bring up stacked autoencoders at all, if modern reconstruction training is just $\mathcal{L} = \|\boldsymbol{x} - \hat{\boldsymbol{x}}\|_2$?

Mentor. Optimization history. Before normalization layers and careful initialization, more than about three layers could not be trained end-to-end: gradients pass through saturating sigmoid/tanh derivatives (bounded by 0.25 each), so the partial derivative of the loss with respect to early encoder weights decays exponentially in depth. The 2006-era workaround (Vincent et al., 2010) is greedy layer-wise pretraining: train a shallow autoencoder $\boldsymbol{x} \to \boldsymbol{z}_1 \to \hat{\boldsymbol{x}}$; freeze the encoder; treat its codes as raw data and train $\boldsymbol{z}_1 \to \boldsymbol{z}_2 \to \hat{\boldsymbol{z}}_1$; repeat to layer N; then unfreeze and fine-tune end-to-end. Every step keeps the gradient path shallow, so no gradient ever vanishes through the full depth during initialization. Batch Normalization standardizes each layer's pre-activations per mini-batch, $\hat{a} = (a - \mu_B)/\sqrt{\sigma_B^2 + \varepsilon}$ (Ioffe and Szegedy, 2015), bounding activations and keeping gradients alive, which makes the whole stacking ritual unnecessary — though, and this is the subtle point, optimization stability does not change the representation problem above: a deep normalized autoencoder with too much capacity still memorizes.

Student. Deterministic autoencoders apparently cannot generate. Walk me through exactly how the variational autoencoder fixes that, starting from why the naive objective is intractable — including the Jensen step, which the slide just applies.

Mentor. Why generation fails first: a deterministic encoder pins each image to an isolated coordinate in $\boldsymbol{z}$-space, and the decoder is trained only at those points. Sampling a fresh $\boldsymbol{z}$ from anywhere else lands in a gap, and the decoder returns nonsense — this is quiz Q16's answer in the session. The VAE's fix (Kingma and Welling, 2014; Rezende et al., 2014) is probabilistic: let $\boldsymbol{z}$ be a random variable with prior $p(\boldsymbol{z}) = \mathcal{N}(0, I)$, encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \operatorname{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\boldsymbol{x})))$, and decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})$. The quantity of interest is the marginal likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})\, p(\boldsymbol{z})\, d\boldsymbol{z}$, which is intractable: it integrates over all latent configurations, almost all of which give negligible likelihood. So optimize a bound instead. Two facts carry the derivation. Jensen's inequality: for convex h, $\mathbb{E}[h(X)] \geq h(\mathbb{E}[X])$; since log is concave ($d^2 \log x / dx^2 = -1/x^2 < 0$), $\mathbb{E}[\log X] \leq \log \mathbb{E}[X]$. Now introduce an arbitrary proposal $q(\boldsymbol{z}) > 0$ and swap log and integral:

$$\begin{aligned} \log p(\boldsymbol{x}) &= \log \int p(\boldsymbol{z})\, p(\boldsymbol{x} \mid \boldsymbol{z})\, d\boldsymbol{z} = \log \int q(\boldsymbol{z})\, \frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\, p(\boldsymbol{x} \mid \boldsymbol{z})\, d\boldsymbol{z} = \log \mathbb{E}_q\Big[\frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\, p(\boldsymbol{x} \mid \boldsymbol{z})\Big] \\ &\geq \mathbb{E}_q\Big[\log \frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\Big] + \mathbb{E}_q[\log p(\boldsymbol{x} \mid \boldsymbol{z})] = -D_{\mathrm{KL}}(q(\boldsymbol{z}) \,\|\, p(\boldsymbol{z})) + \mathbb{E}_q[\log p(\boldsymbol{x} \mid \boldsymbol{z})], \end{aligned} \tag{E.1}$$

where the last step uses $D_{\mathrm{KL}}(q\|p) = \mathbb{E}_q[\log q - \log p] = -\mathbb{E}_q[\log(p/q)]$. This is the evidence lower bound (ELBO). Setting $q(\boldsymbol{z}) = q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ links the bound to the encoder, and the training loss is the negative ELBO:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}) = D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p(\boldsymbol{z})) - \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})].$$

Read it: the KL term presses every input's posterior cloud toward the standard Gaussian, so the clouds overlap and no gaps survive — answering Q4 (the KL term regularizes the approximate posterior toward the prior) and Q9 (a VAE differs from a plain AE by imposing a structured latent distribution, so sampling $\boldsymbol{z} \sim p(\boldsymbol{z})$ and decoding generates new data). One notational detail the session caught on the slide: the bullet points swap $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ between encoders and decoders; follow the equation's convention, $\boldsymbol{\phi}$ for the encoder, $\boldsymbol{\theta}$ for the decoder.

Student. The second ELBO term $\mathbb{E}_q[\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})]$ — why does the slide claim it is "just" the L2 reconstruction loss?

Mentor. Assume a Gaussian observation model: the decoder outputs the mean $G_{\boldsymbol{\theta}}(\boldsymbol{z})$ of a D-dimensional isotropic Gaussian with variance η,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) = \frac{1}{(2\pi\eta)^{D/2}} \exp\Big(-\frac{1}{2\eta}\,\|\boldsymbol{x} - G_{\boldsymbol{\theta}}(\boldsymbol{z})\|^2\Big).$$

Splitting the logarithm,

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z}) = -\frac{D}{2}\log(2\pi\eta) - \frac{1}{2\eta}\,\|\boldsymbol{x} - G_{\boldsymbol{\theta}}(\boldsymbol{z})\|^2,$$

and the first term does not depend on $\boldsymbol{\theta}$, so maximizing the expected log-likelihood is equivalent, up to a constant and a global scale $1/(2\eta)$, to minimizing mean squared error. The entire deterministic autoencoder fell out of the probabilistic model as a special case — a fact the lecture uses twice. First, η quietly weights the two ELBO terms: small assumed noise prioritizes reconstruction over the KL regularizer. Second, this Gaussian/L2 equivalence is the origin of the famous VAE blur (Q15): the expected squared error is minimized by the conditional mean of all plausible high-frequency details, so sharp alternatives get averaged into mush. It is also the reason a pretrained VAE remains embedded in modern systems as a latent-space compressor for latent diffusion models (Rombach et al., 2022), where the blur of a first-stage compressor matters less than its regularized geometry.

Student. The ELBO requires sampling $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$ inside the network. Sampling has no gradient — the slide draws a red box around it — so how does backpropagation through a VAE work at all?

Mentor. Push the randomness out of the graph. We need $\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\boldsymbol{f}(\boldsymbol{z})]$. Writing the expectation as an integral and pushing the gradient inside with the Leibniz rule leaves $\nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})\, \boldsymbol{f}(\boldsymbol{z})$ under the integral; the log-derivative (score-function) trick rewrites this as $\mathbb{E}_{q_{\boldsymbol{\phi}}}[\boldsymbol{f}(\boldsymbol{z})\, \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})]$, which is differentiable but has extremely high variance — it scales with the magnitude of $\boldsymbol{f}$ itself. The reparameterization trick (Kingma and Welling, 2014; Doersch, 2016) instead declares $\boldsymbol{z} = g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \boldsymbol{x}) = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}) \odot \boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$ and $\odot$ the Hadamard product. The law of the unconscious statistician then moves the expectation off the parameter-dependent distribution:

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\boldsymbol{f}(\boldsymbol{z})] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{f}(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \boldsymbol{x}))],$$

and since $p(\boldsymbol{\epsilon})$ has no parameters, the gradient slides inside directly:

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{f}(g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \boldsymbol{x}))] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{z}} \boldsymbol{f} \cdot \nabla_{\boldsymbol{\phi}} g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \boldsymbol{x})], \qquad \nabla_{\boldsymbol{\phi}} g_{\boldsymbol{\phi}} = \nabla_{\boldsymbol{\phi}} \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}) + \boldsymbol{\epsilon} \odot \nabla_{\boldsymbol{\phi}} \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}).$$

The variance of the sampling is unchanged — $\boldsymbol{z}$ is still distributed as $\mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2))$; only the computational routing changed — but the gradient now follows $\boldsymbol{f}$'s own curvature and is low-variance (Q8's answer in one line: reparameterization lets gradients flow through the stochastic sampling operation).

Two practical corollaries the session worked through. First, the Gaussian–Gaussian KL term needs no sampling at all; $D_{\mathrm{KL}}(\mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2)) \| \mathcal{N}(0, I))$ has the closed form

$$\frac{1}{2} \sum_{j=1}^{J} (\mu_j^2 + \sigma_j^2 - 1 - \log \sigma_j^2),$$

so only the reconstruction expectation is Monte-Carlo'd — with a single sample $L = 1$ per step, which minibatching makes stable enough. Second, the encoder should output $\log \sigma_j^2$, not σ_j^2: an unconstrained real-valued output for the log-variance never needs positivity constraints. The failure mode to know cold (Q14): *posterior collapse*, where a too-powerful decoder reconstructs from its own context, ignores $\boldsymbol{z}$, and the encoder minimizes the KL term exactly by outputting $\boldsymbol{\mu} = \mathbf{0}$, $\boldsymbol{\sigma}^2 = \mathbf{1}$ for every input — the code carries nothing. Countermeasures: anneal the KL weight from 0 to 1; enforce a free-bits minimum per dimension; or weaken the decoder.

Student. Masked autoencoders look like the same story — encode, decode, reconstruct — yet the lecture says the design is "asymmetric" and masks 75% of the image, where BERT masks 15%. Why these numbers, and why do the mask tokens never enter the encoder?

Mentor. Two answers: a statistical one and a computational one (He et al., 2022). Statistically, images are spatially redundant while language is information-dense. Mask one patch in seven and the encoder reconstructs the rest by local interpolation — a sky patch is surrounded by sky — and learns nothing about objects. At $\rho = 75\%$–80% masking, nearest visible evidence is far away, so reconstruction requires reasoning about global layout: this is Q18 (high masking forces broader contextual inference). The masking ratio ρ borders two degenerate regimes: as $\rho \to 0$ the task collapses to interpolation; as $\rho \to 1$ (95%) the visible evidence cannot support the scene at all.

