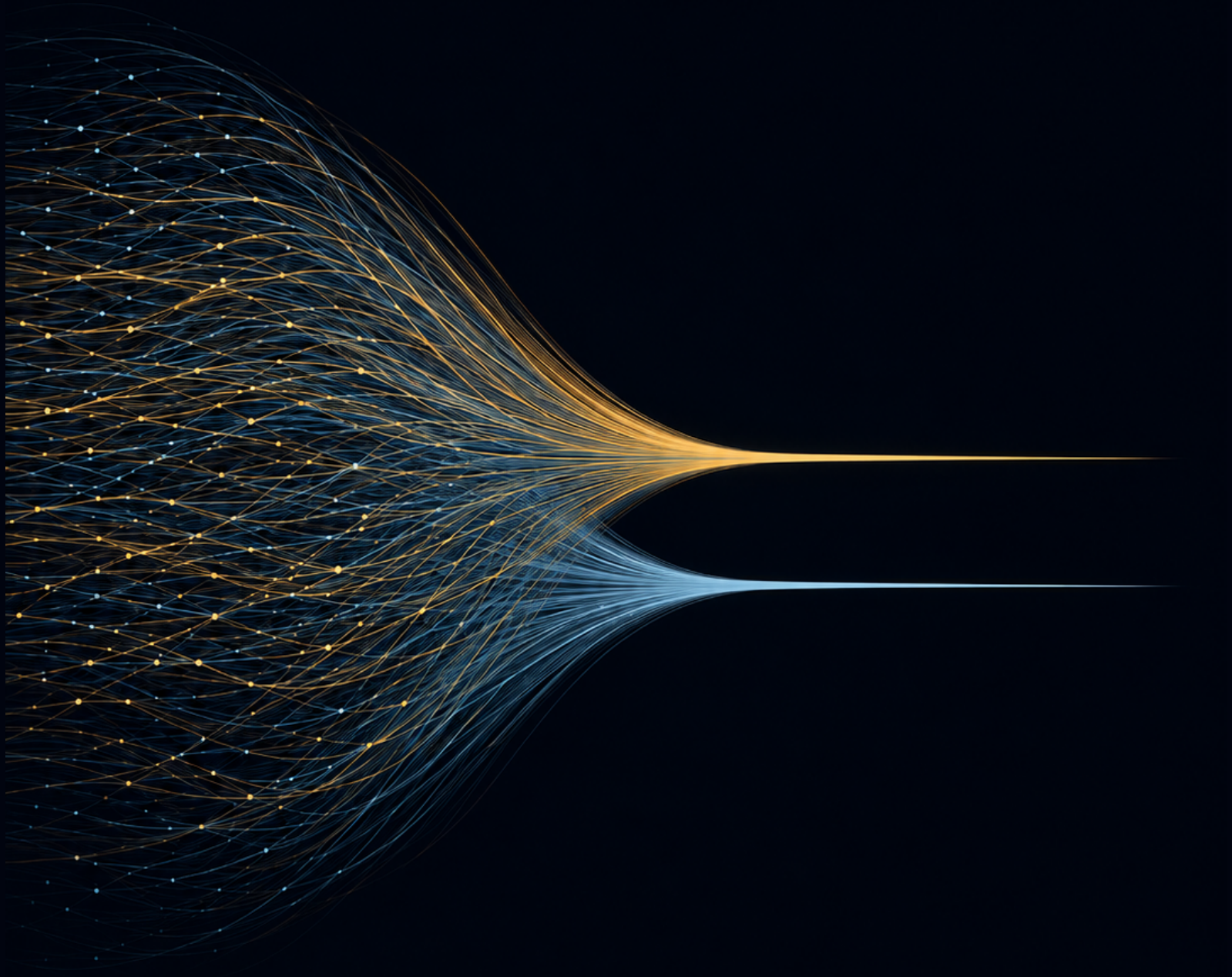


Deep Learning

Making It Learnable

WHAT TO LEARN • WHAT TO BUILD IN • WHAT TO REUSE



HEMAN SHAKERI

A FIRST-PRINCIPLES COURSE IN PYTHON AND PYTORCH

Deep Learning: Making It Learnable

What to Learn, What to Build In, and What to Reuse

Heman Shakeri

Rolling post-v1.2.1 build · August 2026

A first-principles course in Python and PyTorch

Table of contents

Preface	1
How to read this book	2
Course route and dependencies	2
Plan a reading session	3
Exercises, hints, and solutions	3
Acknowledgments	3
Support this open book	4
I. From Lines to Networks	5
1. Linear Regression, the Mother Model	6
1.1. Learning from examples	6
1.1.1. Why is this hard?	7
1.2. The linear model and its geometry	8
1.3. The loss: what makes a hyperplane “bad”?	9
1.4. Finding the best weights, method 1: solve it exactly	9
1.5. Finding the best weights, method 2: walk downhill	10
1.6. Build the complete model three ways	12
1.6.1. Take 1: the closed form	14
1.6.2. Take 2: gradient descent, from scratch	14
1.6.3. Take 3: the framework way	15
1.7. Why squared error? The maximum-likelihood view	18
1.8. A first look at bias and variance	20
1.8.1. Regularization, briefly	21
1.9. Okay, so — what did we just build?	23
Sources and further reading	23
Exercises	24
2. Linear Models that Classify: Logistic and Softmax	25
2.1. What actually changes	25
2.2. Binary classification	25
2.2.1. From score to probability: the sigmoid	25
2.2.2. The loss: cross-entropy from maximum likelihood	26
2.2.3. The beautiful gradient	27
2.3. Multiclass classification: softmax	27
2.3.1. Cross-entropy, multiclass edition	29
2.3.2. One numerical landmine	30
2.4. Build it: three classes, from scratch	30
2.5. Okay, so — classification is regression plus a normalizer	33
Sources and further reading	33
Exercises	33

3. Nonlinearity and the MLP	35
3.1. The tiny dataset that embarrassed a field	35
3.2. The neuron: a threshold with a bend	38
3.3. Layers of hinges: bumps and polytopes	39
3.4. The universal approximation theorem	41
3.5. Why depth: boundaries that bend	43
3.6. Your first real MLP	44
3.7. A cautionary tale, and a look ahead	47
3.8. Okay, so — one bend changed everything	48
Sources and further reading	48
Exercises	48
4. Training: Loss and Stochastic Gradient Descent	50
4.1. Where losses come from, one more time	50
4.2. The computational bottleneck	50
4.3. Stochastic gradient descent	51
4.4. The two zones of SGD	51
4.5. The learning rate	55
4.6. The batch size	57
4.6.1. What a batch may estimate — and what it may not	57
4.7. Momentum: give the ball some mass	58
4.8. Adam: adaptive steps per knob	58
4.9. The race, on a problem where we know the truth	58
4.10. Three regularizers that live inside training	61
4.11. Okay, so — the training loop	64
Sources and further reading	65
Exercises	66
5. Backpropagation	67
5.1. One neuron, one chain	67
5.2. The game of blame	68
5.3. Build it, then check it against autograd	71
5.4. A tiny scalar autograd engine	73
5.5. <code>torch.autograd</code> in practice: five rules	75
5.6. The gate, and the problem with deep chains	77
5.7. Initialization: setting the signal's energy	81
5.8. Okay, so — the chain rule, organized	83
Sources and further reading	83
Exercises	83
6. When It Fails: Generalization in Pictures, and Inductive Bias	85
6.1. Victory	85
6.2. Two experiments that should worry you	87
6.2.1. Experiment 1: move everything two pixels	87
6.2.2. Experiment 2: destroy the images entirely	89
6.3. The autopsy, in pictures	90
6.4. Naming the disease	91
6.4.1. Hunting the U-curve	91
6.4.2. The curse of dimensionality	92

6.5.	The cure has a name	94
6.5.1.	Data augmentation: put the belief in the examples	94
6.6.	The pivot	95
6.7.	One sealed test check	95
6.8.	Okay, so — the lesson of Part I	96
	Deeper dive	96
	Sources and further reading	97
	Exercises	97
Interlude: Who Trains the Trainer? Learning by Experiment		98
	Two levels of learning	98
	Start with the claim, not the sweep	99
	“Fair” has more than one meaning	99
	A worked experiment: BatchNorm changes the learning-rate story	100
	Ablation is an argument about cause	107
	Hyperparameter optimization is a budgeted outer loop	108
	Choose the space before choosing the algorithm	108
	Spend small budgets before large ones	108
	Train, validation, and test have different jobs	109
	Seeds are a panel, not decoration	109
	The experiment ledger	110
	A protocol for the rest of this book	110
	Okay, so — tune the contender, ablate the claim	111
	Sources and further reading	111
	Exercises	112
II. Vision: Learning the Filters		113
7. Filters and Convolution (Fixed Kernels)		114
7.1.	Two principles, one strategy	114
7.2.	Start in 1D: the moving average	114
7.3.	To 2D: the recipe	115
7.4.	The filter zoo	117
7.5.	Naming the operation	119
7.6.	The property we paid for: equivariance	119
7.7.	The matrix view: a structured, weight-shared linear map	120
7.8.	The bottleneck: someone has to design the kernel	121
7.9.	Okay, so — the machine before the learning	122
	Sources and further reading	122
	Exercises	123
8. CNNs: Making the Filters Learnable		124
8.1.	Nine numbers, learned	125
8.2.	From one detector to a layer of detectives	127
8.3.	Padding and stride: the bookkeeping with a payoff	129
8.4.	Pooling: trading exact location for local tolerance	130
8.5.	How far a deep neuron sees	132
8.6.	LeNet: the whole machine	133
8.7.	The rematch	138

8.8.	One sealed test check	140
8.9.	Okay, so — the kernel became learnable	141
	Sources and further reading	142
	Exercises	142
9.	Modern CNNs and Transfer Learning	143
9.1.	Question 1: why 5×5 ? Stack small kernels instead	146
9.2.	The stabilizer we owe you: batch normalization	147
9.3.	Building with blocks: a small VGG	149
9.4.	Question 3: 1×1 convolutions, and firing the flatten head	150
9.5.	Question 2: going deeper, and the wall you hit	154
9.6.	The scorecard: what a decade of design bought	159
9.7.	Transfer learning: the mechanics, and an honest experiment	161
9.7.1.	The full-data rematch	164
9.8.	Okay, so — modern CNNs are an optimization story	167
	Sources and further reading	168
	Exercises	168
Interlude:	Autoencoders—Making PCA Learnable	170
	Compress, then reconstruct	170
	Make PCA learnable	171
	What if the map could bend?	172
	A code is not yet a distribution	176
	Let the autoencoder see locally	177
	Why a one-shot encoder cannot be the whole sequence model	182
	Okay, so — PCA became a network, then the code became a bottleneck	183
	Sources and further reading	183
	Exercises	183
III.	Sequences	185
10.	Sequences and Recurrence	186
10.1.	Stretching the old toolkit	186
10.2.	A network with a loop	187
10.3.	Blame through time	188
10.3.1.	Training with a finite horizon	191
10.4.	Engineering a memory: the LSTM	191
10.4.1.	The memory test	193
10.4.2.	Watching the valves	195
10.5.	GRU: the streamlined cousin	197
10.6.	The book reads itself	198
10.7.	A full-scale rematch: words, layers, and corpus size	204
10.8.	What a fixed-size state cannot hold	205
10.9.	Okay, so — weight sharing moved into time	206
	Sources and further reading	206
	Exercises	207
11.	Encoder–Decoder, Teacher Forcing, Beam Search	208
11.1.	Two RNNs and a handoff	208

11.2. The padding trap	211
11.3. Teacher forcing: training on the gold rail	216
11.4. Getting answers out: search	218
11.5. The fixed-size state in the middle	221
11.6. Okay, so — the fixed-size handoff is the bottleneck	221
Sources and further reading	222
Exercises	222
IV. Attention: Learning the Similarity	223
12. Kernel Regression: Attention Before It Was Learnable	224
12.1. Chapter 1 already mixed old answers	224
12.2. The magnifying glass becomes a kernel	225
12.3. Chapter 2’s scores-to-weights machine	229
12.4. Bandwidth: the honesty gate	229
12.5. From one query to an attention matrix	231
12.6. Return the fixed operator to the memory bank	235
12.7. Okay, so — attention is normalized memory mixing	236
Sources and further reading	236
Exercises	237
13. Attention: Making the Kernel Learnable	238
13.1. The scores-to-weights machine, one more time	239
13.2. Additive attention: learn the comparison itself	239
13.3. Scaled dot product: learn the spaces, simplify the score	241
13.3.1. Why divide by $\sqrt{d_k}$?	243
13.4. Mask before softmax, and mask the right thing	245
13.5. Return to Chapter 11’s date task	247
13.5.1. Put the learned read inside the recurrent decoder	249
13.5.2. The matched-schedule rematch	252
13.6. One learned view, for now	256
13.7. The bottleneck that remains	257
13.8. Okay, so — attention softens the address	257
Sources and further reading	258
Exercises	258
14. Self-Attention and the Transformer	259
14.1. Let the sequence query itself	259
14.1.1. The position debt	260
14.2. Position as a bank of clocks	262
14.3. One routing operation, many heads	264
14.3.1. The mask is part of the model	265
14.4. Build a Transformer block	268
14.4.1. The residual stream	268
14.4.2. The position-wise feedforward network	271
14.5. From the original Transformer to this experiment	271
14.6. The book reads itself, again	272
14.6.1. What did the heads route?	280
14.6.2. Listen, but do not grade by fluency	281