Computationally, the trick is where the work goes. With N patches, the encoder $f_{\boldsymbol{\theta}}$ receives only the visible set V with $|V| = (1-\rho)N$, embedding each as $\boldsymbol{z}_i = f_{\boldsymbol{\theta}}(\mathrm{LinearProj}(\boldsymbol{x}_p^{(i)}) + e_i)$ for positional embedding e_i. The decoder sees the reassembled sequence

$$y_i = \begin{cases} \boldsymbol{z}_i + e'_i & \text{if } i \in V, \\ \boldsymbol{E}_{\mathrm{mask}} + e'_i & \text{if } i \in M, \end{cases} \qquad M = \{1, \dots, N\} \setminus V,$$

and the loss touches only the masked patches:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \frac{1}{|M|} \sum_{i \in M} \left\| \hat{\boldsymbol{x}}_p^{(i)} - \boldsymbol{x}_p^{(i)} \right\|_2^2.$$

Excluding visible patches from the loss removes the copying shortcut; keeping the shared learnable token $\boldsymbol{E}_{\mathrm{mask}}$ out of the encoder cuts the encoder's sequence fourfold, and since attention costs $O(N^2)$ that is a $16\times$ saving in encoder attention (He et al., 2022) — the reason a ViT-Huge fits a normal budget (Q5 and Q17). Positional embeddings are added on the decoder side to both token kinds, since otherwise the decoder could not know where each patch belongs.

Anticipating Chapter 5's ablation curve that the student asked the mentor to summarize: linear-probe accuracy is exquisitely sensitive to ρ, peaking sharply near 75% and collapsing to about 54.6% at 10% masking, while full fine-tuning is robust, peaking near 50%–60% masking around 85% and staying flat to 80%. The diagnosis the session gave: low masking minimizes pretraining loss via local shortcuts, so the frozen features are not linearly separable; fine-tuning can undo the damage, which is precisely why the linear probe is the more honest probe of the pretext task's difficulty.

Three revision questions the student implicitly asked.

1. *(From the bottleneck exchange)* Show that a linear encoder–decoder pair minimizing $\|\boldsymbol{x} - W_d W_e \boldsymbol{x}\|_2^2$ spans the principal subspace of the data, and explain where the argument fails for a high-capacity nonlinear pair on a finite dataset. → Ch. 4 exercises, bottleneck set.
2. *(From the ELBO exchange with the follow-up in Lecture 05)* Derive the ELBO from $\log p(\boldsymbol{x}) = \log \int q(\boldsymbol{z}) \, \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} d\boldsymbol{z}$ without skipping the Jensen step, and state precisely which

expectation must be Monte-Carlo approximated in practice and which has a closed form. → Ch. 4 exercises, variational set.

3. *(From the MAE exchanges)* Quantify the encoder compute saving from excluding mask tokens at masking ratio ρ, and predict the shape of linear-probe and fine-tune accuracy curves as a function of ρ. → Ch. 5 exercises, masked-modeling set.

E.3 Lecture 04: generative models — KDE, normalizing flows, GANs

Companion chapter: Chapter 6 (generative models). The session taught the deck in five blocks over slides 1–29: generative foundations and KDE; flow theory and change of variables; Real NVP and GLOW; conditional flows and a flow summary; GANs and their failure modes. The StyleGAN and Pix2Pix tail of the deck (slides 30–43) was never reached in this session — a gap the mentor flagged at the time; Chapter 6 covers those slides from the deck directly.

Student. Seeing "generative models" in a self-supervised course confused me — where is the supervision? And why does the lecture start with kernel density estimation, which predates deep learning by decades?

Mentor. The supervision is the data itself: modeling $p(\boldsymbol{x})$ needs only $\boldsymbol{x}$, never $(\boldsymbol{x}, y)$ pairs. That makes generative modeling a pretext task — and a demanding one, since generating a realistic face forces the model to internalize that faces have two symmetric eyes above a nose. The features learned along the way transfer downstream, exactly as in MAE. Note also that the task really has two halves — *density estimation* (evaluate $q(\boldsymbol{x}) \approx p(\boldsymbol{x})$) and *synthesis* (draw $\boldsymbol{x} \sim q(\boldsymbol{x})$) — and, as the taxonomy on the deck's final slide makes explicit, some families do one well but not the other: GANs sample beautifully but cannot evaluate a likelihood; autoregressive models evaluate but sample slowly.
KDE is the instructive failure (Jolliffe, 2002). Place a normalized bump on each of n observations,

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{x - x_i}{h}\Big) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h\sqrt{2\pi}} \exp\Big(-\frac{(x - x_i)^2}{2h^2}\Big) \quad \text{(Gaussian case)},$$

where h is the bandwidth. Two bookkeeping details matter for the exam: the prefactor $1/h$ is what keeps each scaled kernel integrating to one — wider must mean shorter — and the number of effective parameters grows with n, so evaluating a single density costs $O(n)$. The bandwidth is the regularizer: the session's concrete probe was a 1D dataset $\{2.0, 2.1, 2.2, 8.0, 8.1, 8.2\}$, two tight clusters separated by a gap of about 5.8. With $h = 10$, each Gaussian's decay $\exp(-(x - x_i)^2/2h^2)$ is still near 1 at the midpoint $x = 5$, both clusters' density waves overlap there, and the model assigns high probability to a region with no data — the correct answer to quiz Q1 (the density becomes smoother and may peak in the gap). Sampling from a KDE is revealing: pick a training point uniformly, add $\boldsymbol{\epsilon} \sim \mathcal{N}(0, h^2 I)$, done. The lecture's 8×8 digit demo shows the consequence: "new" digits are just noisy copies of training digits. In 64 dimensions the data mass lives on isolated islands; small h memorizes, large h smears structure into noise. For a face image with $256 \times 256 \times 3 = 196\,608$ dimensions, the volume argument is fatal — Q2's answer: KDE scales poorly in high-dimensional spaces. Hence learned, parametric density models.

Student. Normalizing flows claim *exact* likelihoods where VAEs only get a bound. Where does the Jacobian come from — and why must architecture obey it?

Mentor. Conservation of probability mass. The slide's sand picture is the right one: if p_X is a pile of sand and $g : \mathbb{R}^D \to \mathbb{R}^D$ a bijective, differentiable map moving each grain, the mass in any region $\mathcal{R}_X$ equals the mass in its image:

$$\int_{\mathcal{R}_X} p_X(\boldsymbol{x})\, d\boldsymbol{x} = \int_{g(\mathcal{R}_X)} p_Z(\boldsymbol{z})\, d\boldsymbol{z}.$$

Substituting $\boldsymbol{z} = g(\boldsymbol{x})$ on the right rescales the volume element by $d\boldsymbol{z} = |\det J_g(\boldsymbol{x})|\, d\boldsymbol{x}$, with $J_g(\boldsymbol{x}) = \partial g(\boldsymbol{x})/\partial \boldsymbol{x}^\top$ the Jacobian matrix. Since $\mathcal{R}_X$ was arbitrary, the integrands agree pointwise, and taking logs gives the flow objective:

$$p_X(\boldsymbol{x}) = p_Z\big(g(\boldsymbol{x})\big)\, \big|\det J_g(\boldsymbol{x})\big|, \qquad \log p_X(\boldsymbol{x}) = \log p_Z\big(g(\boldsymbol{x})\big) + \log\big|\det J_g(\boldsymbol{x})\big|.$$

The map says *where* the sand goes; the Jacobian determinant says how much it is locally squeezed or stretched. Two contrasts with the VAE follow immediately (this was quiz Q3): the mapping is deterministic — no reparameterized sampling inside — and the bijection leaves no holes in latent space, so $g^{-1}(\boldsymbol{z})$ with $\boldsymbol{z} \sim \mathcal{N}(0, I)$ generates without the empty-region nonsense of a plain autoencoder's decoder.
Depth is free, which is the elegance: build the generator from simple invertible layers, $\boldsymbol{x} = f_T \circ \cdots \circ f_1(\boldsymbol{z})$, with $\boldsymbol{z}_0 = \boldsymbol{z}$, $\boldsymbol{z}_t = f_t(\boldsymbol{z}_{t-1})$, $\boldsymbol{z}_T = \boldsymbol{x}$ and a simple prior p_{Z_0}. The product rule for determinants turns the composed log-Jacobian into a sum,

$$\log p_X(\boldsymbol{x}) = \log p_{Z_0}(\boldsymbol{z}_0) - \sum_{t=1}^{T} \log\big|\det J_{f_t}(\boldsymbol{z}_{t-1})\big|.$$

(Watch the index direction: generation applies f_1 first; likelihood evaluation inverts the chain and applies f_T^{-1} first — the session flagged this inversion as a frequent exam slip.) The price is architectural: for general layers a $D \times D$ determinant costs $O(D^3)$, impossible at $D = 196\,608$ pixels. So every flow layer must be engineered for a cheap determinant — ideally a triangular Jacobian, whose determinant is the product of its diagonal entries, an $O(D)$ operation (Dinh et al., 2015). Q4's answer is now one line: the family engineered to compute exact likelihoods while remaining generative is the normalizing flow; and Q6's — medical imaging needing realistic samples, exact likelihoods for anomaly detection, and an invertible latent code — selects flows over GANs, which cannot evaluate $p(\boldsymbol{x})$ at all.

Student. How exactly does an affine coupling layer keep both directions cheap — and what did GLOW add on top?

Mentor. Split the input and only transform half of it, conditioned on the other half (Dinh et al., 2017). Forward:

$$\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}, \qquad \boldsymbol{y}_{d+1:D} = \boldsymbol{x}_{d+1:D} \odot \exp\big(s(\boldsymbol{x}_{1:d})\big) + t(\boldsymbol{x}_{1:d}),$$

inverse:

$$\boldsymbol{x}_{1:d} = \boldsymbol{y}_{1:d}, \qquad \boldsymbol{x}_{d+1:D} = \big(\boldsymbol{y}_{d+1:D} - t(\boldsymbol{y}_{1:d})\big) \odot \exp\big(-s(\boldsymbol{y}_{1:d})\big).$$

The suitcase analogy from the session is precise: pack suitcase 1 ($\boldsymbol{y}_{1:d} = \boldsymbol{x}_{1:d}$) plainly; let a friend look inside and then apply an arbitrary squashing rule s, t to suitcase 2; unpacking suitcase 2 needs only a look at suitcase 1. The networks s and t may be arbitrarily deep and non-invertible — they are only ever evaluated forward. The exp guarantees positive scales so the divide-back never hits zero. The Jacobian is block lower-triangular by construction,

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^\top} = \begin{bmatrix} I_d & 0 \\ \partial \boldsymbol{y}_{d+1:D} / \partial \boldsymbol{x}_{1:d}^\top & \operatorname{diag}\big(\exp[s(\boldsymbol{x}_{1:d})]\big) \end{bmatrix}, \qquad \log|\det J| = \sum_{i=1}^{D-d} s(\boldsymbol{x}_{1:d})_i,$$

an $O(D)$ log-determinant from the diagonal product. That answers quiz Q5 structurally: if inverting a flow layer required numerical optimization, sampling $\boldsymbol{x} = f^{-1}(\boldsymbol{z})$ would be impractical; coupling layers dodge it because inversion is algebraic and closed-form. "Non-volume preserving" in Real NVP's name marks the departure from NICE's purely additive ($s \equiv 0$, $\det J = 1$) couplings: allowing local volume change is what buys expressivity. Real NVP additionally fights the dimension-preservation cost ($\boldsymbol{z}$ and $\boldsymbol{x}$ have equal dimension in any flow) with a multiscale architecture: squeeze spatial size into channels, then progressively factor half the channels straight to the Gaussian prior, summing their log-prior terms into the likelihood.

GLOW then replaced the rigid fixed permutations between blocks with a learnable invertible 1×1 convolution — per pixel, a channel matrix $W \in \mathbb{R}^{C \times C}$ — whose log-determinant would naively cost $O(C^3)$ (Kingma and Dhariwal, 2018). Parameterizing W directly in LU form, $W = PLU$ with L unit lower-triangular and U upper-triangular, makes $\log|\det W|$ the sum of the log-magnitudes of U's diagonal, $O(C)$, and turns channel mixing into a learnable, data-adaptive operation. And because flows maximize exact likelihood over the entire dataset, they are forced to cover every mode — the session's contrast with GAN mode dropping.

Student. GANs: how does the minimax game work, and where exactly does it break — this keeps coming back as the "mode collapse" question.

Mentor. Abandon the likelihood and replace the loss with an adversary (Goodfellow et al., 2014). The generator G maps $\boldsymbol{z} \sim p(\boldsymbol{z})$ to data space with an unconstrained, non-invertible network (no bijectivity, no dimension preservation — pooling and strided convolutions welcome). The discriminator $D(\cdot) \in [0,1]$ estimates "real", and the two play

$$\min_G \max_D \; V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\big[\log D(\boldsymbol{x})\big] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}\big[\log(1 - D(G(\boldsymbol{z})))\big].$$

With G fixed, the discriminator's half is ordinary binary cross-entropy: drive $D(\boldsymbol{x}) \to 1$ on data and $D(G(\boldsymbol{z})) \to 0$ on fakes. With D fixed, the first expectation does not involve G, so the generator works on the second term, pushing $D(G(\boldsymbol{z})) \to 1$. The practical wrinkle, worked out in the session, is saturation: early in training D rejects everything, $D(G(\boldsymbol{z})) \approx 0$, $\log(1 - D(G(\boldsymbol{z}))) \to \log 1 = 0$, and its gradient with respect to G's parameters flatlines — the forger gets no coaching. The standard fix is the non-saturating objective, training G to maximize $\mathbb{E}_{\boldsymbol{z}}[\log D(G(\boldsymbol{z}))]$ instead; equal optimum, strong early gradients.

Two failure modes for quiz Q8. *Mode collapse*: with no likelihood to account for, nothing charges the generator for ignoring part of the data distribution; mapping a broad latent region onto a few outputs that fool the current discriminator is a valid local optimum, and the deck's demos are the signature — diverse real modes, repetitive fake ones. *Instability*: the alternating updates hunt a saddle point of $V(D,G)$ rather than the minimum of a single loss, so a discriminator that outruns the generator saturates gradients, and one that falls behind gives uninformative ones. The taxonomy placement to carry forward: GAN is the implicit-density extreme — perfect synthesis machinery, no density evaluation — sitting opposite exact flows (great likelihood, architecturally straitjacketed), with VAEs in between (bounded likelihood, clean sampling).

Three revision questions the student implicitly asked.

1. *(From the KDE exchanges)* For the dataset $\{2.0, 2.1, 2.2, 8.0, 8.1, 8.2\}$, argue quantitatively why Gaussian KDE with $h = 10$ assigns substantial density near $x = 5$, and describe how sampling from a KDE produces "noisy training points" in high dimensions. → Ch. 6 exercises, density-estimation set.
2. *(From the flow exchanges)* Starting from probability-mass conservation on a region $\mathcal{R}_X$, derive $\log p_X(\boldsymbol{x}) = \log p_Z(g(\boldsymbol{x})) + \log|\det J_g(\boldsymbol{x})|$, then show that for an affine coupling layer the log-determinant reduces to the sum of the scale outputs, $\sum_i s(\boldsymbol{x}_{1:d})_i$. → Ch. 6 exercises, flow set.
3. *(From the GAN exchange)* Explain why the generator's gradient saturates under $\log(1 - D(G(\boldsymbol{z})))$ early in training, why maximizing $\log D(G(\boldsymbol{z}))$ avoids it, and why nothing in the minimax objective prevents mode collapse. → Ch. 6 exercises, adversarial-training set.

E.4 Lecture 05: denoising generative models — DDPM algebra, rectified flows, latent diffusion, DIFT

Companion chapter: Chapter 7 (denoising and diffusion). This session had a different teaching contract from the others — flowing notebook prose per slide rather than checklists — and covered slides 1–61 of the 69-slide deck, followed by the richest exchange of the whole series: a single student question about the tutorial quiz deck that produced six full derivations of the exam-trap concepts. Those six are reproduced in the second half of this section.

Student. Rectified flow looks like diffusion from another angle: slide 19 writes $x_t = (1-t)x + tz$ and calls the network a "velocity" regressor. Why is a straight path better, and how do I train and sample it?

Mentor. Draw the straight line between clean data x (at $t = 0$) and noise z (at $t = 1$): $x_t = (1-t)x + tz$. The time derivative is constant and equal to the endpoint difference,

$$\frac{dx_t}{dt} = z - x,$$

so exactly one network output — the velocity $v = z - x$ — determines the whole path. Training is plain least-squares regression toward that target (Lipman et al., 2023; Liu et al., 2023):

$$\mathcal{L}_{\mathrm{RF}}(\boldsymbol{\theta}) = \mathbb{E}_{x,\,z,\,t}\big[\|f_{\boldsymbol{\theta}}(x_t, t) - (z - x)\|_2^2\big].$$

To generate, integrate the learned vector field backward from $t = 1$ to $t = 0$ with an Euler solver,

$$x_{\mathrm{new}} = x_{\mathrm{old}} - \frac{f_{\boldsymbol{\theta}}(x_t, t)}{T},$$

stepping *down* in t. The subtraction sign is the quiz's trap (Q04): the network predicts the forward noising direction — from data toward noise — so sampling must move against it; add instead of subtract and every step deepens the noise rather than removing it.

Why "rectified"? Randomly paired (x, z) endpoints make ideal trajectories that cross, and the least-squares fit of several crossing targets is their average — a curved field. Curvature forces small Euler steps. The reflow trick: regenerate synthetic pairs (x_{gen}, z) with the trained first-stage model, then retrain on those; the new pairings uncross the trajectories, the learned field straightens, and a perfectly straight trajectory admits an exact one-step Euler solve ($\Delta t = -1$), which is what enables distilled samplers running in one to four steps. The theory anchor for why straight is good: a straight line is the shortest path between its endpoints, and each reflow round never increases the transport cost between the two distributions (it does not, in general, reach the optimal transport plan).

Student. The CNFs section says rectified flows are actually continuous normalizing flows and evaluates an exact likelihood with a trace. How does that work, and why the "Hutchinson trick"?

Mentor. A CNF replaces the discrete invertible layers of Lecture 04's flows with a continuous vector field (Chen et al., 2018; Grathwohl et al., 2019): the bijection F is defined by integrating an ODE, $d\boldsymbol{z}_t/dt = f(\boldsymbol{z}_t, t)$, so $\boldsymbol{x} = F(\boldsymbol{z}_0) = \boldsymbol{z}_T = \boldsymbol{z}_0 + \int_0^T f(\boldsymbol{z}_t, t)\, dt$. Density tracking no longer multiplies per-layer Jacobian determinants; instead the instantaneous change of variables says the log-density's rate of change is the negative divergence of the field,

$$\frac{\partial \log p(\boldsymbol{z}_t)}{\partial t} = -\operatorname{div} f(\boldsymbol{z}_t, t) = -\operatorname{tr}\Big(\frac{\partial f(\boldsymbol{z}_t, t)}{\partial \boldsymbol{z}_t}\Big), \qquad \log p(\boldsymbol{x}) = \log p(\boldsymbol{z}_0) - \int_0^T \operatorname{tr}\Big(\frac{\partial f}{\partial \boldsymbol{z}_t}\Big) dt.$$

Compare with Lecture 04: the discrete flow priced each layer's Jacobian determinant at $O(D^3)$ unless the layer was hand-built to be triangular. Here the integrand is a *trace*, and traces admit a stochastic estimate (Hutchinson, 1990): for any square matrix A, draw $\boldsymbol{\epsilon}$ with zero-mean, identity-covariance entries; then $\mathbb{E}_{\boldsymbol{\epsilon}}[\boldsymbol{\epsilon}^\top A \boldsymbol{\epsilon}] = \operatorname{tr}(A)$. And $\boldsymbol{\epsilon}^\top \frac{\partial f}{\partial \boldsymbol{z}}$ is a vector–Jacobian product, which reverse-mode autodiff produces in a single backward pass — one pass instead of the D passes needed to build the full Jacobian, and no $O(D^3)$ determinant. That is the complete chain from ODE to tractable likelihood: ODE field → divergence → trace → Hutchinson → VJP.

Student. Latent diffusion: if diffusion models work in pixel space, why move into a latent space — and doesn't that drag back exactly the blur problems VAEs had?

Mentor. Compute geography. A 1024×1024 image is about 3.1×10^6 dimensions, and a pixel-space diffusion model must denoise all of them at every step — the session noted the model then spends most of its capacity on high-frequency texture that is perceptually redundant, like recording the position of every grain of mortar in a brick wall when the useful concept is "a red brick wall." Latent diffusion (Rombach et al., 2022) splits the problem in two. Stage 1 (perceptual compression): train an autoencoder with encoder E downsampling by a factor f (typically 8 or 4, so $512 \times 512 \to 64 \times 64$) and decoder D; crucially this compressor is trained as a VAE–GAN hybrid, $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{recon}} + \beta\,\mathcal{L}_{\text{KL}} + \mathcal{L}_{\text{adv}}$, where the KL term keeps the latent space dense and gap-free — exactly the "no empty regions" property Lecture 03 said sampling needs — and the adversarial term restores the high-frequency sharpness that the Gaussian-likelihood blur analysis in Lecture 03 predicted a plain VAE would lose. Stage 2: freeze the autoencoder and train the diffusion model entirely on latents, $\boldsymbol{z}_t = (1-t)\boldsymbol{z} + t\boldsymbol{\epsilon}$ with $\boldsymbol{z} = E(\boldsymbol{x})$, ignoring pixels. Stage 3: sample $\boldsymbol{z}_T \sim \mathcal{N}(0, I)$, denoise iteratively in latent space, and decode once, $\boldsymbol{x}_{\text{gen}} = D(\boldsymbol{z}_0)$. The blur question dissolves because the generative model never optimizes a per-pixel Gaussian likelihood: it models structure in the compressed space and lets the adversarially trained decoder render texture.