14.7. What the architecture bought	284
14.8. Okay, so — the Transformer routes, then computes	284
Exercises	285
Interlude: Attention as Test-Time Regression	287
One regression, three solvers	287
One objective, four dials	287
Solver 1: keep the data and fit at the query	288
Solver 2: collapse a factorized kernel into sufficient state	289
Solver 3: take one gradient step and get the delta rule	290
The price list	292
Mechanism test: recall under a fixed capacity	292
Okay, so — the solver is part of the architecture	296
Sources and further reading	296
Exercises	296
15. The BERT Moment: Pretraining as the New Regime	298
15.1. Separate controls, not one mask	298
15.2. Text supplies supervision	300
15.3. The masked-language-model objective	301
15.3.1. Fifteen percent, then 80/10/10	301
15.4. From a Transformer encoder to BERT	303
15.4.1. [CLS] is a learned meeting place	306
15.5. Fine-tuning means the backbone moves	306
15.6. A controlled transfer lab	307
15.6.1. Build a latent regularity	307
15.6.2. A small BERT-style encoder	310
15.6.3. Pretrain, then adapt	313
15.6.4. What transferred?	318
15.7. Three pretraining families	321
15.8. What changed after 2018	322
15.9. Okay, so — pretraining manufactures supervision	323
Sources and further reading	323
Exercises	324
16. Vision Transformers and Scaling Laws	325
16.1. Let the image become a sequence	325
16.2. A learned meeting place for patches	328
16.3. Inductive bias strikes again	329
16.4. A Fashion rematch, not a referendum	329
16.5. Scale changes the comparison	338
16.5.1. Three places to buy back useful bias	338
16.6. The word “scaling” changes jobs	339
16.6.1. Compound scaling: three knobs, one budget	339
16.7. A fitted power law, not a promise	340
16.8. Compute-optimal means balanced	343
16.9. Where the forecast stops	347
16.10. Okay, so — patches become tokens, but regime still matters	347
Sources and further reading	348
Exercises	348

V. The Pretrained Era	350
17. Adapting Pretrained Models: Prompting, PEFT, Quantization	351
17.1. Prompting: move evidence into context	351
17.1.1. A frozen-context mechanism test	353
17.1.2. Retrieval is another context lever	357
17.2. What if the prompt itself were learnable?	358
17.3. LoRA: a low-rank correction beside a frozen path	359
17.3.1. Make the rank bottleneck fail on purpose	360
17.4. Quantization: put weights on a smaller grid	364
17.4.1. One scale can erase a quiet channel	366
17.4.2. What a quantization workflow protects	369
17.4.3. QLoRA: compose a frozen grid with a trainable delta	369
17.5. Choose by the bill you need to reduce	370
17.6. The next question: what deserves the update?	372
17.7. Okay, so — adaptation has three separate bills	372
Sources and further reading	372
Exercises	373
18. Alignment and RL Fine-Tuning	375
18.1. Same model, different supervision	376
18.2. A preference is a measurement, not a value	377
18.3. Train a judge, then try to please the judge	380
18.3.1. The exact finite-response policy	382
18.3.2. PPO in outline: a second anchor	383
18.4. When the proxy becomes the target	385
18.5. DPO removes a stage, not the assumptions	391
18.6. Choose by the feedback and exploration you have	394
18.7. Alignment is an evaluation contract	394
18.8. The next question: where did the generator come from?	396
18.9. Okay, so — where to update and what to optimize are separate	396
Sources and further reading	397
Exercises	397
19. Generative Models: From Codes to Samples	399
19.1. Put probability around the code	400
19.1.1. The ELBO is an exact gap identity	400
19.1.2. Move randomness outside the learned map	404
19.2. An adversarial detour: let the judge move	404
19.3. Destroy data on purpose	407
19.4. Learn the path back	409
19.4.1. Why the denoiser needs time	412
19.5. From a scalar denoiser to structured data	418
19.6. A generator needs an evaluation contract too	419
19.7. Okay, so — generation needs a sampling contract	421
Sources and further reading	421
Exercises	422

20. Multimodal Learning: One Space, Two Views	423
20.1. Chapter 2 returns: scores become weights	424
20.1.1. What counts as a negative?	425
20.2. Make the pair relation learnable	426
20.3. Retrieval is not generation	435
20.3.1. Zero-shot classification is retrieval over descriptions	435
20.4. Shortcuts enter through the pair definition	436
20.4.1. A minimum evaluation contract	436
20.5. Okay, so — two towers learn a comparison, not a world model	437
Sources and further reading	437
Exercises	438
Epilogue: The Question Is Yours	439
The ladder we climbed	439
The choices no architecture makes for you	441
Learning about learning	441
From fitting the past to evaluating futures	442
Roads this book did not take	444
Sources and further reading	444
One question, now with better follow-ups	444
Appendices	445
A. Linear Algebra and the SVD	445
A.1. Read the map from its dimensions	445
A.1.1. One vector on paper, a row batch in PyTorch	446
A.2. Span, rank, and orthogonality	447
A.3. Solve the problem you actually have	448
A.4. Eigenvectors are a square-matrix special case	450
A.5. SVD: two spaces and one ordered scale	450
A.5.1. Rank-one layers and truncation	451
A.6. Batched linear algebra	454
A.7. PCA is SVD after centering	455
A.8. Practical <code>torch.linalg</code> checklist	457
Sources and further reading	458
Exercises	458
B. Tensors in Practice	460
B.1. The six-part tensor contract	460
B.2. The book’s shape dictionary	461
B.2.1. The <code>nn.Linear</code> storage convention	462
B.3. Broadcasting: convenient, but not semantic	463
B.3.1. <code>expand</code> and <code>repeat</code> are different promises	464
B.4. Shape, stride, and contiguity	465
B.5. Products: which axes meet?	467
B.6. Indexing and masks	469
B.7. Dtype- and device-aware construction	470
B.8. A debugging routine	471
B.9. Where the version-fragile details live	472

Table of contents

Sources and further reading	472
Exercises	472
C. Numerical Precision and Hardware Efficiency	474
C.1. One operation, several precision contracts	474
C.2. What a binary float can represent	475
C.3. Rounding is part of the algorithm	477
C.3.1. A nonzero update can become no update	477
C.3.2. Stable algebra prevents avoidable overflow	479
C.4. Mixed precision is a policy	480
C.5. Work is not time: the Roofline model	482
C.5.1. Matrix multiplication earns reuse	484
C.6. FlashAttention through the Roofline lens	485
C.6.1. The quick recap: tile, normalize online, and fuse	486
C.7. A measurement contract	489
Sources and further reading	490
Exercises	490
D. Notation	492
D.1. Typeface tells you what kind of object to expect	492
D.2. Decorations carry a second message	493
D.3. Indices and dimensions: a compact dictionary	494
D.4. The book's recurring shape contracts	495
D.5. Probability and optimization symbols are role labels	495
D.6. A quick notation audit	496
E. Statistical Learning Contracts	497
E.1. One dataset, two risks	497
E.2. Name the distribution that carries the claim	498
E.3. A loss is a conditional probability contract	498
E.3.1. Why the Gaussian is plausible, and where it is not	499
E.4. Three ways to compare distributions	499
E.5. What the code estimates	500
E.6. Standard deviation is not standard error	501
Sources and further reading	502
Exercises	502

Preface

A first-principles course in Python and PyTorch.

Much of modern deep learning can be understood as familiar machinery, made learnable and placed inside the right structure.

What to learn. What to build in. What to reuse.

Take a linear model and let it learn its own nonlinear features: the core of a multilayer perceptron. Take fixed convolutional filters and learn them from data: the core of a convolutional network. Take similarity-weighted averaging and learn how to compare and mix: the core of attention. Learn a representation once and adapt it many times: the pretrained era.

The underlying mathematical move, in its simplest instance:

$$\underbrace{f_{\mathbf{w}}(x) = \mathbf{w}^\top \phi(x)}_{\text{linear model on fixed features}} \longrightarrow \underbrace{f_{\mathbf{w},\theta}(x) = \mathbf{w}^\top \phi_\theta(x)}_{\text{linear readout on learned nonlinear features}} .$$

What we build in — through architecture, objectives, data, or training — is **inductive bias**: regularities the model should not have to rediscover from scratch. Three questions organize everything that follows:

What is learned? features, filters, comparisons, representations

What is built in? locality, sharing, order, invariance, visibility

What is reused? pretrained representations and models

MLPs, convolutional networks, attention, and pretraining are examples of this framework, not separate inventions, and each part of the book returns to the same design question in a new regime.

This book develops those moves by construction and experiment. We build the simplest mechanism, test where it fails, and let the failure motivate the next design. By the end, modern AI should feel less like a zoo of architectures and more like a small set of ideas — made learnable, shaped by structure, and reused at scale.

Make the right parts learnable. Build in the structure you trust. Reuse what transfers.

i About this edition

This is the free, open book for [DS 6050 Deep Learning](#) at UVA’s School of Data Science. The course arc is published, and the book continues to receive corrections and teaching improvements. The current stable edition is **v1.2.1 (August 2, 2026)**; its [archived release and PDF](#) preserve a fixed version. The live site is a rolling post-v1.2.1 build. The HTML edition is canonical; the PDF is a derived print conversion. Executable figures and numerical results are produced by code in the book’s source: experiments carry their executable code, revealed step by step or all at once, and concept diagrams fold theirs. Every printed snippet is the executed artifact — nothing on the page

is retyped — and the repository’s checks re-execute the book’s notebooks. Source, provenance, and licenses remain visible in the repository.

For the graduate continuation—from numerical and hardware contracts through high-dimensional probability, random-matrix spectra, and optimization dynamics—see [Deep Learning: Making It Trainable](#).

Suggested citation: Shakeri, Heman. 2026. *Deep Learning: Making It Learnable*. Version 1.2.1. <https://shakeri-lab.github.io/dl-book/>.

How to read this book

Every chapter follows the course’s learning loop: **read** → **predict** → **run** → **audit**. Before running an experiment, write down the direction you expect; after running it, audit the code the way you would audit a colleague’s — or an AI assistant’s. Five questions cover most audits: what function class does this code implement, what objective does it optimize, what estimator does each batch compute, what information is visible to the model (masks, padding, order), and what claim does the output support? The code is written directly in Python and PyTorch and runs on an ordinary laptop CPU — small data, honest experiments. Watch the [lecture videos](#), read the chapter, run the cells, and test yourself against the self-checks on the [course site](#).

Course route and dependencies

The route is cumulative. Modules sometimes revisit a chapter because the course uses the same idea first as a mechanism and later as evidence. The appendices are just-in-time support, not a separate prerequisite block: **A** is linear algebra, **B** is tensors, **C** is precision and performance, **D** is notation, and **E** is statistical learning contracts.

Course module	Primary book route	Supporting appendices
1 · foundations and MLPs	1 → 2 → 3 → 4	A · B · D · E
2 · backpropagation	5	B · D
3 · optimization and ablation	4 → 6 → experiment interlude	B · D · E
4 · convolutional networks	6 → 7 → 8	B · D · E
5 · modern CNNs and transfer	9	B · C · D
6 · PCA and autoencoders	PCA → autoencoder interlude	A · B · D
7 · recurrent sequences	10 → 11	B · D
8 · attention	12 → 13	A · B · D
9 · Transformers	14 → test-time-regression interlude	B · C · D
10 · pretrained models, ViTs, scaling	15 → 16	B · C · D
11 · adaptation and alignment	17 → 18	B · C · D · E
12 · generative and multimodal models	19 → 20	A · B · C · D · E

Plan a reading session

Time depends more on whether you stop to derive and run than on page count. As a planning range, allow roughly **45–90 minutes for a careful first read** of most chapters, with the experiment-heavy later chapters often taking the longer end or a second sitting. Once the environment and data are ready, reproducing a few key cells commonly adds **20–45 minutes**; rerunning every training cell from scratch can take substantially longer and varies by machine. Exercises are best treated as their own work session. These are orientation bands, not completion-time promises.

If you have one short sitting, read the opening problem, study the figures, run one cell that tests the main claim, and answer any closing **Check yourself** prompts without looking back. Return for the derivations and exercises when you can explain what failed, what changed, and what evidence supports the repair.