Student. DIFT claims a trained diffusion U-Net yields free per-pixel semantic correspondences. How do you extract them, and why does the timestep choice matter?

Mentor. The extraction pipeline is four lines long (Tang et al., 2023). Pick an intermediate timestep t, typically in $[100, 500]$; noise the clean image to x_t; run it through the frozen pretrained U-Net and grab intermediate feature maps from the bottleneck and decoder blocks; bilinearly upsample them to $H \times W$ so each pixel p owns a feature vector $f_p \in \mathbb{R}^C$. A match between pixel p_A in image A and p_B in image B is then just the usual cosine rule — the Lecture 02 metric, $\text{sim}(p_A, p_B) = f_{p_A} \cdot f_{p_B} / (\|f_{p_A}\|\,\|f_{p_B}\|)$ — maximized over pixels of B. The deck's three headline demos show the features transcend appearance: cross-instance (bill of one duck species to another), cross-category (polar bear's ear to brown bear's ear), and cross-domain (real guitar to a cubist painted guitar). The intuition for *why* denoisers encode semantics is coarse-to-fine: at $t \approx 500$ the image is almost pure static and the model must first commit to a high-level guess — a cat-shaped silhouette; at $t \approx 250$ it localizes parts; at small t it sharpens fur and whiskers. So the network's interior layers hold a semantic hierarchy as a byproduct of the objective. The timestep question gets its own derivation below (exchange on granularity).

Student. Now the quiz deck for this tutorial, please — there are concepts on it the lecture slides never taught. Walk me through the extra ones.

Mentor. Six concepts, in exam order. First, *structured noising* (Q01, Q15). DDPM's forward chain is fixed and non-learned (Ho et al., 2020): $q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}\, x_{t-1}, \beta_t I)$ with schedule $\beta_t \in (0,1)$. Because compositions of Gaussians remain Gaussian, training never simulates the chain: with $\alpha_t = 1-\beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, one can jump directly,

$$q(x_t \mid x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}\, x_0, (1-\bar{\alpha}_t) I), \qquad x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, I).$$

The schedule is variance-preserving: $\mathrm{Var}(x_t) = \bar{\alpha}_t \mathrm{Var}(x_0) + (1-\bar{\alpha}_t)\mathrm{Var}(\boldsymbol{\epsilon}) = \bar{\alpha}_t I + (1-\bar{\alpha}_t) I = I$ for normalized data. As $t \to T$, $\bar{\alpha}_t \to 0$: $t \approx 0$ carries maximal information about x_0 and $t \approx T$ is pure noise, tracked by the signal-to-noise ratio $\mathrm{SNR}(t) = \bar{\alpha}_t/(1-\bar{\alpha}_t)$.
Second, *parameterization equivalences and error amplification* (Q02, Q03). Since x_t is a fixed linear combination of x_0 and $\boldsymbol{\epsilon}$, the network may predict noise $\boldsymbol{\epsilon}$, clean image x_0, or velocity v interchangeably at zero theoretical cost — any two of the three determine the third. Solving the marginal for x_0 gives the conversion,

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\, \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(x_t, t)}{\sqrt{\bar{\alpha}_t}},$$

and the trap lives in the denominator: at large t, $\bar{\alpha}_t \to 0$, so the factor $1/\sqrt{\bar{\alpha}_t}$ explodes and small noise-prediction errors become huge x_0-space errors — unstable gradients precisely where generation begins. That is why $\boldsymbol{\epsilon}$- or v-prediction is numerically preferred even though the parameterizations are algebraically equivalent.
Third, *Tweedie's identity* (Q13; slide 40, "denoising and the score are the same thing"). Let $y \mid x \sim \mathcal{N}(y; x, \sigma^2 I)$. Tweedie expresses the posterior mean through the marginal density of the observation,

$$\mathbb{E}[x \mid y] = y + \sigma^2 \nabla_y \log p(y).$$

With $x_t = x_0 + \sigma\varepsilon$ and the optimal denoiser $D(x_t) = \mathbb{E}[x_0 \mid x_t]$ substituted, $D(x_t) = x_t + \sigma^2\, \nabla_{x_t} \log p(x_t)$, and hence

$$\nabla_{x_t} \log p(x_t) = \frac{D(x_t) - x_t}{\sigma^2}.$$

The vector from the noisy point to its clean reconstruction *is*, up to the $1/\sigma^2$ factor, the score of the *noisy* data distribution — the direction of steepest density increase. (For DDPM's scaled $x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sigma_t \varepsilon$ the same argument gives $(\sqrt{\bar{\alpha}_t}\, D(x_t) - x_t)/\sigma_t^2$; see Eq. (7.7).) Training a Gaussian denoiser is therefore mathematically equivalent to score estimation (Hyvärinen, 2005); denoising is not a heuristic but an estimate of $\nabla \log p$.
Fourth, *the ELBO versus true-likelihood trap* (Q11) — sharpening the Lecture 03 derivation. The exact identity is

$$\log p(\boldsymbol{x}) = \mathrm{ELBO} + D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})),$$

where $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \boldsymbol{x})$ is the true posterior under the decoder. Exam scenario: a team updates *only* the encoder and reports the ELBO rose. Did $\log p(\boldsymbol{x})$ rise? No. The true likelihood depends on the prior and decoder parameters $\boldsymbol{\theta}$ only; with $\boldsymbol{\theta}$ fixed, $\log p(\boldsymbol{x})$ on the left-hand side is constant, and the encoder merely shrank the posterior-approximation KL gap, moving the bound up toward an unmoved likelihood. The bound improved; the model's

Three revision questions the student implicitly asked.

1. *(From the rectified-flow exchange)* For $x_t = (1-t)x + tz$, derive the training loss and the Euler sampling update from $dx_t/dt = z - x$, justify the sign of the update, and prove a perfectly straight trajectory makes a one-step Euler solve exact. → Ch. 7 exercises, flow-matching set.
2. *(From the quiz-deck answers)* Starting from the closed-form marginal $q(x_t \mid x_0)$, verify variance preservation, derive the $\hat{x}_0$ conversion, and show how $1/\sqrt{\bar{\alpha}_t}$ amplifies estimation error at large t. → Ch. 7 exercises, diffusion-geometry set.
3. *(From Tweedie/ELBO answers)* Prove the score–denoiser identity $\nabla_{x_t} \log p(x_t) = (D(x_t) - x_t)/\sigma^2$, and write the counterexample showing that an encoder-only ELBO improvement need not change $\log p(\boldsymbol{x})$. → Ch. 7 exercises, score-matching set (and the Ch. 4 bound set).

E.5 Lecture 06: pretext tasks and augmentations — RotNet, PIRL, InfoNCE, SimCLR, MoCo, Barlow Twins

Companion chapters: Chapter 8 (pretext tasks and augmentations), Chapter 9 (contrastive learning), and Chapter 10 (non-contrastive methods). This 73-slide deck got the longest session of the series, taught in nine blocks; it also contains the student's only mid-course content request (a typed-with-elbows "discuss RotNet in detail"), which the mentor honored with an expanded treatment, reproduced below.

Student. The early pretext tasks all "invent a label from the data." How does predicting a rotation teach a network anything — and why exactly four rotations, not eight or thirty-six?

Mentor. RotNet trains an ordinary classifier to say which of four right-angle rotations — $0°, 90°, 180°, 270°$ — was applied to a natural image (Gidaris et al., 2018). The learning pressure is the *visual common-sense hypothesis*: you cannot tell that a bird is lying on its side unless you know what an upright bird looks like, so minimizing the four-way cross-entropy forces the network to internalize canonical object orientation — heads at the top, feet at the bottom, gravity's direction. The empirical vindication is the first-layer-filter comparison on slide 27: RotNet's conv1 filters are visually indistinguishable from supervised AlexNet's — the same clean bank of oriented edge detectors and color opponencies. When a self-supervised task is broken, that grid looks like static.
Why four orientations is an artifact-engineering answer, and it generalizes to the design rule for every handcrafted pretext task (Q4, Q6). Rotations by angles other than multiples of 90° require resampling: bilinear interpolation blurs the image and padding fills the corners, and those interpolation and border artifacts are low-level fingerprints that let the network read off the angle without ever learning objects — *transformation shortcut*. Right angles are exact pixel permutations, so the only signal left is semantic. Two angles ($0°/180°$) are too coarse to bear much signal; the coarse right-angle set is the sweet spot. The same "don't leave fingerprints" reasoning is why jigsaw (Noroozi and Favaro, 2016) leaves random gaps between its tiles — otherwise the network could solve the puzzle by matching edges and textures that continue across tile borders. And the acknowledged failure domain (Q5-radar): for symmetric or orientation-free imagery — flowers seen from above, aerial photos — the canonical-up assumption fails and RotNet's signal vanishes.

Student. Then the deck pivots: PIRL keeps the jigsaw machinery but refuses to predict the permutation. What changed conceptually — covariance versus invariance?

Mentor. Slide 38 poses it with two photos of the same bird, one rotated: does the feature serve you better by *discriminating the rotation* or by *recognizing that both are birds*? RotNet chose the first — its features must track the transform to predict it, i.e. they are *covariant*: $\boldsymbol{f}(t(\boldsymbol{x})) = t'(\boldsymbol{f}(\boldsymbol{x}))$ records the parameter t. Classification benchmarks reward the second — an *invariant* representation, $\boldsymbol{f}(t(\boldsymbol{x})) \approx \boldsymbol{f}(\boldsymbol{x})$, that deletes the nuisance transform entirely. And the trade-off is strict: a representation perfectly invariant to rotation can never again say which way is up. You must abandon the prediction objective to buy semantic utility. That is the Q3 answer in one sentence: RotNet features covary with the applied transform; PIRL features are invariant to it.
PIRL's construction (Misra and van der Maaten, 2020) keeps the transform and changes the game: shuffle a jigsaw view I^t of image I and require its embedding to match the cached embedding of the clean I, against a memory bank of other images' embeddings as negatives, via a noise-contrastive objective (τ-scaled softmax; the λ term blends the loss on clean-view and transformed-view representations). The memory bank is the enabling engineering detail here — it supplies thousands of negatives without a gigantic batch — and its cost, stale features, is precisely the problem MoCo will solve below.