Exercises, hints, and solutions

The book supports both enrolled and self-paced readers. For-credit assignment rules on the course site and in the syllabus take precedence. To protect those assignments, the public book does not publish full worked solutions to overlapping graded work. Public **Check yourself** prompts are retrieval practice, not answer banks; the chapter recap and course-site self-check feedback provide the first layer of hints.

For self-paced work, use these acceptance criteria:

- A **conceptual answer** names the mechanism, predicts a direction or limiting case, and states one condition under which the claim could fail.
- A **shape derivation** labels every axis before the operation and checks the resulting shape against the code.
- A **code exercise** runs from a fresh session, fixes its seed, prints or asserts the important shapes, and reproduces the requested qualitative relationship. Small machine-dependent numerical differences are not failures unless the exercise sets a tolerance.
- An **experiment** states the question before the run, separates tuning from final evaluation, keeps the comparison recipe fixed, reports seed variation when requested, and ends with a limitation rather than a universal claim.

When stuck, inspect the nearest **Trap:** or tip callout, write the expected shape or trend before running anything, and reduce the code to one batch or one example. Enrolled students should use the course's approved help channels for assignment-specific hints; complete graded solutions stay there rather than in the public text.

Acknowledgments

To the students of DS 6050, whose questions shaped every explanation here.

Support this open book

This book is free to read and download at **\$0**, and no contribution unlocks additional content. Reading it, sharing it, or reporting a correction is already meaningful support. If it has been useful and you would like to help sustain ongoing corrections, new figures, and open releases, you may make an optional contribution here:

Support the open book: buymeacoffee.com/hshakeri

Text and figures licensed [CC BY-NC-SA 4.0](#); all code [MIT](#). Source on [GitHub](#). The graduate companion is [Deep Learning: Making It Trainable](#).

Part I.

From Lines to Networks

1. Linear Regression, the Mother Model

The core task of everything we will do in this course is to teach a computer to learn a function from examples. Before we can appreciate what is *deep* about deep learning, we need one complete, honest example of learning: a model, a loss, and an update rule. Linear regression is that example. It is the smallest model that contains the entire supervised learning story, and by the end of this chapter we will build it three ways: in closed form, by gradient descent from scratch, and with PyTorch modules. All three will agree, and you will know exactly why.

1.1. Learning from examples

We start with a dataset of examples,

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n,$$

where each $\mathbf{x}_i \in \mathbb{R}^d$ is an input with d features and y_i is the desired output, aka the *supervision signal*. Stack the inputs as rows and you get the blue data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$: n samples down, d features across, with the purple targets collected in $\mathbf{y}$.

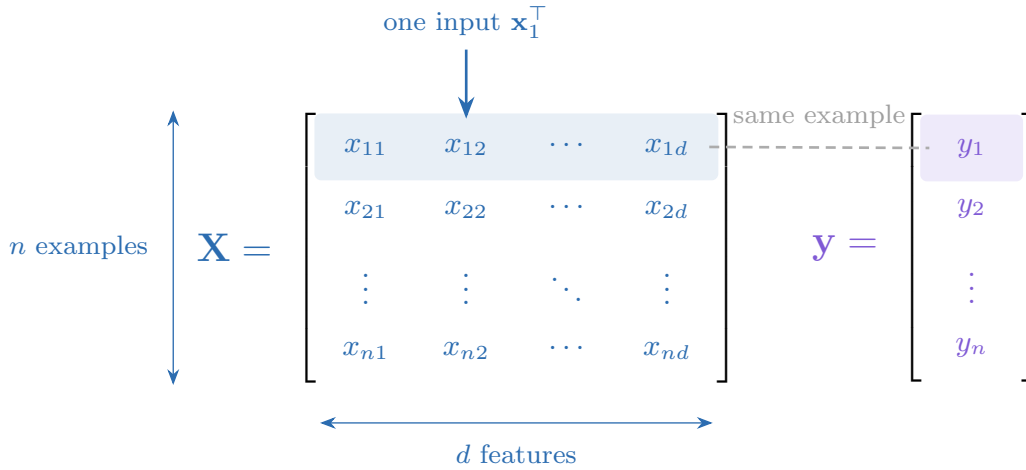


Figure 1.1.: Stacking examples turns a dataset into two aligned objects: one row of $\mathbf{X}$ per input and one entry of $\mathbf{y}$ per desired output. Color will keep these roles visible in the equations that follow.

Our goal is a flexible function f such that

$$f(\mathbf{x}_i) \approx y_i \quad \text{and, crucially, } f \text{ generalizes to unseen data.}$$

That second clause is the whole game. We do not care how well f recites the training examples; we care how it behaves on a new $\mathbf{x}$ it has never seen, one drawn from the same distribution as the training data.

In statistical notation, training minimizes the **empirical risk**

$$\widehat{R}_{\mathcal{D}}(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i),$$

while the quantity we ultimately care about is the **population risk**

$$R_P(\theta) = \mathbb{E}_{(\mathbf{X}, \mathbf{Y}) \sim P}[\ell(f_{\theta}(\mathbf{X}), \mathbf{Y})].$$

We will assume this distinction is familiar and use it precisely. An independent validation set drawn from P estimates performance under P ; it does not automatically estimate performance under a transformed or deployment distribution Q . Chapter 6 will make that difference visible; Appendix E later gathers the full statistical contract in one reference.

i Supervised learning, three flavors

The range of y decides the task and, as we will see, the natural loss:

Task	Range of y	Typical loss
Regression	$\mathbb{R}$	Mean squared error
Classification	$\{0, \dots, K-1\}$	Cross-entropy
Structured prediction	sequences, images, graphs	task-specific

This chapter is regression; Chapter 2 handles classification with the same machinery.

1.1.1. Why is this hard?

This task sounds simple, but it is fundamentally hard: for any finite set of examples, there are *infinitely many* functions that fit them perfectly. We want the one that is most suited to our problem $\rightarrow$ the learning algorithm needs a nudge in the right direction. That guiding assumption is called its **inductive bias**, and it is a thread we will pull on for the rest of the book.

- A **linear model** has a *strong* bias: it assumes the world is linear. Simple and stable, but systematically wrong when the data is not linear (high bias, low variance).
- A **deep neural network** has a *weak* bias. Its flexibility lets it learn complex patterns, but it can memorize the training data instead of the underlying rule (low bias, high variance). Some networks can memorize an entire training set and then perform poorly at deployment.

The key to modern deep learning is to use a flexible model and fight overfitting with data, regularization, and (the deepest idea of all) inductive biases matched to the task. That last idea gets its own chapter (Chapter 6).

To make any of this concrete, we parameterize a family of functions $f_{\mathbf{w}}$ by a set of tuning parameters $\mathbf{w}$. Think of each parameter as a knob. *Learning* is finding a good setting $\widehat{\mathbf{w}}$ of the knobs; *inference* is using $f_{\widehat{\mathbf{w}}}$ to predict on unseen data. That is the vocabulary we will use all book.

1.2. The linear model and its geometry

Linear regression assumes a linear relationship:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

The second term, b , is the model's default prediction when the input carries no relevant information. The first term, $\mathbf{w}^\top \mathbf{x}$, adjusts that baseline up or down based on how the input matches the weight vector.

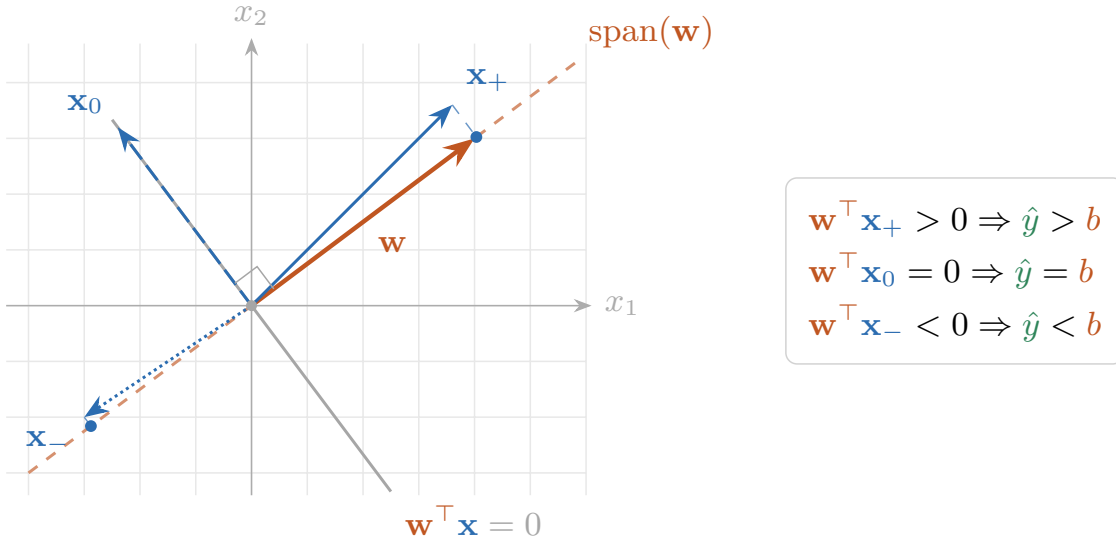


Figure 1.2.: The dot product measures the signed shadow of an input along the weight vector. Aligned inputs raise the prediction above b , orthogonal inputs leave it at b , and opposing inputs lower it.

Now we can name what the figure shows: **a dot product is a similarity score**. More precisely,

$$\mathbf{w}^\top \mathbf{x} = \|\mathbf{w}\|_2 \|\mathbf{x}\|_2 \cos \theta.$$

For normalized vectors it measures directional similarity directly: positive when $\mathbf{x}$ points along $\mathbf{w}$, zero when they are orthogonal, and negative when they oppose. Without normalization, the two norms also control the score, so a large dot product can mean strong alignment, large magnitude, or both. A linear model is therefore a learned *pattern scorer*: $\mathbf{w}$ is a template whose direction and scale both matter. In Part II, an image filter will slide this same dot product across an image. In Part IV, attention will build learned similarities between queries and keys from the same primitive.

For compact matrix notation, we will absorb b into the weights.¹ Then

$$\hat{y} = \mathbf{w}^\top \mathbf{x} \quad (\text{bias included}).$$

¹Append a column of ones to $\mathbf{X}$ and append b to $\mathbf{w}$. This is often called *bias augmentation* or the use of *homogeneous coordinates*. It is bookkeeping, not a new learning principle. A formula or software layer without an explicit bias may instead be deliberately constraining $b = 0$.

1.3. The loss: what makes a hyperplane “bad”?

To find the best hyperplane we must first quantify error. Write the residual on example i as $e_i = y_i - \hat{y}_i$. The standard choice for regression is the **mean squared error (MSE)**:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n e_i^2 = \frac{1}{n} \|\mathbf{e}\|_2^2, \quad (1.1)$$

$$\mathbf{e} = \mathbf{y} - \mathbf{X}\mathbf{w}.$$

The wine residual names each miss; squaring and averaging those misses produces one neutral scalar. The loss is the guiding principle of training: it translates model error into a score of failure, giving the optimizer a precise measure of how much corrective work remains. The choice of loss is not arbitrary, and we will justify this one at the end of the chapter.

With two parameters, (w, b) , this scalar becomes a surface over the parameter plane. That surface is the **loss landscape**. Learning means finding a low point on it; the next two methods differ in whether they solve for that point directly or approach it step by step.

1.4. Finding the best weights, method 1: solve it exactly

For this one special problem we can characterize the exact minimizer. Set the gradient of Equation 1.1 to zero and you get the **normal equations**:

$$\mathbf{X}^\top \mathbf{X} \hat{\mathbf{w}} = \mathbf{X}^\top \mathbf{y}. \quad (1.2)$$

Four facts keep this equation honest without detouring into a linear-algebra treatise:

- **Full column rank.** The minimizer is unique, and only then may we write $\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$.
- **Rank deficiency.** The fitted prediction is still a unique projection, but the parameter vector need not be unique. The minimum-norm choice is $\hat{\mathbf{w}}_{\min} = \mathbf{X}^+ \mathbf{y}$, where $\mathbf{X}^+$ is the Moore–Penrose pseudoinverse. This matters in deep learning because $d > n$ is common, not exceptional.
- **Conditioning.** Forming the normal equations squares the spectral condition number: $\kappa_2(\mathbf{X}^\top \mathbf{X}) = \kappa_2(\mathbf{X})^2$. QR-, SVD-, or library-based least-squares routines are the computational default.
- **Cost.** For a tall dense matrix, forming $\mathbf{X}^\top \mathbf{X}$ costs $O(nd^2)$, followed by about $O(d^3)$ for factorization. Either term may dominate.