Student. The contrastive block walks a "loss ladder": triplet loss, N-pair, NCE, InfoNCE. Reconstruct the ladder — what does each rung fix?

Mentor. The shared ancestor is the Siamese idea: same network twice, two views (Becker and Hinton, 1992; Bromley et al., 1993). Rung one, *triplet loss* (Schroff et al., 2015): anchor, positive, a single negative, with margin ϵ between the two distances. Its defect is exactly the one-negative limit — one distractor at a time is a weak contrastive signal. Rung two, *N-pair loss* (Sohn, 2016), swaps in $N-1$ simultaneous negatives and is where the ladder becomes a softmax over similarity scores:

$$\begin{aligned}\mathcal{L}_{\text{N-pair}} &= \log\Big(1+\textstyle\sum_{i=1}^{N-1}\exp\big(\boldsymbol{f}(\boldsymbol{x})^\top\boldsymbol{f}(\boldsymbol{x}_i^-)-\boldsymbol{f}(\boldsymbol{x})^\top\boldsymbol{f}(\boldsymbol{x}^+)\big)\Big)\\ &= -\log\frac{\exp\big(\boldsymbol{f}(\boldsymbol{x})^\top\boldsymbol{f}(\boldsymbol{x}^+)\big)}{\exp\big(\boldsymbol{f}(\boldsymbol{x})^\top\boldsymbol{f}(\boldsymbol{x}^+)\big)+\sum_{i=1}^{N-1}\exp\big(\boldsymbol{f}(\boldsymbol{x})^\top\boldsymbol{f}(\boldsymbol{x}_i^-)\big)}.\end{aligned}$$

(The exam likes the algebra: these two display lines are the same function; derive each from the other.) Rung three, *NCE* (Gutmann and Hyvärinen, 2010), reframes the softmax as density estimation: a binary classifier scoring the probability that the pair (I, I^t) came from data rather than noise. In self-supervised use there is no explicit noise distribution — the other images in the batch *play* the noise, which is the conceptual step to rung four, *InfoNCE* (van den Oord et al., 2018b; Poole et al., 2019): literally the categorical cross-entropy of retrieving the positive among K negatives over temperature-scaled cosine scores,

$$\ell = -\log\frac{\exp s(\boldsymbol{x},\boldsymbol{x}^+)}{\exp s(\boldsymbol{x},\boldsymbol{x}^+)+\sum_{k=1}^{K}\exp s(\boldsymbol{x},\boldsymbol{x}_k^-)}, \qquad s(\boldsymbol{x},\boldsymbol{y}) = \frac{\boldsymbol{f}(\boldsymbol{x})^\top\boldsymbol{f}(\boldsymbol{y})}{\tau\,\|\boldsymbol{f}(\boldsymbol{x})\|\,\|\boldsymbol{f}(\boldsymbol{y})\|}.$$

This is Q13 verbatim ("classifying which key is the positive among the negatives"), and it doubles as a lower bound on mutual information between the two views — the gradient therefore *is* a representation-learning gradient, not just a matching heuristic. Two sharp edges. The temperature τ (Q14): lowering it sharpens the softmax so that gradients concentrate on the hardest negatives; take $\tau \to 0$ and optimization is hostage to false negatives. Which is the second edge: without labels, a second cat in the negative pool is declared "not this cat" and actively pushed away — class collision, the systematic price of batch negatives.

Student. SimCLR then works with nothing but in-batch negatives, yet needs batches of 8192. What are its real moving parts, and why is the projection head thrown away?

Mentor. SimCLR (Chen et al., 2020a) is the minimal recipe: sample two independent augmentation stacks per image — crop/flip, color jitter, grayscale, blur — encode both through a shared backbone to representations $\boldsymbol{h}_i, \boldsymbol{h}_j$, push them through a small nonlinear MLP head $\boldsymbol{g}(\cdot)$, and apply InfoNCE (NT-Xent) on the projections $\boldsymbol{z}_i, \boldsymbol{z}_j$, with all $2N-2$ other views in the batch as negatives. Three design facts carried the session. First, the projection head as an *information shield*: the invariance objective wants color and crop information erased, but downstream tasks may want them; optimizing in $\boldsymbol{z}$-space quarantines that erasure in the head while $\boldsymbol{h}$ keeps the detail — so at evaluation the head is discarded and $\boldsymbol{h}$ is the feature. Second, negatives live in the batch, so dictionary size equals batch size, and the quality curve runs up to batches of 8192 — a pure compute moat. Third, augmentation *composition* — crop plus color jitter, especially together — was found to be the single biggest quality lever, because preventing one view from mimicking the other forces semantics to be the remaining invariant.

Student. MoCo gets the same dictionary size with a normal batch. Queue plus momentum encoder — show me why both are needed.

Mentor. MoCo (He et al., 2020) decouples the dictionary from the batch with a FIFO queue: after each step, the current mini-batch's keys are enqueued as negatives and the oldest are dequeued, so the dictionary holds tens of thousands of keys from *past* steps while the batch stays small. Gradients flow only through the query encoder (`k = k.detach()`), so the queue costs no backprop memory. The catch — and this is the insight the quiz probes — is consistency: as the query encoder trains, the keys in the queue were computed under different, older functions; comparing new queries to stale keys mismeasures similarity. Momentum encoding is the patch: a key encoder updated as an exponential moving average,

$$\boldsymbol{\theta}_k \leftarrow m\,\boldsymbol{\theta}_k + (1-m)\,\boldsymbol{\theta}_q,$$

with m near 1 — the slow camera in the session's photo-album analogy: small batches, an album of 65,000 negatives, and settings that drift slowly enough that all the album's photos share a look. Both halves load: the queue without momentum gives scale but incoherence; momentum without the queue (negatives only from the current batch) is just SimCLR with extra cost.

Student. Closer: Barlow Twins keeps the twin architecture but drops negatives entirely. How can a loss with no negatives avoid the constant-output collapse?

Mentor. By changing the axis on which the pressure acts: from sample-level repulsion to dimension-level decorrelation (Zbontar et al., 2021). Encode two augmented views to batch features $Z^A, Z^B \in \mathbb{R}^{B\times d}$ and compute the cross-correlation matrix across the batch (batch-normalized features),

$$C_{ij} = \frac{\sum_b z^A_{b,i}\, z^B_{b,j}}{\sqrt{\sum_b (z^A_{b,i})^2}\,\sqrt{\sum_b (z^B_{b,j})^2}} \in [-1,1], \qquad \mathcal{L}_{\mathrm{BT}} = \sum_i (1 - C_{ii})^2 + \lambda \sum_i \sum_{j\neq i} C_{ij}^2.$$

The diagonal terms of the first sum demand that dimension i agree across the two views — invariance. The off-diagonal second sum demands that distinct dimensions of the representation carry uncorrelated information — redundancy reduction. Now watch collapse die. A constant encoder has zero variance in every dimension, so once features are centred C_{ii} cannot reach 1 and the first sum stays at its maximum. The subtler collapse, where every dimension copies the same varying signal, drives the off-diagonal correlations to ± 1 and makes the second sum explode. The session's security-guard version: each guard must describe a distinctive feature — one does height, another hair, another clothing — because overlapping descriptions are penalized, so nobody can collapse into mouthing the same sentence. The closing slide positioned the taxonomy: generative/reconstructive (MAE), pretext-predictive (RotNet), alignment/contrastive (SimCLR, MoCo), and now redundancy-reductive (Barlow Twins) — four different places to put the pressure that stops the trivial answer.

Three revision questions the student implicitly asked.

1. *(From the RotNet deep dive)* Explain the shortcut problem for handcrafted pretext tasks through RotNet: why rotations by non-right angles break the task, what the conv1-filter comparison proves, and where the canonical-orientation assumption fails. → Ch. 8 exercises, pretext-design set.
2. *(From the PIRL and ladder exchanges)* Define covariance and invariance operators on a representation, exhibit the algebraic identity between the two display forms of the N-pair loss, and derive InfoNCE as a softmax cross-entropy and a mutual-information bound. → Ch. 9 exercises, contrastive-loss set.
3. *(From SimCLR/MoCo/Barlow)* Compare the three anti-collapse mechanisms — in-batch negatives plus projection head, queue plus EMA encoder, and cross-correlation whitening — and identify, for each, its compute cost and its failure mode. → Ch. 9–10 exercises, joint-embedding set.

E.6 Lecture 07: representation learning — SimSiam, BERT, DINO, DINOv2, data engines

Companion chapters: Chapter 10 (non-contrastive joint-embedding methods) and Chapter 11 (distillation-style SSL and the DINO family). The session taught the 55-slide deck in six blocks after an opening diagnostic recap of exam-style questions from earlier weeks (a DBSCAN case and a deep-clustering implementation review), and it ended with the student's hand-written-notes request for a compact DINOv2 revision sheet.

Student. SimSiam claims to learn representations with *no* negatives, *no* momentum encoder, and *no* big batch. After Lecture 06 I thought those were the only things preventing collapse. What is going on?

Mentor. First picture the failure precisely. Two views $\boldsymbol{x}_1, \boldsymbol{x}_2$ pass through the shared encoder f, yielding projections $\boldsymbol{z}_1, \boldsymbol{z}_2$; maximize their agreement and there is a trivial global optimum — the encoder maps every input to the same constant, agreement is perfect, the loss hits its floor. Slide 13 of the deck catches the trap in the wild: a training curve with the loss bottoming out while k-NN accuracy sits at chance level (0.1%). Low loss does not guarantee good features when the objective itself has a collapsed optimum — the session marked this as the most reusable exam reflex of the block, and it recalls the Lecture 02 lesson that training loss and representation quality are different axes.

SimSiam's answer is two intertwined asymmetries (Chen and He, 2021). A lightweight predictor MLP h transforms one side's projection, $\boldsymbol{p}_1 = h(\boldsymbol{z}_1)$, before comparison; and the comparison is symmetrized with the gradient stopped on the target side,

$$\mathcal{L} = \tfrac{1}{2}\, D(\boldsymbol{p}_1, \operatorname{sg}(\boldsymbol{z}_2)) + \tfrac{1}{2}\, D(\boldsymbol{p}_2, \operatorname{sg}(\boldsymbol{z}_1)), \qquad D(\boldsymbol{p}, \boldsymbol{z}) = -\frac{\boldsymbol{p}}{\|\boldsymbol{p}\|_2} \cdot \frac{\boldsymbol{z}}{\|\boldsymbol{z}\|_2},$$

a negative cosine similarity — the normalized geometry from Lecture 02 used as a training loss. Stop-gradient turns the target branch into a constant for that step: the encoder cannot satisfy the objective by collapsing toward a target that itself collapses, because the target does not move in response. The reading that made this click in the session is *alternating optimization*: view $\boldsymbol{z}_2$ as an estimate of a fixed target code $\boldsymbol{\eta}$, with sg severing backprop through it, so each step solves a small isolated prediction problem — an EM-like alternation — rather than a joint system with a degenerate solution. BYOL had achieved negatives-free training with a momentum teacher (Grill et al., 2020); SimSiam's contribution was showing that the momentum encoder is not what carries the load: stop-gradient plus predictor alone suffice. For the quiz (Q3, Q7, Q12), the invariant summary is: predictor supplies asymmetry, stop-gradient breaks the feedback loop, and a diagnosis of "loss low, downstream zero" is collapse, not convergence.

Student. The deck then jumps to NLP: BERT. Beyond the acronym, what exactly are the two pretraining objectives, and why the odd 80/10/10 masking rule?

Mentor. The framing problem BERT solves (Devlin et al., 2019): autoregressive language models are unidirectional by construction — the left-to-right factorization is what stops self-attention from trivially seeing the token it predicts — but bidirectional context is what rich token representations need. Masked language modeling (MLM) buys bidirectionality with corruption: randomly choose 15% of tokens, hide their identities (mostly), and force the network to reconstruct each masked token from *both* sides of the context; no left-to-right leak is possible because the answer itself is withheld from the input.
The 80/10/10 rule engineers away a subtle distribution mismatch. If the chosen 15% of tokens were *always* replaced by the literal `[MASK]` symbol, pretraining inputs would contain a token that fine-tuning inputs never do, and the encoder would specialize part of its capacity to it. So for each chosen token: 80% of the time substitute `[MASK]`; 10% substitute a random vocabulary token (the model cannot trust that a visible token is real, and the reconstruction target remains the original); 10% leave the token unchanged (the model must give every position a trained representation). Paired with MLM is next-sentence prediction (NSP): the `[CLS]` token's final state answers whether sentence B followed sentence A in the source corpus (balanced 50/50 against random pairings), giving the model a sentence-level relationship signal. The session recorded the later-verdict caveat that came up in the Q&A: subsequent work (RoBERTa) removed NSP — longer contiguous spans proved better for long-range dependencies — so NSP is historically load-bearing, not sacred. The parallel the mentor drew to Chapter 5: MLM at 15% masking alongside MAE at 75% is the same "how much context can we remove before the task stops being solvable by shortcuts" parameter, answered differently for dense-text and redundant-pixel regimes.

Student. Self-distillation with no labels: DINO trains a student to match its own EMA-updated copy. What stops the two networks from agreeing on garbage — and where do the segmentation maps come from?

Mentor. The loop DINO closes carefully (Caron et al., 2021). Multiple crops of one image are drawn — two global, several local; the teacher (an EMA of the student, $\boldsymbol{\theta}_t \leftarrow m\,\boldsymbol{\theta}_t + (1-m)\boldsymbol{\theta}_s$) sees only the global crops, and the student sees all of them; matching is a cross-entropy between the teacher's K-way softmax distribution and the student's. Since the teacher is just a time-lagged student, both could drift together into a constant output — collapse by mutual agreement. DINO's answer is a balanced pair of opposing pressures on the teacher's logits, and the session insisted they be learned as antidotes to *distinct* failure modes. First *centering*: maintain a running center vector $\boldsymbol{c} \in \mathbb{R}^K$, updated per batch by $\boldsymbol{c} \leftarrow m\,\boldsymbol{c} + (1-m)\,\frac{1}{B}\sum_{i=1}^{B} g_{\boldsymbol{\theta}_t}(\boldsymbol{x}_i)$, and subtract it before the softmax. This pushes the batch-averaged teacher prediction toward uniform across the K dimensions, so no single dimension can dominate — blocking the dominance form of collapse. Second, *sharpening*: divide the centered teacher logits by a small temperature $\tau_t < \tau_s$, $\mathrm{softmax}\big((g_{\boldsymbol{\theta}_t}(\boldsymbol{x}) - \boldsymbol{c})/\tau_t\big)$ — without it, centering alone would push the other way, toward the flat $1/K$-everywhere uniform distribution, which carries no information for the student to match. In the code, one line does both on the detached teacher logits: `t = softmax((t - C)/tpt, dim=1)`. The emergent phenomenon that made DINO famous: although the loss touches only the global `[CLS]` token, the ViT self-attention heads latch onto object boundaries and produce, for free, fine-grained unsupervised segmentation maps — a global consistency objective discovering local scene structure. This is also the frozen backbone that Lecture 02's STEGO distills correspondences from. That pairing of Q8's answer — centering blocks dimension dominance, sharpening blocks uniform blur, EMA stabilizes targets — next to SimSiam's — stop-gradient breaks the feedback loop, predictor supplies the asymmetry — is the cleanest two-line anti-collapse comparison of the session.

Student. Give me short notes on DINOv2 — what actually changed from DINO, and what is a "data engine"?

Mentor. DINOv2 keeps the skeleton and triples the objective (Oquab et al., 2024; Zhou et al., 2022; Sablayrolles et al., 2019). (1) The image-level DINO loss stays: global `[CLS]` distillation carries scene semantics. (2) An iBOT-style masked-image-modeling loss adds patch-level structure: mask patches in the student's input while the teacher sees the unmasked view, and match per-patch output distributions — DINO's global alignment supplemented by local granularity. (3) The KoLeo regularizer guards the geometry of the feature space itself: a Kozachenko–Leonenko differential-entropy estimate that penalizes nearest-neighbor crowding, $\mathcal{L}_{\text{KoLeo}} = -\frac{1}{B}\sum_i \log d_i$ over per-sample nearest-neighbor distances d_i on the sphere, pushing batch features to spread — a batch-level defense against the creeping dimensional collapse that a pure matching objective permits at scale.

The second half of the answer is data. On uncurated web corpora, quality gains come less from objectives than from a *curation engine*: embedding-based deduplication against curated seed sets, retrieval of domain-relevant images, and hierarchical k-means rebalancing over clusters of the embedding space to flatten the crawl's long tail — scaling from ImageNet's about 1.3M images to DINOv2's curated LVD-142M and, in its successor DINOv3, to the 1689M-image LVD-1689M the deck cites (Vo et al., 2024; Siméoni et al., 2025). The framing the session's notes sheet closed with: at foundation-model scale, SSL's bottleneck stops being the loss and becomes the data engine — deduplicate, retrieve, rebalance, *then* train the multi-objective model.

Three revision questions the student implicitly asked.

1. *(From the SimSiam exchange)* Define collapse for joint-embedding objectives, show why the plain symmetric cosine-agreement objective collapses, and explain how stop-gradient (with the asymmetric predictor) converts the dynamics into alternating optimization. → Ch. 10 exercises, anti-collapse set.
2. *(From the BERT exchange)* Write out the MLM corruption rule (15% selection, 80/10/10 substitution), justify each branch against pretrain–finetune mismatch, and contrast the 15% text masking rate with MAE's 75% image masking rate. → Ch. 11 exercises, masked-modeling set.
3. *(From the DINO exchanges with the DINOv2 notes)* Give DINO's two matched collapse modes (dimension dominance, uniformity) and their operator antidotes, then list the three DINOv2 objectives and state what problem the curation engine solves that no loss can. → Ch. 11 exercises, distillation set.

E.7 Lecture 08: mixtures — self-training, distillation, CycleGAN, world models, and the case study

Companion chapters: Chapter 12 (self-training, distillation, and distillation at scale) and Chapter 13 (latent world models). The session taught the 41-slide "mix" deck in six blocks, then closed with the deepest single answer of the series: a step-by-step read of a PyTorch case-study script implementing four of the week's ideas end to end on Fashion-MNIST and CartPole, reproduced in full below.

Student. Pseudo-labeling shows up in the deck right after the classical clusters. Is pseudo-labeling even self-supervised — and where exactly does it go wrong?

Mentor. The taxonomy answer is a fence and the deck sits deliberately on it (Lee, 2013): pseudo-labeling is strictly *semi-supervised* — it needs some labeled seed — yet is built from the SSL toolbox, which is why this course counts it in. Mechanically: train on the labeled pool, predict on the unlabeled pool, take the arg max as a hard label, retrain. Slide 6's reading of the mechanism is entropy minimization: hard pseudo-labels push the classifier's decision boundaries through low-density regions, since confident commitments minimize predictive entropy.

The failure is equally canonical: *confirmation bias*. The model's own confident mistakes become training targets; the next round is even more confident about them. This is the closed feedback loop the session had already met in Lecture 02's DeepCluster discussion — a network that is both label generator and label consumer — and the mitigations are mechanical rather than elegant: confidence filtering, loss down-weighting of pseudo-samples against real labels, strong perturbation of the student, and class-balance monitoring so the accepted pool does not collapse onto the majority class. The Deck-08 quiz ties each mitigation to a question (Q1, Q7, Q8, Q13), and the case study below exercises all four with real numbers.

Student. Noisy Student claims a student can *beat* its teacher. How can learning from someone ever outperform the teacher?

Mentor. By making the student's job harder (Xie et al., 2020). The pipeline inherits DeepCluster's teacher-runs-first shape (the Lecture 02 callback the mentor made explicit), but Noisy Student tilts two knobs at once: the student is at least as large as the teacher (usually larger), and it trains on the pseudo-labeled data under *stronger* noise — aggressive augmentation, dropout, stochastic depth — that the teacher never saw. The noise is the informational margin: the teacher's soft/hard labels encode clean-data decisions, while the student must reproduce those decisions *under corruption*, which forces it to learn features that agree with the teacher but, by robustness, exceed it. Iterated, teacher becomes student and the ratchet clicks upward. This is also the bridge into distillation proper: next the targets stop being hard labels.

Student. Distillation with soft targets: walk me through the temperature algebra, including that T^2 factor every implementation carries.