There is a picture behind these facts worth keeping. Every prediction can be written as a weighted combination of the feature columns:

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}} = \sum_{j=1}^d \hat{w}_j \mathbf{X}_{:,j}.$$

It must therefore live in the **column space** of $\mathbf{X}$. If $\mathbf{y}$ lies outside that space, the best prediction is its projection. The residual $\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}}$ is perpendicular to the column space, so no adjustment of the weights can reduce it further.

The same projection can be read from the other side:

$$\hat{y}_i = \sum_{j=1}^n (\mathbf{X}\mathbf{X}^+)_{ij} y_j.$$

Each fitted value is a weighted combination of the training targets y_j . Part IV will turn this same idea into attention by choosing mixing weights through similarity.

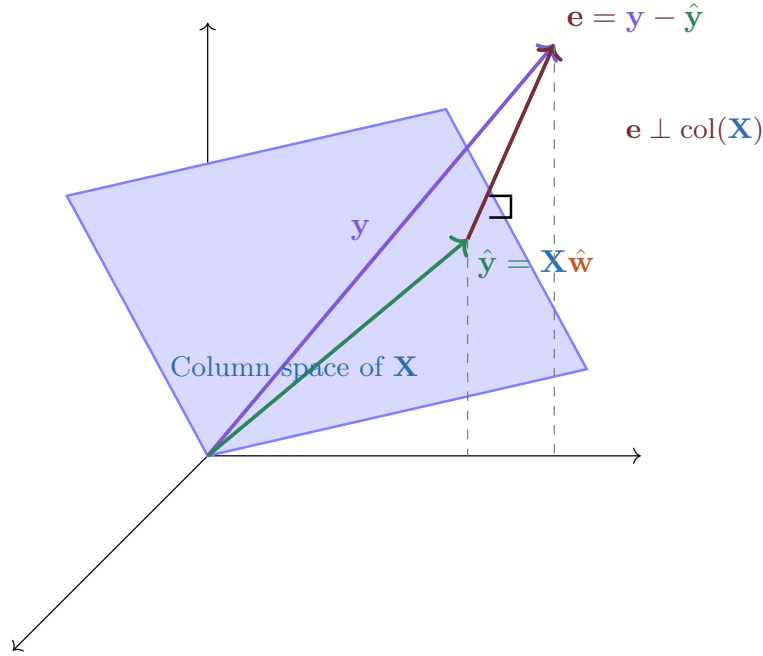


Figure 1.3.: The normal equations, geometrically: the optimal prediction $\hat{\mathbf{y}} = \mathbf{X}\hat{\mathbf{w}}$ is the projection of $\mathbf{y}$ onto the column space of $\mathbf{X}$; the residual $\mathbf{e}$ is orthogonal to it.

The exact characterization is elegant, but deep models have millions or billions of parameters and nonlinear compositions. There is no comparable closed form to solve. We need another way.

1.5. Finding the best weights, method 2: walk downhill

Here is the geometric intuition. Picture the loss as a landscape over the parameter space, and imagine standing on it *blindfolded*, trying to find the lowest valley. You cannot see the valley, but you can feel the slope under your feet. So you feel the slope, take a small step downhill, and repeat.

The mathematical form of this hill-descending intuition is **gradient descent**:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^{(t)}), \quad (1.3)$$

where the *learning rate* η sets the step size. For the MSE loss the gradient has a clean closed form. With $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{w} \in \mathbb{R}^d$, and $\mathbf{y} \in \mathbb{R}^n$:

1. Linear Regression, the Mother Model

$$\begin{aligned}\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) &= -\frac{2}{n} \mathbf{X}^\top \mathbf{e} \\ &= \frac{2}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \in \mathbb{R}^d.\end{aligned}\tag{1.4}$$

PLAN

1. Create a two-parameter regression problem and its loss surface.
 2. Follow the gradient from a deliberately poor starting point.
 3. Show the same landscape in perspective and from overhead.
-

CODE

```
import torch
import numpy as np
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
x1 = torch.randn(60)
y1 = 2.5 * x1 - 1.0 + 0.3 * torch.randn(60)

def mse(w, b):
    return ((y1 - (w * x1 + b)) ** 2).mean()

W, B = np.meshgrid(
    np.linspace(-1.5, 5.5, 100),
    np.linspace(-4.0, 3.0, 100),
)
L = np.vectorize(lambda wi, bi: float(mse(wi, bi)))(W, B)

# [2]
w, b, path = -0.5, 2.0, []
for _ in range(20):
    path.append((w, b))
    err = (w * x1 + b) - y1
    w -= 0.25 * float(2 * (err * x1).mean())
    b -= 0.25 * float(2 * err.mean())

pw, pb = zip(*path)
path_loss = [float(mse(wi, bi)) for wi, bi in path]

# [3]
fig = plt.figure(figsize=(9.2, 3.8))
ax3d = fig.add_subplot(1, 2, 1, projection="3d")
ax3d.plot_surface(W, B, L, cmap="Blues", alpha=0.78, linewidth=0)
ax3d.plot(pw, pb, path_loss, "o-", color="#E57200", ms=3, lw=1.5)
```

```

ax3d.set(xlabel="$w$", ylabel="$b$", zlabel="MSE")
ax3d.view_init(elev=27, azim=-58)

ax2d = fig.add_subplot(1, 2, 2)
ax2d.contour(W, B, L, levels=25, cmap="Blues", alpha=0.85)
ax2d.plot(pw, pb, "o-", color="#E57200", ms=4, lw=1.5,
          label="descent path")
ax2d.plot(2.5, -1.0, "k*", ms=12, label="data-generating parameters")
ax2d.set(xlabel="$w$", ylabel="$b$")
ax2d.legend(fontsize=8)
plt.tight_layout()
plt.show()

```

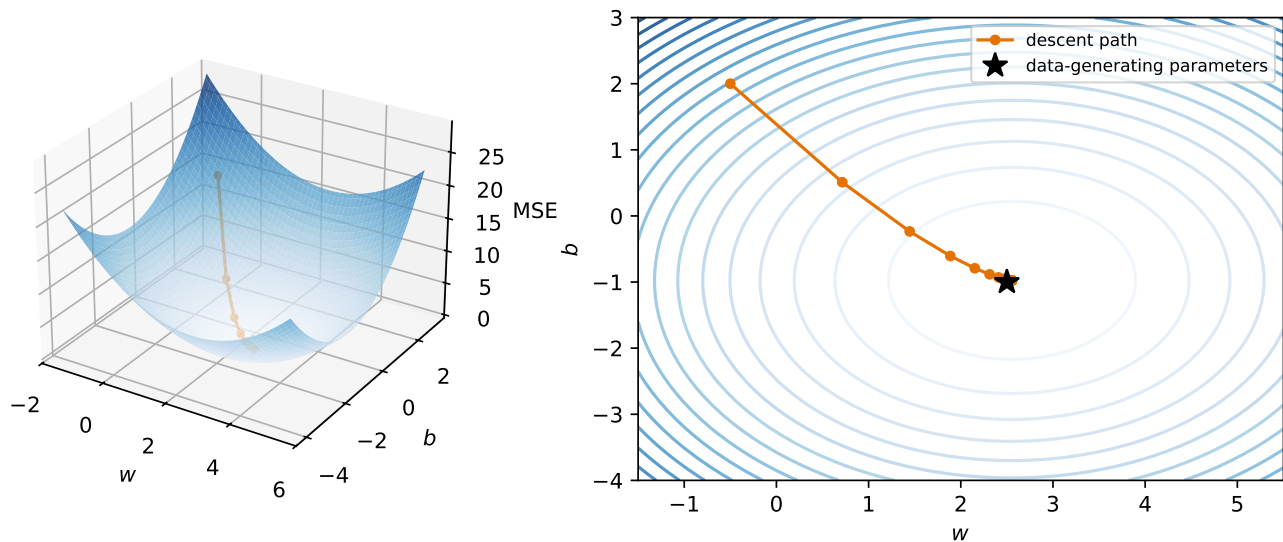


Figure 1.4.: One loss landscape, two views. The surface makes the scalar height of MSE visible; the contour view makes the update path legible. Gradient descent needs only the local slope, not a view of the whole bowl.

This iterative loop (predict $\rightarrow$ measure the loss $\rightarrow$ feel the gradient $\rightarrow$ step) is exactly what modern deep learning frameworks automate, at billion-parameter scale. When the dataset itself is huge we will not even use all of it per step: a small random *batch* gives a good-enough gradient. That refinement (stochastic gradient descent) matters enough to get its own treatment in [Chapter 4](#).

1.6. Build the complete model three ways

The two-parameter loop above was deliberately a one-off drawing instrument: it exposed the loss landscape and the update path. Now we will fit one reusable, d -feature regression model to the same dataset three complete ways, first with the algebra exposed and then with the framework abstractions we will reuse.

💡 Coding hygiene

Two habits are worth forming from the first cell: **fix seeds** so runs are reproducible, and **annotate functions with types** so shapes and interfaces stay readable. The cells in this book are terse teaching kernels, not production systems; they expose the mechanism and the checks needed for the claim.

PLAN

1. Load the chapter dependencies and establish reproducible state.

CODE

```
import torch
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
```

<torch._C.Generator at 0x133220d70>

First, synthetic data. We control the ground truth (the true weights and bias), so we can check whether each method actually recovers them.

PLAN

1. Define the reusable `make_synthetic_data` helper.
2. Generate noisy targets from known weights, then check the batch shapes.

CODE

```
# [1]
def make_synthetic_data(
    weights: torch.Tensor, bias: float, n_samples: int, noise: float = 0.1
) -> tuple[torch.Tensor, torch.Tensor]:
    """y = Xw + b + noise. Returns X: (n, d) and y: (n,)."""
    X = torch.randn(n_samples, len(weights))
    y = X @ weights + bias + noise * torch.randn(n_samples)
    return X, y

# [2]
true_w, true_b = torch.tensor([2.0, -3.4]), 4.2
X, y = make_synthetic_data(true_w, true_b, n_samples=200)
X.shape, y.shape          # always check your shapes
```

```
(torch.Size([200, 2]), torch.Size([200]))
```

1.6.1. Take 1: the closed form

On the augmented matrix, with a column of ones carrying the bias, use the rank-aware least-squares routine directly:

PLAN

1. Append a constant feature to represent the bias.
 2. Solve least squares with a rank-aware library routine.
 3. Compare the estimate with the planted parameters.
-

CODE

```
# [1]
X_aug = torch.cat([X, torch.ones(X.shape[0], 1)], dim=1)

# [2]
w_ols = torch.linalg.lstsq(X_aug, y).solution

# [3]
print(f"closed form:      w = {w_ols[:2].numpy().round(3)}, b = {w_ols[2]:.3f}")
print(f"ground truth:    w = {true_w.numpy()}, b = {true_b}")
```

```
closed form:      w = [ 2.01 -3.402], b = 4.196
ground truth:    w = [ 2. -3.4], b = 4.2
```

Two lines, and we recover the truth up to the noise floor. `lstsq` handles rank and conditioning more honestly than explicitly forming $\mathbf{X}^\top \mathbf{X}$. Now for the method that extends to nonlinear models.

1.6.2. Take 2: gradient descent, from scratch

We write the model as a small class: a forward pass, manually derived gradients (Equation 1.4), and an update step. No autograd yet; we want to feel the mechanics once with our own hands.

PLAN

1. Store the parameters and define the linear prediction.
 2. Convert batch residuals into the exact MSE update.
 3. Repeat that update until the loss settles.
 4. Compare the learned parameters with the other solution.
-

CODE

```
# [1]
class LinearRegressionScratch:
    """Linear regression trained with manually derived gradients."""

    def __init__(self, input_size: int, lr: float = 0.1):
        self.w = 0.01 * torch.randn(input_size)
        self.b = torch.zeros(1)
        self.lr = lr

    def forward(self, X: torch.Tensor) -> torch.Tensor:
        return X @ self.w + self.b

    def step(self, X: torch.Tensor, y: torch.Tensor) -> float:
        # [2]
        err = self.forward(X) - y          # 1. predict, 2. measure (n,)
        self.w -= self.lr * 2 * X.T @ err / len(y) # 3. feel the slope,
        self.b -= self.lr * 2 * err.mean()         # 4. step downhill
        return float((err ** 2).mean())

# [3]
model = LinearRegressionScratch(input_size=2)
losses = [model.step(X, y) for _ in range(100)]

# [4]
print(f"gradient descent: w = {model.w.numpy().round(3)}, b = {model.b.item():.3f}")
```

```
gradient descent: w = [ 2.01 -3.402], b = 4.196
```

1.6.3. Take 3: the framework way

Finally, the idiom you will use for every model from here on: `nn.Linear` holds the knobs, `autograd` feels the slope for us, and the optimizer takes the step.

i A deliberate framework preview

This cell shows the complete PyTorch training loop so you can recognize its shape. We use the three framework calls as black boxes here: [Chapter 3](#) introduces modules, [Chapter 4](#) opens the optimizer, and [Chapter 5](#) explains what `backward()` computes. Until then, the manual implementation above is the chapter's working toolbox.

PLAN

1. Pair a linear module with MSE and an SGD optimizer.
 2. Repeat the framework forward, backward, and update sequence.
 3. Detach and report the learned parameters.
-

CODE

```
from torch import nn

# [1]
net = nn.Linear(in_features=2, out_features=1)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
loss_fn = nn.MSELoss()
nn_losses = []

# [2]
for _ in range(100):
    optimizer.zero_grad()          # clear old gradients: they accumulate!
    loss = loss_fn(net(X).squeeze(-1), y)  # (n, 1) -> (n)
    nn_losses.append(float(loss.detach()))
    loss.backward()                # autograd feels the slope for us
    optimizer.step()

w_nn = net.weight.detach().squeeze()
b_nn = net.bias.item()
# [3]
print(f"nn.Linear:          w = {w_nn.numpy().round(3)},  b = {b_nn:.3f}")
```

```
nn.Linear:          w = [ 2.01  -3.402],  b = 4.196
```

The three methods now support the same diagnostic:

PLAN

1. Collect the direct-solve and iterative loss records.
-

CODE

```
# [1]
closed_losses = [
    float((y ** 2).mean()),
    float(((X_aug @ w_ols - y) ** 2).mean()),
]
```

1. Linear Regression, the Mother Model

```
records = [closed_losses, losses, nn_losses]
titles = ["closed form", "scratch gradient descent", "PyTorch module"]
```

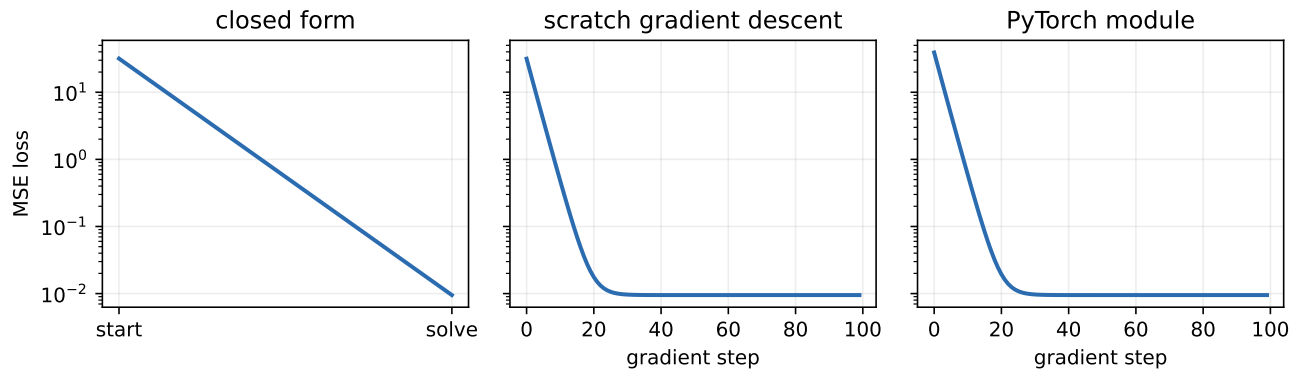


Figure 1.5.: Three routes to the same loss floor. The closed form is a before-and-after solve, not an optimization trajectory; both gradient methods expose the sequence of corrective steps.

Loss-versus-step plots will recur throughout the book. They reveal whether an optimization process is descending, stalling, or diverging, and they make competing update rules comparable. They do not, by themselves, establish generalization.

Three implementations, one answer:

PLAN

1. Choose a one-dimensional slice through the two-feature data.
2. Evaluate each learned model along that same slice.
3. Plot the adjusted observations and all three fits.

CODE

```
# [1]
grid = torch.linspace(X[:, 0].min(), X[:, 0].max(), 50)
x2_mean = X[:, 1].mean()

# [2]
def prediction_slice(weights: torch.Tensor, bias: float) -> torch.Tensor:
    """Predictions along x1 while x2 is held at its sample mean."""
    return weights[0] * grid + weights[1] * x2_mean + bias

y_on_slice = y - true_w[1] * (X[:, 1] - x2_mean)

plt.figure(figsize=(5.5, 3.6))
# [3]
plt.scatter(X[:, 0], y_on_slice, s=12, alpha=0.4, label="data adjusted to slice")
```

```
plt.plot(grid, prediction_slice(w_ols, float(w_ols[2])), lw=3, label="closed form")
plt.plot(grid, prediction_slice(model.w, model.b.item()), "--", lw=2,
         label="gradient descent")
plt.plot(grid, prediction_slice(w_nn, b_nn), ":", lw=2, label="nn.Linear")
plt.xlabel("$x_1$"); plt.ylabel("$y$"); plt.legend()
plt.tight_layout()
plt.show()
```

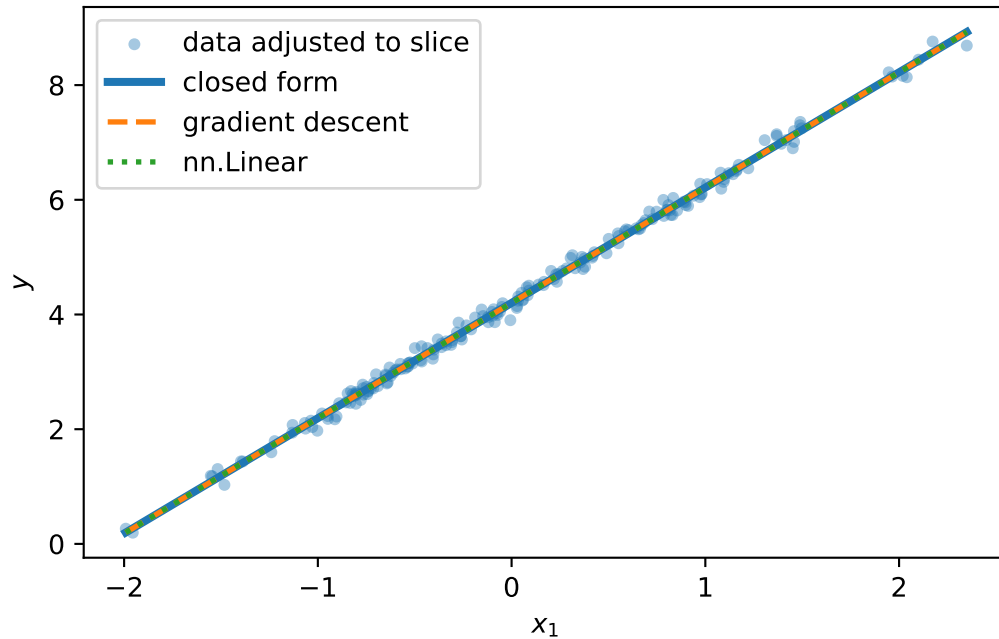


Figure 1.6.: All three methods find the same line on the slice $x_2 = \bar{x}_2$. Because this is synthetic data, the points can be adjusted to that same slice using the known true coefficient; line and points now show the same question.

The computation shared by all three implementations can also be drawn as a small circuit:

The next chapter will keep this entire circuit and add one operation after its output. That small addition will change what the model can represent without changing the weighted-sum machinery underneath.

1.7. Why squared error? The maximum-likelihood view

Mean squared error is not an arbitrary penalty. Assume targets are generated by a linear process with additive Gaussian noise.²

²This is more than algebraic convenience. A residual often aggregates many small, unobserved effects, such as sensor variation, environmental fluctuations, omitted features, and rounding. Under the conditions of the central limit theorem, sums of many independent or weakly dependent finite-variance effects become approximately Gaussian. There is also an informational argument: among continuous distributions with a fixed mean and variance, the Gaussian has maximum differential entropy, so it adds no further structure beyond those two constraints. Neither argument makes every residual Gaussian; the assumption is a useful model to check, not a law of nature.

1. Linear Regression, the Mother Model

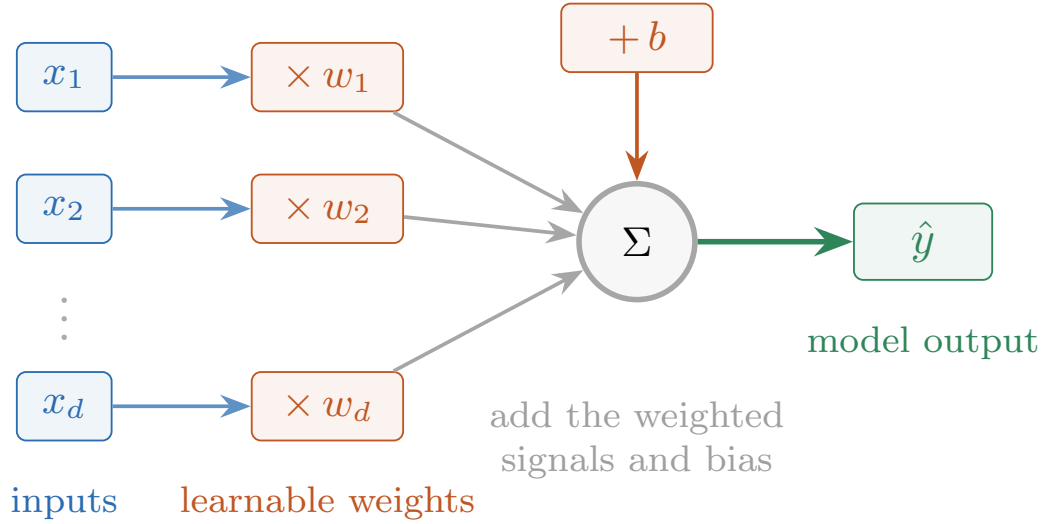


Figure 1.7.: A linear model as a computation circuit: each blue input is scaled by an orange learnable weight, the signals and orange bias are added, and the green prediction leaves the circuit.

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

The conditional probability of observing y given $\mathbf{x}$ is therefore a Gaussian centered at the deterministic prediction $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$:

$$p(y \mid \mathbf{x}; \mathbf{w}, b) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \hat{y})^2}{2\sigma^2}\right).$$

Maximum likelihood asks which $(\mathbf{w}, b)$ make the observed dataset most probable. Independence turns the dataset likelihood into a product; the logarithm turns that product into a sum and brings the Gaussian exponent down:

$$\log \mathcal{L}(\mathbf{w}, b) = C - \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - \hat{y}_i)^2.$$

Maximizing this log-likelihood is exactly minimizing the sum of squared residuals. A loss function is a probability model in disguise: it states how targets may vary around a prediction, then takes the negative log-likelihood.

This is the master pattern that continues in [Chapter 2](#). Continuous targets suggest a Gaussian likelihood and MSE; binary and categorical targets suggest Bernoulli and categorical likelihoods and cross-entropy. A Laplace likelihood would instead produce absolute error. Choosing a loss is therefore a modeling decision, not memorizing a lookup table.

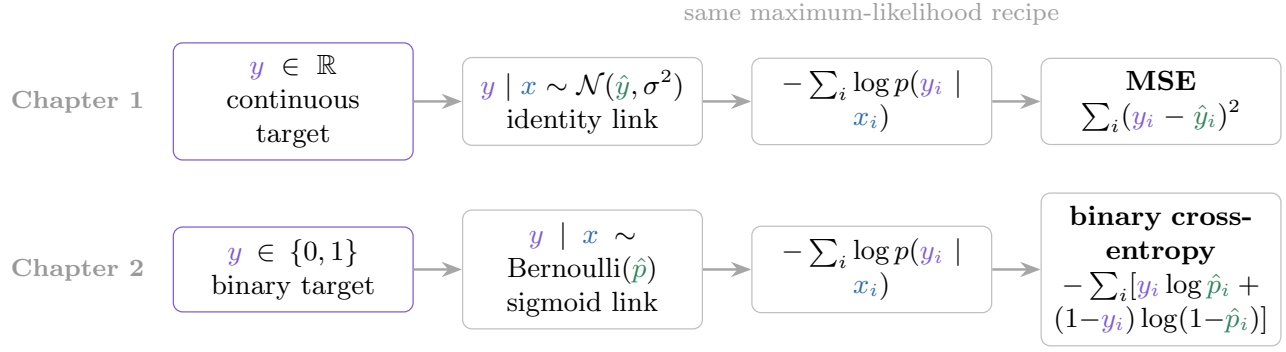


Figure 1.8.: The target type changes the conditional distribution and link function, but the maximum-likelihood recipe stays fixed. Gaussian continuous targets yield MSE; Bernoulli binary targets yield binary cross-entropy.

1.8. A first look at bias and variance

Keep the fitting recipe fixed, but collect the data again. The observed points move, so the fitted line moves too. At one fixed input, the green distribution of predictions has a center and a spread. Its center can miss the black data-generating curve, and fresh outcomes still vary around that curve even if the prediction were perfect. Keep the output axis fixed between the first two panels: the middle panel is a literal vertical slice through the left one.