Mentor. Hinton's distillation (Hinton et al., 2015) trains the student against the teacher's full softened distribution rather than its arg max. With logits $\boldsymbol{u}_t, \boldsymbol{u}_s$ and temperature T,

$$\mathcal{L}_{\text{soft}} = T^2 \cdot D_{\text{KL}}\Big(\text{softmax}(\boldsymbol{u}_t/T) \,\|\, \text{softmax}(\boldsymbol{u}_s/T)\Big).$$

Raising T flattens both softmaxes, surfacing what Hinton called the *dark knowledge*: the teacher saying 51% shirt, 49% coat is structurally more informative than the hard label "shirt" — argmax discards exactly the alternatives, and those alternatives encode the semantic similarity between classes. (The case study's plot title is the whole argument: "argmax discards alternatives"; hard labels keep 100% shirt and erase the shirt–coat relation.) Why the T^2 multiplier: scaling logits by $1/T$ shrinks their gaps, and the gradients of the KL term with respect to the student logits scale like $1/T^2$ — without the compensating factor the soft-target gradient's magnitude collapses as T grows and loss-weight comparisons across temperatures become meaningless. The limit reading the session worked through: as $T \to \infty$, matching softened distributions degenerates toward matching raw logits in mean-squared error. The rebranding worth remembering: DINO, from Lecture 07, is the same student–EMA-teacher machinery with the labeled dataset removed — "distillation with no labels"; and DUNE generalizes the head structure: one shared student encoder with a per-teacher projection head lets heterogeneous teachers — DINOv2-style semantics, MASt3R 3D geometry, pose — distill into one encoder even though their output spaces cannot be averaged (Sariyildiz et al., 2025; Leroy et al., 2024). (Q15 in the quiz asks exactly this, and the case study answers it in code.)

Student. CycleGAN: why can't the adversarial loss alone do unpaired translation? And state the failure signature the exam always asks me to diagnose.

Mentor. In unpaired translation there is no ground-truth pair checking content (Zhu et al., 2017; Isola et al., 2017). The adversarial loss checks only *target-domain realism*: a generator $G : X \to Y$ that maps every sneaker to the same plausible boot fools the discriminator perfectly. Nothing in the loss sees the input after mapping. Cycle consistency closes that hole with the round trip, $F(G(\boldsymbol{x})) \approx \boldsymbol{x}$ and $G(F(\boldsymbol{y})) \approx \boldsymbol{y}$ via an L_1 reconstruction: a mapping that must be (approximately) invertible cannot destroy content, because the content is what you rebuild on the way back. The diagnostic signature (Q16/Q17): realistic target-domain outputs *with* a failed round trip implies the cycle weight is too low, missing, or misimplemented — raise λ or inspect the reconstruction term. The paired/unpaired boundary is the taxonomy lesson: Pix2Pix's paired supervision buys content supervision directly; CycleGAN buys it via invertibility, and the price is exactly this sensitivity to the cycle weight — the case study below makes the failure visible with $\lambda = 0$ versus $\lambda = 3$.

Student. Last block: latent world models. Is training a transition model on a frozen encoder "self-supervised," and how does DINO-WM plan with it?

Mentor. The boundary question first (Q12's answer): when the next observation encodes the prediction target for free — $\boldsymbol{z}_{t+1} = E(o_{t+1})$ is generated by time itself — then yes, dynamics learning is self-supervised in the strict sense; "time acts as the supervisor." DINO-WM (Zhou et al., 2024) freezes a DINOv2 encoder as the state space and learns an action-conditioned transition model in that latent space. Autoregressive next-state prediction is the vanilla variant; the action-conditioned residual variant — predict $\Delta\boldsymbol{z}$, return $\boldsymbol{z} + \Delta\boldsymbol{z}$ — is the one the case study implements, because residual structure is easy to learn and keeps rollouts anchored. Planning then happens without ever touching pixels: search over action sequences (DINO-WM uses the cross-entropy method; the case study below simply enumerates them), roll each through the learned dynamics in latent space, score the final latent against a goal latent, execute the first action of the winner, replan. The argument that closed the session: this is Chapter 13's point that good SSL features are not just for linear probes — they can be the state space of a planner.

Student. Explain the case study script as well, in depth — step by step, so I can use it directly on the exam.

Mentor. The script is four systems on two playgrounds: 1,000 labeled plus 3,000 unlabeled Fashion-MNIST images, and CartPole. Section by section.

Section 1 — self-training with confidence filtering (Q13). Stage one: a `FashionCNN(width=16)` teacher trained on the clean labeled set under weak augmentation (minor cropping). Stage two: inference on the pool, `probs.max(dim=1)` producing both the hard pseudo-label and a confidence score. Stage three: a *relative* filter — `torch.quantile(confidence, 0.60)` keeps the top 40% — rather than an absolute threshold that drifts with calibration. Stage four: retrain a wider student (`width=20`) on the union, with per-sample loss weights — 1.0 on true labels, 0.5 on accepted pseudo-labels — under strong augmentation (random crop, h-flip, random erasing). Four line items for the written exam as a defensive design answer: confidence filtering, asymmetric loss weights, perturbed student, class-balance monitoring of the accepted pool. The confirmation-bias teaching point sits in the script's own plot: the teacher's confidence histogram contains confident mistakes that pass the quantile and get reinforced.

Section 2A — soft-target distillation (Q14). Teacher width 16 to student width 8 (half channels). The loss is `F.kl_div(log_softmax(student/T), softmax(teacher/T)) * T**2` with $T = 2.0$, plus ordinary cross-entropy on the labeled points. Both exam lines from the exchange above appear verbatim in code: soft targets preserving the 51%-shirt / 49%-coat relation, and the T^2 gradient-magnitude compensation.

Section 2B — heterogeneous teachers (Q15). One shared encoder feeds two heads: a semantic head (linear to a 10-way softmax, KL loss) and an edge head (conv from the width-20 features to a sigmoid edge map, L_1 loss), trained against a *frozen Sobel filter* as the second teacher — a non-neural teacher kept deliberately, to force the point that "teacher" need not be a network. The structural reading: a 10-vector of class logits and a $1 \times 7 \times 7$ spatial map cannot be averaged into one target, so task-specific projection heads translate each teacher's space, and both losses backpropagate into the shared encoder — the DUNE architecture answered in miniature.

Section 3 — CycleGAN in two runs (Q16–Q17). Domain X: normal Fashion-MNIST; domain Y: inverted images. Generators $G : X \to Y$ and $F : Y \to X$, discriminators D_X, D_Y, and the single knob `cycle_weight`. At `cycle_weight=0`: outputs adopt target style — dark garments on bright background — but content drifts; a boot returns as a t-shirt. Adversarial loss never checked the input. At `cycle_weight=3`: style transfers while garment identity and layout survive the round trip. The diagnostic answer, memorized as a unit: target-realistic output plus failed reconstruction means raise the cycle weight or fix the L_1 term.

Section 4 — action-conditioned world model and planning (Q6, Q12, Q18). CartPole observations are encoded to a 4-dimensional latent $\boldsymbol{z}$; actions are one-hot. The `Dynamics` net concatenates [$\boldsymbol{z}$, one-hot(a)] ($4 + 2 = 6$ inputs) through a $6 \to 48 \to 48 \to 4$ MLP with SiLU activations and predicts a *residual*: output $\Delta\boldsymbol{z}$, return $\boldsymbol{z} + \Delta\boldsymbol{z}$. Three phases: encode observations; train the transition model with plain `F.mse_loss(pred_next_z, next_z)` — the next latent is the free target, "time is the supervisor"; then test-time planning: enumerate all 1,024 possible 10-step action sequences, roll each horizon-10 latent rollout through the dynamics, score the terminal latent against the upright-goal latent, execute the best first action, replan — the predict–search–execute loop DINO-WM demonstrates at image scale.

The closing table the mentor drew is worth reconstructing on the fly in an exam: weighted losses manage confirmation bias; soft targets preserve uncertainty; projection heads unify

Three revision questions the student implicitly asked.

1. *(From the pseudo-label and case-study Section 1 exchanges)* Design a self-training protocol for 1,000 labeled and 3,000 unlabeled images: justify a quantile-based confidence filter, the 1.0/0.5 loss weighting, student perturbation, and class-balance monitoring, and name the failure mode each mechanism addresses. → Ch. 12 exercises, self-training set.
2. *(From the distillation exchanges)* Derive why the soft-target KL loss is multiplied by T^2, explain what hard labels discard relative to soft targets, and describe how projection heads allow distilling from teachers with incompatible output spaces. → Ch. 12 exercises, distillation set.
3. *(From the CycleGAN and world-model exchanges)* Show why the adversarial objective alone cannot preserve content in unpaired translation, diagnose the boot-to-t-shirt failure from the training configuration, and construct the latent-rollout planning loop for an action-conditioned residual dynamics model. → Ch. 12–13 exercises, translation and world-model set.

Bibliography

Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 49–60, 1999. doi: 10.1145/304182.304187.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, 2007.

Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. BEiT: BERT pre-training of image transformers. In *International Conference on Learning Representations*, 2022.

Adrien Bardes, Jean Ponce, and Yann LeCun. VICReg: Variance-invariance-covariance regularization for self-supervised learning. In *International Conference on Learning Representations*, 2022.

Suzanna Becker and Geoffrey E. Hinton. Self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, 1992.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In *International Conference on Learning Representations*, 2019.

Jane Bromley, James W. Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. Signature verification using a "Siamese" time delay neural network. In *Advances in Neural Information Processing Systems 6*. 1993.

Tim Brooks, Aleksander Holynski, and Alexei A. Efros. InstructPix2Pix: Learning to follow image editing instructions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023.

Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. Deep clustering for unsupervised learning of visual features. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 139–156, 2018.

Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments. In *Advances in Neural Information Processing Systems 33*, 2020.

Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems 31*, 2018.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning*, 2020a.

Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E. Hinton. Big self-supervised models are strong semi-supervised learners. In *Advances in Neural Information Processing Systems 33*, 2020b.

Xinlei Chen and Kaiming He. Exploring simple Siamese representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. Improved baselines with momentum contrastive learning, 2020c.

Xinlei Chen, Saining Xie, and Kaiming He. An empirical study of training self-supervised vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. In *Proceedings of Robotics: Science and Systems XIX*, 2023.

Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

Harm de Vries, Florian Strub, Jérémie Mary, Hugo Larochelle, Olivier Pietquin, and Aaron Courville. Modulating early visual processing by language. In *Advances in Neural Information Processing Systems*, 2017.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

Prafulla Dhariwal and Alexander Quinn Nichol. Diffusion models beat GANs on image synthesis. In *Advances in Neural Information Processing Systems 34*, 2021.

Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: Non-linear independent components estimation, 2015.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. In *International Conference on Learning Representations*, 2017.

Carl Doersch. Tutorial on variational autoencoders, 2016.

Carl Doersch, Abhinav Gupta, and Alexei A. Efros. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. In *International Conference on Learning Representations*, 2017.

Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The Pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

Jonas Frey, Hermann Blum, Francesco Milano, Roland Siegwart, and César Cadena. Continual adaptation of semantic segmentation using complementary 2D-3D data representations. *IEEE Robotics and Automation Letters*, 2022.