Now we can name what the picture separated. Our learned predictor $\hat{f}_{\mathcal{D}}$ depends on the random training set $\mathcal{D}$. Let $f^*(\mathbf{x}) = \mathbb{E}[Y | X = \mathbf{x}]$ be the regression function, and write the average learned prediction as $\bar{f}(\mathbf{x}) = \mathbb{E}_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(\mathbf{x})]$. For a fresh outcome independent of the training set once $X = \mathbf{x}$ is fixed, the expected squared error decomposes into three parts:

$$\begin{aligned}
 & \underbrace{\mathbb{E}_{\mathcal{D}, Y} \left[\left(\hat{f}_{\mathcal{D}}(\mathbf{x}) - Y \right)^2 \mid X = \mathbf{x} \right]}_{\text{Expected squared error}} \\
 &= \underbrace{\left(\bar{f}(\mathbf{x}) - f^*(\mathbf{x}) \right)^2}_{\text{Bias}^2} \\
 &+ \underbrace{\mathbb{E}_{\mathcal{D}} \left[\left(\hat{f}_{\mathcal{D}}(\mathbf{x}) - \bar{f}(\mathbf{x}) \right)^2 \right]}_{\text{Variance}} \\
 &+ \underbrace{\text{Var}(Y \mid X = \mathbf{x})}_{\text{Irreducible noise}}.
 \end{aligned} \tag{1.5}$$

The **bias** term measures how far the **average prediction** is from the truth because of limiting assumptions like linearity: a simple model on complex data is systematically wrong (underfitting). The **variance** term measures how much the prediction would change if we collected a fresh training set (the model's sensitivity to sampling noise); a complex model can fit the noise itself (overfitting). The **irreducible noise** belongs to the data-generating process and cannot be learned away.

This is why we split data into training, validation, and test: the model sees the training set, the validation set referees the bias–variance trade-off, and the test set stays untouched until the end.

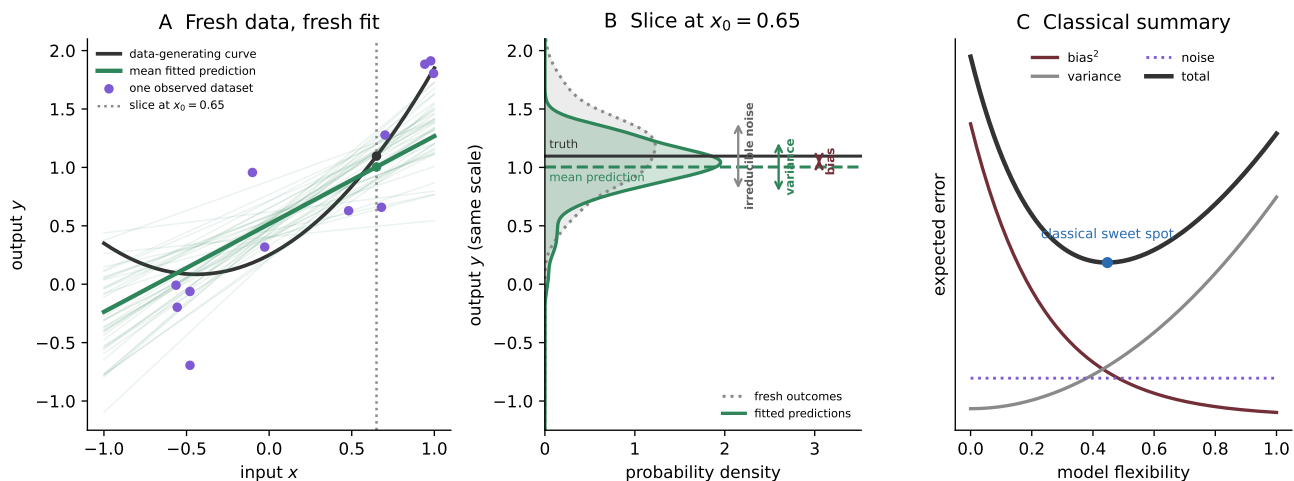


Figure 1.9.: Bias and variance, shown before they are named. Left: forty fits displayed from 200 fresh noisy datasets under one unchanged linear-regression recipe; the dotted line marks the slice at $x_0 = 0.65$. One dataset is purple, the mean prediction is green, and the data-generating curve is black. Middle: the same vertical output scale turns that slice into two sampling distributions. Bias is the distance from truth to mean prediction; the green and gray arrows each span ± 1 sample standard deviation, visualizing the spreads whose squared values enter prediction variance and irreducible noise. Right: the familiar U-curve compresses the three effects into a classical complexity cartoon. This seeded construction reveals the decomposition; it does not claim that every modern model follows one U-shaped path.

⚠ Trap: the U-shaped curve is a cartoon, not a law of nature

The textbook picture, validation error falling then rising as complexity grows, is a helpful first mental model, and you should have it. But bias and variance are not a mechanical see-saw. In classical fixed-dimensional, well-specified settings, variance often falls roughly like $1/n$ as data grows; that rate is not a universal law for modern models. Regularization and inductive biases matched to the task can buy capacity without exploding variance. Modern over-parameterized networks can pass an interpolation threshold and then improve again as capacity grows, producing **double descent** rather than one U. We will study that regime in [Chapter 6](#).

1.8.1. Regularization, briefly

One lever we can pull right now: penalize complexity in the loss itself. **Ridge regression** adds an L_2 penalty,

$$\mathcal{L}_\lambda(\mathbf{w}) = \frac{1}{n} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2, \quad \hat{\mathbf{w}}_{\text{ridge}} = (\mathbf{X}^\top \mathbf{X} + n\lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y},$$

nudging the model toward smaller, more stable weights: a little more bias for a lot less variance.³ The factor n is there because our data-fit term is a *mean*. This displayed solution applies when there is no

³For a small update nearly perpendicular to $\mathbf{w}$, $\Delta\theta \approx \|\Delta\mathbf{w}_\perp\|_2 / \|\mathbf{w}\|_2$: the same nudge rotates a larger vector less. Here L_2 adds $2\lambda\mathbf{w}$, so $\mathbf{w} \leftarrow (1 - 2\eta\lambda)\mathbf{w} - \eta\nabla\mathcal{L}_{\text{data}}$. This radial shrink limits scale and can preserve angular responsiveness in a large layer; it does not by itself explain or cure stalled training.

intercept or when features and targets have been centered so the intercept is recovered separately. We normally do not penalize that intercept. In augmented-matrix notation, replace I by $\text{diag}(1, \dots, 1, 0)$ so the final bias coordinate remains free.

Let us see that, not just assert it. We build a problem designed to punish an unregularized model: 20 features of which only 5 matter, noisy targets, and (the cruel part) barely more samples than parameters. This is exactly the regime where variance explodes and regularization shines:

PLAN

1. Create a small-sample problem with many irrelevant features.
 2. Solve the same data with unregularized and ridge estimators.
 3. Compare both estimates with the planted weights.
-

CODE

```
# [1]
n, d = 25, 20                                # barely more samples than knobs!
w_true = torch.zeros(d)
w_true[:5] = torch.tensor([3.0, -2.0, 1.5, 2.5, -1.0]) # only 5 real features
Xh, yh = make_synthetic_data(w_true, bias=0.0, n_samples=n, noise=2.0)

# [2]
w_ols_h = torch.linalg.lstsq(Xh, yh).solution
lam = 0.4
w_ridge = torch.linalg.solve(Xh.T @ Xh + n * lam * torch.eye(d), Xh.T @ yh)

# [3]
def report(name: str, w_hat: torch.Tensor) -> None:
    err = ((w_hat - w_true) ** 2).mean()
    print(f"{name:>6}: weight MSE = {err:.4f}")

report("OLS", w_ols_h)
report("ridge", w_ridge)
```

```
OLS: weight MSE = 1.2347
ridge: weight MSE = 0.4400
```

OLS, with all its freedom and almost no data to discipline it, assigns large, confident, *wrong* weights to the 15 noise features. That is pure variance. Ridge shrinks everything and cuts the damage several-fold: a little bias, traded for a lot of variance. (Rerun this cell with $n = 100$ and watch OLS become more stable, as classical fixed-dimensional theory predicts.) On the road ahead, the same L_2 idea will return as weight decay in large neural networks.

Check yourself

Close the book for one minute and retrieve the complete learning story.

- What are the model, loss, and update rule in linear regression?
- Why do fitted predictions live in the column space of the design matrix?
- When do the closed form, scratch loop, and PyTorch module represent the same estimator?

1.9. Okay, so — what did we just build?

We taught a computer a function from examples, completely:

1. **A model family:** hyperplanes parameterized by knobs $\mathbf{w}, b$, with the bias absorbed into the weight vector when compact notation helps.
2. **A loss:** MSE, which is maximum likelihood under Gaussian noise and converts prediction error into the scalar signal that guides every update.
3. **An optimizer:** the exact projection when linearity permits, and gradient descent (feel the slope, step downhill) when it does not, which is always from here on.
4. **The generalization lens:** bias vs. variance, the data splits that referee them, and regularization as a first counter-measure to overfitting.

And one reframing that will carry the whole book: linear regression is a single neuron with the identity activation,

$$\underbrace{\hat{y} = \mathbf{w}^\top \mathbf{x} + b}_{\text{this chapter}} \xrightarrow{\text{add a nonlinearity}} \underbrace{\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + b)}_{\text{a neuron}}.$$

First, in [Chapter 2](#), that squashing function turns our regressor into a classifier. Then, in [Chapter 3](#), we stack neurons and discover why the nonlinearity is not optional.

💡 The first “make it learnable” step

We first saw the geometry of the linear response in [Figure 1.2](#). Now we can name its neural-network form. A neuron computes the same weighted response and bias, then applies σ . Linear regression is the identity-activation case. Losses, gradients, and updates carry over unchanged; the nonlinearity is the first ingredient that expands what the model can learn.

Sources and further reading

- [Hoerl and Kennard, “Ridge Regression: Biased Estimation for Nonorthogonal Problems” \(1970\)](#) introduces the biased estimator behind this chapter’s stability trade-off and regularization geometry.
- [Belkin et al., “Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-Off” \(2019\)](#) is the promised next picture: why the classical U-curve can become double descent in interpolating models.

Exercises

1. **(Pencil.)** Derive the normal equations Equation 1.2 by expanding $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$, taking the gradient with respect to $\mathbf{w}$, and setting it to zero. Notice that this derivation does not require an inverse. Then state what becomes unique when $\mathbf{X}$ has full column rank and what remains unique when it is rank deficient.
2. **(Pencil.)** Repeat the derivation with the ridge penalty $\lambda\|\mathbf{w}\|_2^2$ and show the solution becomes $(\mathbf{X}^\top\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$ when the data-fit term is the MSE. Why does the $n\lambda\mathbf{I}$ term guarantee invertibility for any $\lambda > 0$? If you augment $\mathbf{X}$ with a bias column, which diagonal entry should be zero so the intercept is not penalized?
3. **(Code.)** In `make_synthetic_data`, raise `noise` to 1.0 and rerun all three implementations 10 times with different seeds. How much do the learned weights vary across runs? Which term of Equation 1.5 are you watching?
4. **(Code.)** Set the learning rate in `LinearRegressionScratch` to 1.0 and rerun. Describe what the loss curve does, and explain it using the blindfolded-descent picture.
5. **(Code.)** In the ridge experiment, sweep $\lambda \in \{0.01, 0.1, 1, 10, 100\}$ and plot both weight MSE and $\|\mathbf{w}\|_2$ against λ . Where is the sweet spot, and what happens to error and weight scale at the two extremes? Connect your answer to bias and variance.

2. Linear Models that Classify: Logistic and Softmax

Regression predicts numbers; most of what the world wants predicted is *categories* — cat or dog, spam or not, which of ten digits. This chapter converts the linear machinery of [Chapter 1](#) into a classifier without discarding a single idea: the same weighted sums, the same maximum-likelihood recipe for the loss, the same gradient descent. What changes is one squashing function at the output, and it changes less than you might think.

2.1. What actually changes

Return to the computation circuit in [Figure 1.7](#). Keep every weighted input and the bias, but now call the circuit’s output a raw **score** o . Classification adds one operation after that score; the weighted-sum machinery itself stays intact.

In regression, $y \in \mathbb{R}$ and the Gaussian noise model handed us the MSE ([Chapter 1](#)). In classification, y is a label from $\{0, \dots, K - 1\}$. Could we just regress onto the labels with MSE anyway? We could, and it fails for a principled reason: **the loss is a noise model, and Gaussian is the wrong model for a coin flip**. A binary outcome deviates from its probability like a Bernoulli variable, not like additive Gaussian noise $\rightarrow$ maximum likelihood demands a different loss. Everything in this chapter falls out of taking that seriously.

What we need is two things:

1. a way to turn the model’s raw score (its **logit**, $o \in \mathbb{R}$) into a probability,
2. a loss that measures how wrong a *probability* is.

2.2. Binary classification

2.2.1. From score to probability: the sigmoid

Our linear model produces $o = \mathbf{w}^\top \mathbf{x} + b$, an unbounded score. The **sigmoid** squashes it into a probability:

$$\sigma(o) = \frac{1}{1 + e^{-o}}. \tag{2.1}$$

PLAN

1. Evaluate the sigmoid across a range of logits.
-

CODE

```
import torch
import matplotlib.pyplot as plt

# [1]
o = torch.linspace(-6, 6, 300)
sigmoid_response = torch.sigmoid(o)
```

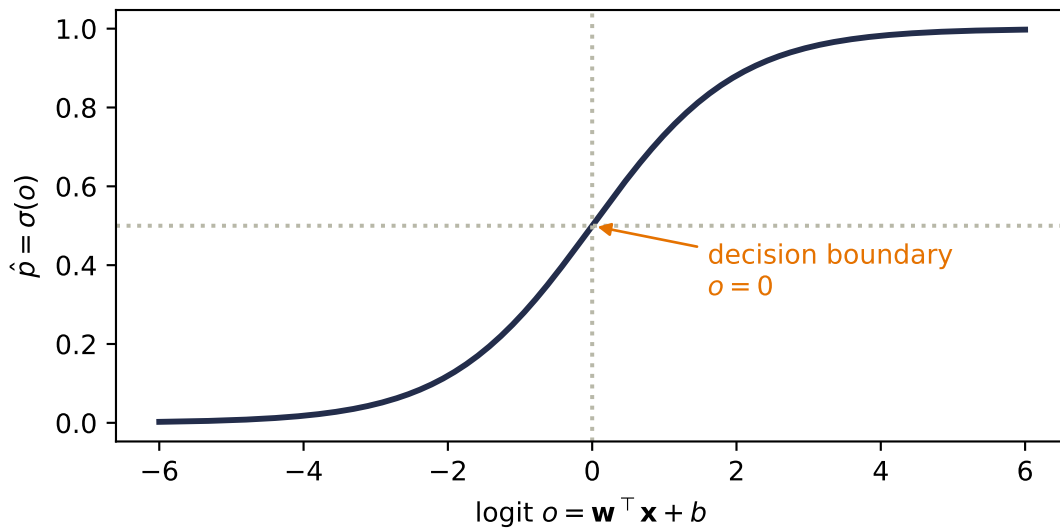


Figure 2.1.: The sigmoid maps any real score to $(0,1)$, crossing $\frac{1}{2}$ exactly at $o = 0$. The model’s uncertainty lives in the middle; its confident regions saturate at the tails.

Notice what did *not* change: the decision boundary $\hat{p} = \frac{1}{2}$ is exactly $o = 0$, i.e. $\mathbf{w}^\top \mathbf{x} + b = 0$ — the same hyperplane geometry from [Chapter 1](#), with $\mathbf{w}$ still the template the input is scored against. The sigmoid does not bend the boundary; it grades our confidence on either side of it.

2.2.2. The loss: cross-entropy from maximum likelihood

Run the maximum-likelihood recipe with the correct noise model. If the label is a coin flip with $P(y=1 \mid \mathbf{x}) = \hat{p}$, the likelihood of the dataset is $\prod_i \hat{p}_i^{y_i} (1 - \hat{p}_i)^{1-y_i}$; taking the negative log gives the **binary cross-entropy**:

$$\mathcal{L}_{\text{BCE}} = -[y \log \hat{p} + (1 - y) \log(1 - \hat{p})]. \quad (2.2)$$

Read it as *surprise*: if the true label is 1, the loss is $-\log \hat{p}$ — small when the model believed in the outcome, exploding as the model's belief goes to zero. A confident wrong answer is punished without mercy, which is exactly the pressure a probability deserves.

2.2.3. The beautiful gradient

Chain the sigmoid into the cross-entropy and differentiate with respect to the logit, and something remarkable drops out (Exercise 1 walks you through it):

$$\frac{\partial \mathcal{L}}{\partial o} = \hat{p} - y. \quad (2.3)$$

The gradient is *literally the prediction error*, the same form MSE gave us for regression. All the logs and exponentials annihilate each other. This is not luck, and it is not the last time you will see it.

2.3. Multiclass classification: softmax

With K classes, run K templates at once: $\mathbf{W} \in \mathbb{R}^{K \times d}$ produces a score vector $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, one logit per class. To turn K scores into a probability distribution, exponentiate (making everything positive) and normalize:

$$\hat{y}_j = \text{softmax}(\mathbf{o})_j = \frac{e^{o_j}}{\sum_{k=1}^K e^{o_k}}. \quad (2.4)$$

Softmax has three properties worth committing to memory: every output is positive, the outputs sum to one, and — less obviously — it is **invariant to constant shifts**: adding the same c to every logit changes nothing, because $e^{o_j+c} = e^c e^{o_j}$ and the e^c cancels top and bottom. Only the *differences* between scores matter.

PLAN

1. Compare softmax probabilities before and after a common logit shift.
-

CODE

```
# [1]
logits = torch.tensor([2.0, 0.5, -1.0, 1.0])
softmax_cases = [(shift, torch.softmax(logits + shift, dim=0))
                  for shift in (0.0, 100.0)]
```

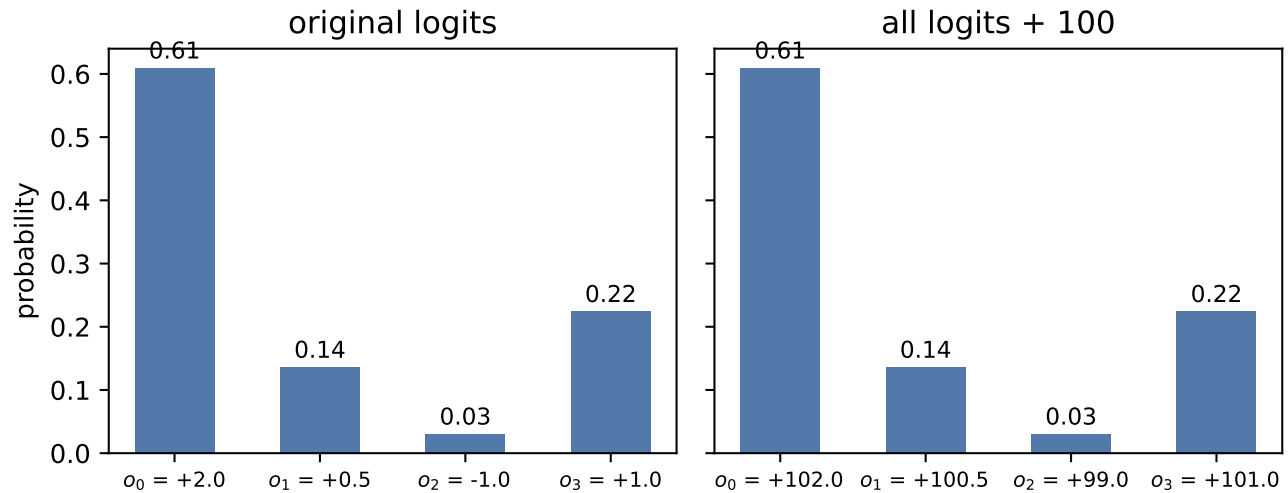


Figure 2.2.: Softmax turns scores into a distribution. Shifting every logit by the same constant (right panel) leaves the probabilities unchanged: only the differences between scores carry information.

💡 Remember this machine: scores $\rightarrow$ weights

Step back from classification for a moment. What softmax really is: a differentiable device that takes any list of scores and returns **positive weights that sum to one**, favoring the large scores without silencing the small ones. Today the scores are class logits. Much later, we will score how well one current request matches several stored candidates, then use the resulting weights to blend what those candidates contain. For now, keep only the reusable machine: scores $\rightarrow$ positive weights that sum to one. One normalizer, the whole book long.

i What “differentiable” buys us — softening the hard

That word *differentiable* deserves a moment in the light, because it names one of the great design patterns of this field.

Consider the obvious way to pick a class: take the **argmax**, then output a one-hot vector with 1 at the winning index and 0 elsewhere. (The scalar **max** operation is different and does have a gradient through its winning input.) Argmax-plus-one-hot is *hard*: nudge a losing logit slightly and the output does not move at all; nudge it past the winner and the output jumps all at once. Its derivative is zero almost everywhere, so the output gives gradient descent no local hint about which losing score should move or by how much. Chapter 5 will formalize how such local derivatives combine across many layers; for now, zero means there is no useful direction to follow.

Softmax is a smooth relaxation of that one-hot selection; the name is literal: a **soft max**. Instead of crowning one winner, it distributes belief according to the scores, and now every logit can affect the loss. We gave up a little decisiveness and bought trainability.

Watch for this move; it is everywhere once you see it. Whenever a useful operation is hard — a lookup, a yes/no choice, even *sorting a list* (yes, researchers have built differentiable sorting!) — the play is the same: replace it with a smooth version, train, and sharpen later if needed. The one to remember for this book is a hard memory lookup: one address wins and every other row is ignored. In Part IV we will replace that single address with a weighted read from several candidates.

PLAN

1. Compare one hard selection with three temperature-softened distributions.
-

CODE

```
# [1]
logits_demo = torch.tensor([2.0, 1.0, 0.2, -0.5])
klassen = ["A", "B", "C", "D"]
hard = torch.zeros(4); hard[logits_demo.argmax()] = 1.0
temperatures = [0.5, 1.0, 4.0]
soft_cases = [torch.softmax(logits_demo / temp, dim=0) for temp in temperatures]
```

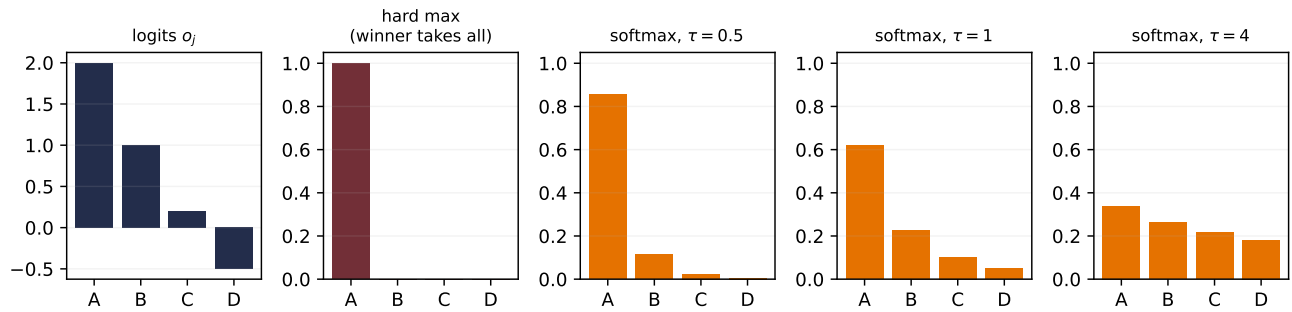


Figure 2.3.: The softening move, in one row. The same four logits (left) pass through the hard argmax-plus-one-hot selection — a cliff with zero gradient almost everywhere — and through softmax at three temperatures. Small τ sharpens toward the hard winner; large τ flattens toward indifference; every version in between is smooth, so a useful derivative reaches every logit. Chapter 13 returns to this same normalization pattern.

2.3.1. Cross-entropy, multiclass edition

With one-hot labels $\mathbf{y}$ (all zeros except a 1 at the true class), the cross-entropy is

$$\mathcal{L}_{\text{CE}} = - \sum_{j=1}^K y_j \log \hat{y}_j = - \log \hat{y}_{\text{true class}}, \quad (2.5)$$

the surprise at the true class, again. And the gradient repeats its trick:

$$\frac{\partial \mathcal{L}}{\partial o_j} = \hat{y}_j - y_j, \quad (2.6)$$

prediction minus target, one more time. This recurrence is no coincidence — sigmoid/BCE and softmax/CE are both members of the exponential family paired with their natural losses, and that pairing always collapses the gradient to the error.

i The information-theory reading

Entropy $H[P] = -\sum_j p_j \log p_j$ measures the uncertainty in a distribution; cross-entropy $H(P, Q) = -\sum_j p_j \log q_j$ is the cost of encoding outcomes from P using codes built for Q . Minimizing our loss is minimizing that encoding cost — equivalently, maximizing the likelihood assigned to the observed labels. Probability and coding give two readings of the same objective. A later chapter will give the remaining mismatch term its own name, when we have a reason to compare whole distributions rather than one-hot labels.

2.3.2. One numerical landmine

Computing e^{o_j} overflows once a logit reaches the high eighties in float32 (about 700 in float64) — scales training easily produces. The fix exploits shift invariance: subtract the max logit before exponentiating (the *log-sum-exp trick*), which changes nothing mathematically and everything numerically. We will use it in the code below; the deeper story of floating-point limits lives in Appendix C.