Jonas Frey, Matías Mattamala, Nived Chebrolu, César Cadena, Maurice Fallon, and Marco Hutter. Fast traversability estimation for wild visual navigation. In *Robotics: Science and Systems XIX*, 2023.

Gemini Team, Google. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*, pages 881–889, 2015.

Spyros Gidaris, Praveer Singh, and Nikos Komodakis. Unsupervised representation learning by predicting image rotations. In *International Conference on Learning Representations*, 2018.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, 1999.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 27, 2014.

Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: Free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations*, 2019.

Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, and Michal Valko. Bootstrap your own latent – a new approach to self-supervised learning. In *Advances in Neural Information Processing Systems 33*, 2020.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, 2017.

Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010.

Mark Hamilton, Zhoutong Zhang, Bharath Hariharan, Noah Snavely, and William T. Freeman. Unsupervised semantic segmentation by distilling feature correspondences. In *International Conference on Learning Representations*, 2022.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In *Advances in Neural Information Processing Systems*, 2017.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance, 2022.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems 33*, 2020.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. In *International Conference on Machine Learning*, pages 2078–2087, 2018.

Michael F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics - Simulation and Computation*, 1990.

Aapo Hyvärinen. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 2005.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998. doi: 10.1145/276698.276876.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456, 2015.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1125–1134, 2017.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

Ian T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer, New York, 2 edition, 2002.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. In *International Conference on Learning Representations*, 2018.

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data. In *Advances in Neural Information Processing Systems*, 2020a.

Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of StyleGAN. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020b.

Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-free generative adversarial networks. In *Advances in Neural Information Processing Systems*, 2021.

Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations*, 2014.

Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019. doi: 10.1561/2200000056.

Diederik P. Kingma, Tim Salimans, Rafal Józefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

John Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, 2001.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Dong-Hyun Lee. Pseudo-Label: The simple and efficient semi-supervised learning method for deep neural networks. ICML 2013 Workshop on Challenges in Representation Learning (WREPL), 2013.

Vincent Leroy, Yohann Cabon, and Jérôme Revaud. Grounding image matching in 3D with MASt3R. In *Computer Vision – ECCV 2024*, 2024.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In *Computer Vision – ECCV 2014*, 2014.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow matching for generative modeling. In *International Conference on Learning Representations*, 2023.

Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *International Conference on Learning Representations*, 2023.

Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. doi: 10.1109/TIT.1982.1056489.

Andreas Lugmayr, Martin Danelljan, Luc Van Gool, and Radu Timofte. SRFlow: Learning the super-resolution space with normalizing flow. In *European Conference on Computer Vision*, pages 715–732, 2020.

James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for GANs do actually converge? In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.

Ishan Misra and Laurens van der Maaten. Self-supervised learning of pretext-invariant representations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Takeru Miyato and Masanori Koyama. cGANs with projection discriminator. In *International Conference on Learning Representations*, 2018.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

Eric Nalisnick, Akihiro Matsukawa, Yee Whye Teh, Dilan Gorur, and Balaji Lakshminarayanan. Do deep generative models know what they don't know? In *International Conference on Learning Representations*, 2019.

Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *Computer Vision – ECCV 2016*, 2016.

Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier GANs. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy V. Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, Mahmoud Assran, Nicolas Ballas, Wojciech Galuba, Russell Howes, Po-Yao Huang, Shang-Wen Li, Ishan Misra, Michael Rabbat, Vasu Sharma, Gabriel Synnaeve, Hu Xu, Hervé Jégou, Julien Mairal, Patrick Labatut, Armand Joulin, and Piotr Bojanowski. DINOv2: Learning robust visual features without supervision. *Transactions on Machine Learning Research*, 2024.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021.

Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018.

Ed Pizzi, Sreya Dutta Roy, Sugosh Nagavara Ravindra, Priya Goyal, and Matthijs Douze. A self-supervised descriptor for image copy detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

Ben Poole, Sherjil Ozair, Aaron van den Oord, Alex Alemi, and George Tucker. On variational bounds of mutual information. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations*, 2016.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf, 2018.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya

Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

Mike Ranzinger, Greg Heinrich, Jan Kautz, and Pavlo Molchanov. AM-RADIO: Agglomerative vision foundation model – reduce all domains into one. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538, 2015.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1278–1286, 2014.

Francisco Rivera Valverde, Juana Valeria Hurtado, and Abhinav Valada. There is more than meets the eye: Self-supervised multi-object detection and tracking with sound by distilling multimodal knowledge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015, Part III*, pages 234–241, 2015. doi: 10.1007/978-3-319-24574-4_28.

Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. Spreading vectors for similarity search. In *International Conference on Learning Representations*, 2019.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs. In *Advances in Neural Information Processing Systems*, 2016.

Mert Bülent Sariyildiz, Philippe Weinzaepfel, Thomas Lucas, Pau de Jorge, Diane Larlus, and Yannis Kalantidis. DUNE: Distilling a universal encoder from heterogeneous 2D and 3D teachers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025.

Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

David Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, pages 1177–1178, 2010. doi: 10.1145/1772690.1772862.

Oriane Siméoni, Huy V. Vo, Maximilian Seitzer, Federico Baldassarre, Maxime Oquab, Cijo Jose, Vasil Khalidov, Marc Szafraniec, Seungeun Yi, Michaël Ramamonjisoa, Francisco Massa, Daniel Haziza, Luca Wehrstedt, Jianyuan Wang, Timothée Darcet, Théo Moutakanni, Leonel Sentana, Claire Roberts, Andrea Vedaldi, Jamie Tolan, John Brandt, Camille Couprie, Julien Mairal, Hervé Jégou, Patrick Labatut, and Piotr Bojanowski. DINOv3, 2025.

Kihyuk Sohn. Improved deep metric learning with multi-class N-pair loss objective. In *Advances in Neural Information Processing Systems 29*, 2016.

Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *International Conference on Learning Representations*, 2021a.

Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2021b.

Hugo Steinhaus. Sur la division des corps matériels en parties. *Bulletin de l'Académie Polonaise des Sciences*, IV(12):801–804, 1956.

Charles Sun, Jędrzej Orbik, Coline Devin, Brian H. Yang, Abhishek Gupta, Glen Berseth, and Sergey Levine. Fully autonomous real-world reinforcement learning with applications to mobile manipulation. In *Proceedings of the 5th Conference on Robot Learning*, 2022.

Luming Tang, Menglin Jia, Qianqian Wang, Cheng Perng Phoo, and Bharath Hariharan. Emergent correspondence from image diffusion. In *Advances in Neural Information Processing Systems 36*, 2023.

Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models. In *International Conference on Learning Representations*, 2016.

Yonglong Tian, Chen Sun, Ben Poole, Dilip Krishnan, Cordelia Schmid, and Phillip Isola. What makes for good views for contrastive learning? In *Advances in Neural Information Processing Systems 33*, 2020.

Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method, 2000.

Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, 2003.

Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

Aäron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis C. Cobo, and Florian Stimberg. Parallel WaveNet: Fast high-fidelity speech synthesis. In *International Conference on Machine Learning*, pages 3918–3926, 2018a.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2018b.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30*, 2017.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1096–1103, 2008. doi: 10.1145/1390156.1390294.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11:3371–3408, 2010.

Huy V. Vo, Vasil Khalidov, Timothée Darcet, Théo Moutakanni, Nikita Smetanin, Marc Szafraniec, Hugo Touvron, Camille Couprie, Maxime Oquab, Armand Joulin, Hervé Jégou, Patrick Labatut, and Piotr Bojanowski. Automatic data curation for self-supervised learning: A clustering-based approach, 2024.

Carl Vondrick, Abhinav Shrivastava, Alireza Fathi, Sergio Guadarrama, and Kevin Murphy. Tracking emerges by colorizing videos. In *Computer Vision – ECCV 2018*, 2018.

Christina Winkler, Daniel Worrall, Emiel Hoogeboom, and Max Welling. Learning likelihoods with conditional normalizing flows, 2019.

Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V. Le. Self-training with noisy student improves ImageNet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny. Barlow twins: Self-supervised learning via redundancy reduction. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. SWAG: A large-scale adversarial dataset for grounded commonsense inference. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Richard Zhang, Phillip Isola, and Alexei A. Efros. Colorful image colorization. In *Computer Vision – ECCV 2016*, 2016.

Richard Zhang, Phillip Isola, and Alexei A. Efros. Split-brain autoencoders: Unsupervised learning by cross-channel prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Shengyu Zhao, Zhijian Liu, Ji Lin, Jun-Yan Zhu, and Song Han. Differentiable augmentation for data-efficient GAN training. In *Advances in Neural Information Processing Systems*, 2020.

Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ADE20K dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Gaoyue Zhou, Hengkai Pan, Yann LeCun, and Lerrel Pinto. DINO-WM: World models on pre-trained visual features enable zero-shot planning, 2024.

Jinghao Zhou, Chen Wei, Huiyu Wang, Wei Shen, Cihang Xie, Alan Yuille, and Tao Kong. iBOT: Image BERT pre-training with online tokenizer. In *International Conference on Learning Representations*, 2022.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, 2017.

Siting Zhu, Guangming Wang, Hermann Blum, Jiuming Liu, Liang Song, Marc Pollefeys, and Hesheng Wang. SNI-SLAM: Semantic neural implicit SLAM. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

Yuliang Zou, Zelun Luo, and Jia-Bin Huang. DF-Net: Unsupervised joint learning of depth and flow using cross-task consistency. In *Computer Vision – ECCV 2018*, 2018.

René Zurbrügg, Hermann Blum, César Cadena, Roland Siegwart, and Lukas Schmid. Embodied active domain adaptation for semantic segmentation via informative path planning. *IEEE Robotics and Automation Letters*, 2022.

Index

catastrophic forgetting, 652
coordinate frame, 627
covariance, 388
cross-entropy method, 608

depth, 659
depth image, 626
dimensional collapse, 726
domain shift, 639
drift, 629

ego-motion, 628, 660
eigenvalues, 726
ensemble, 692

flat, 716
focal loss, 685

invariance, 388

label, 4

manifold, 716
mIoU, 639
model-predictive control, 609
multi-teacher alignment, 687

Non-maximum suppression, 682

object detector, 682
occluded, 631
odometry, 628, 640
Optical flow, 660
optical flow, 631

Paired data, 623
parallax, 633
pixel, 625
point cloud, 626
pose, 628

ray casting, 631
Reinforcement learning, 691
replay buffer, 652
representation, 4
RGB-D, 626

semantic segmentation, 639
SLAM, 629, 640

Transfer, 4
Traversability, 672

voxel, 626
voxels, 640