2.4. Build it: three classes, from scratch

Now we assemble the full classifier on three Gaussian blobs. It has four pieces, all of which you have already met: three templates (one per class), the stable softmax, the cross-entropy loss, and the gradient. And since Equation 2.6 reduced that gradient to $\hat{y} - y$, we can implement it by hand → no autograd needed:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable `stable_softmax` helper.
3. Apply the softmax cross-entropy gradient to all three templates.
4. Report or visualize the measured result.

CODE

```
# [1]
torch.manual_seed(6050)
centers = torch.tensor([[-2.0, 0.0], [2.0, 0.0], [0.0, 2.5]])
X = torch.cat([c + 0.7 * torch.randn(150, 2) for c in centers]) # (450, 2)
y = torch.arange(3).repeat_interleave(150) # (450,)
Y = torch.nn.functional.one_hot(y, 3).float() # (450, 3)

# [2]
def stable_softmax(logits: torch.Tensor) -> torch.Tensor:
    z = logits - logits.max(dim=-1, keepdim=True).values # shift invariance at work
    e = torch.exp(z)
    return e / e.sum(dim=-1, keepdim=True)
```

2. Linear Models that Classify: Logistic and Softmax

```
W, b = torch.zeros(3, 2), torch.zeros(3)
# [3]
for step in range(400):
    P = stable_softmax(X @ W.T + b)          # (450, 3) predicted probabilities
    G = (P - Y) / len(X)                    # the beautiful gradient, eq. (6)
    W -= 1.0 * G.T @ X                      # blame out x signal in
    b -= 1.0 * G.sum(0)

P = stable_softmax(X @ W.T + b)             # evaluate the final updated parameters
ce = -(Y * torch.log(P)).sum(1).mean()
acc = (P.argmax(1) == y).float().mean()
# [4]
print(f"cross-entropy {ce:.3f}    accuracy {acc:.1%}")
```

cross-entropy 0.033 accuracy 98.9%

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Decision regions with confidence shading.
-

CODE

```
import numpy as np

# [1]
gx, gy = np.meshgrid(np.linspace(-4.5, 4.5, 300), np.linspace(-2.5, 5, 300))
grid = torch.tensor(np.stack([gx.ravel(), gy.ravel()], 1), dtype=torch.float32)
Pg = stable_softmax(grid @ W.T + b)
cls = Pg.argmax(1).reshape(gx.shape).numpy()
conf = Pg.max(1).values.reshape(gx.shape).numpy()

plt.figure(figsize=(6, 4.4))
plt.contourf(gx, gy, cls, levels=[-0.5, 0.5, 1.5, 2.5],
             colors=["#DCE6F2", "#FDE3C8", "#E3EEE3"])
plt.contourf(gx, gy, 1 - conf, levels=12, cmap="Greys",
             alpha=0.25, antialiased=True)

# [2]
for k, color in enumerate(["#232D4B", "#E57200", "#6B8E6B"]):
    plt.scatter(X[y == k, 0], X[y == k, 1], s=8, c=color, label=f"class {k}")
plt.xlabel("$x_1$"); plt.ylabel("$x_2$"); plt.legend(loc="lower right")
plt.tight_layout(); plt.show()
```

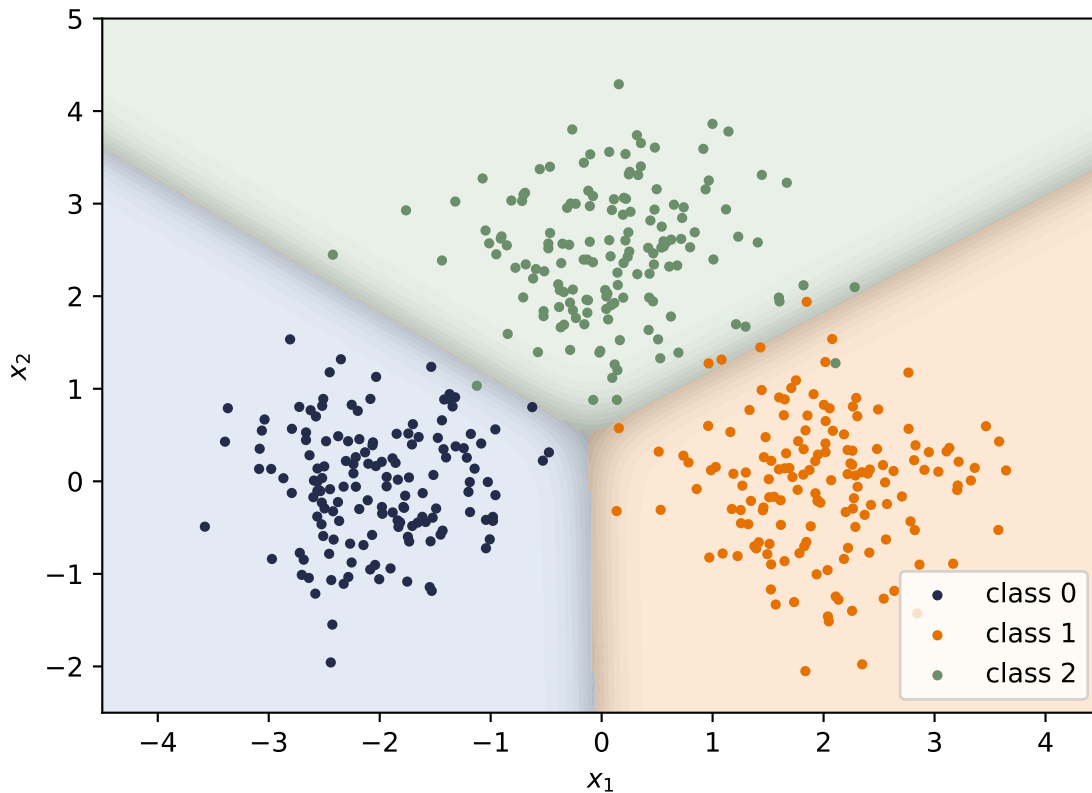


Figure 2.4.: Three templates, three linear boundaries. Color shows the predicted class; shading shows confidence (the max softmax probability) — crisp deep in each region, washed out along the boundaries where classes compete.

The framework version is two lines, with one trap worth a warning. This loss object is the only new framework tool introduced here: it evaluates logits and labels, while the manual loop above still supplies the gradient. Optimizers and automatic backpropagation remain the previews marked in [Chapter 1](#) until Chapters 4 and 5.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Compare the hand-computed cross-entropy with PyTorch's fused loss.
-

CODE

```
# [1]
loss_fn = torch.nn.CrossEntropyLoss()
# [2]
print(f"ours: {ce:.6f}    torch: {loss_fn(X @ W.T + b, y):.6f}")
```

```
ours: 0.033425    torch: 0.033425
```

⚠ Let PyTorch do the logits-to-probabilities conversion

`nn.CrossEntropyLoss` expects **raw logits**, and that is a feature, not a quirk: it fuses the softmax and the logarithm internally using the log-sum-exp trick, which is more numerically stable than any softmax-then-log you would write yourself. So the advice is: have your model output logits, hand them straight to the loss, and only apply softmax when you actually need probabilities to report. (The same goes for binary problems: prefer `binary_cross_entropy_with_logits` over sigmoid-then-BCE.) The classic bug is feeding softmax outputs into `CrossEntropyLoss` — the model still trains, just badly and silently. If a classifier is mysteriously mediocre, check this first.

i Check yourself

Close the book for one minute and reconstruct the classifier.

- What remains linear before sigmoid or softmax is applied?
- Why does the target distribution determine the negative-log-likelihood loss?
- Why should logits, rather than already normalized probabilities, enter PyTorch’s cross-entropy loss?

2.5. Okay, so — classification is regression plus a normalizer

1. **The geometry survived:** logits are still template similarities $\mathbf{w}^\top \mathbf{x} + b$, boundaries are still hyperplanes. Sigmoid and softmax grade confidence; they do not bend anything.
2. **The loss came from the noise model**, exactly as in [Chapter 1](#): Bernoulli $\rightarrow$ BCE, categorical $\rightarrow$ cross-entropy, both read as *surprise at the truth*.
3. **The gradient is the error**, $\hat{y} - y$, both times — the exponential-family pairing at work.
4. **Softmax is a scores-to-weights machine**, shift-invariant, numerically tamed by log-sum-exp — and it will return, unchanged, as the normalizer of attention.

Next, we let the templates themselves be built out of other templates: [Chapter 3](#) stacks these linear units — and discovers why a bend between them is not optional.

Sources and further reading

- Cox, “The Regression Analysis of Binary Sequences” (1958) develops the logistic model as a principled tool for binary outcomes.
- Bridle, “Probabilistic Interpretation of Feedforward Classification Network Outputs” (1990) connects normalized exponential outputs to probabilistic multiclass classification.
- Shannon, “A Mathematical Theory of Communication” (1948) is the original source behind the entropy and information vocabulary used to read cross-entropy.

Exercises

1. **(Pencil.)** Derive Equation 2.3: write $\mathcal{L}_{\text{BCE}}$ as a function of o through $\hat{p} = \sigma(o)$, use $\sigma'(o) = \sigma(o)(1 - \sigma(o))$, and simplify. Which factors cancel, and why is the result independent of the

- sigmoid's saturation?
2. **(Pencil.)** Prove softmax's shift invariance from Equation 2.4, then explain why subtracting the max logit makes the computation overflow-proof: what is the largest value ever exponentiated?
 3. **(Code.)** Temperature: replace `logits` with `logits / T` in the softmax figure for $T \in \{0.5, 1, 2, 10\}$. Describe what happens to the distribution at both extremes. (Keep this dial in mind — it returns when we scale attention scores in Part IV.)
 4. **(Code.)** Train the blobs classifier with MSE on one-hot targets instead of cross-entropy (same manual loop; the gradient changes). Compare accuracy and the confidence shading near boundaries. What does the wrong noise model cost?
 5. **(Code.)** Move the three blob centers closer together ($0.7 \rightarrow 1.2$ noise, or centers at distance 1) and retrain. Where does the confidence shading collapse, and what does the cross-entropy value tell you compared to the accuracy?

3. Nonlinearity and the MLP

Our linear models can regress (Chapter 1) and classify (Chapter 2), and it is tempting to think we could get more power by simply stacking them: feed the output of one linear layer into another. Watch what happens:

$$\mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W}_{\text{new}}\mathbf{x} + \mathbf{b}_{\text{new}}. \quad (3.1)$$

The stack *collapses*. Two linear layers, or twenty, are exactly one linear layer with different knobs $\rightarrow$ no amount of stacking buys any new expressive power. To learn the patterns of the real world we need a magic ingredient: **nonlinearity**. This chapter is about what one small bend does to everything.

3.1. The tiny dataset that embarrassed a field

You do not need real-world data to expose the linear ceiling. Four points suffice. The XOR function outputs 1 exactly when its two binary inputs *differ*:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Plot the four points and try to separate the 1s from the 0s with a single line. The positive examples sit on one diagonal, the negatives on the other; any line you draw puts at least one point on the wrong side. A linear classifier cannot represent this function, no matter how it is trained.

i A historical scar

This is not a toy observation. The limitations of single-layer perceptrons, including XOR, were publicized in 1969 and became emblematic of the case against that research program. They contributed to reduced enthusiasm and funding, but they were not a single-cause explanation for the first “AI winter”; limited compute, data, and unmet promises mattered too. Multi-layer networks and practical backpropagation (Chapter 5) later removed the representational obstacle exposed here.

Let us confirm the ceiling empirically. We reuse the framework loop previewed in Chapter 1 as a measuring instrument; Chapter 4 and Chapter 5 will explain the optimizer and `backward()` rather than asking you to treat them as magic.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable `train` helper.
 3. Run xor training.
 4. Report or visualize the measured result.
-

CODE

```
import torch
from torch import nn

# [1]
torch.manual_seed(6050)
X_xor = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
y_xor = torch.tensor([0., 1., 1., 0.])

# [2]
def train(
    model: nn.Module, X: torch.Tensor, y: torch.Tensor,
    steps: int = 4000, lr: float = 0.5
) -> nn.Module:
    opt = torch.optim.SGD(model.parameters(), lr=lr)
    for _ in range(steps):
        opt.zero_grad()
        loss = nn.functional.binary_cross_entropy_with_logits(model(X).squeeze(-1),
↪      y)
        loss.backward()
        opt.step()
    return model

# [3]
linear = train(nn.Linear(2, 1), X_xor, y_xor)
mlp = train(nn.Sequential(nn.Linear(2, 4), nn.ReLU(), nn.Linear(4, 1)),
            X_xor, y_xor)

with torch.no_grad():
    linear_p = torch.sigmoid(linear(X_xor).squeeze(-1))
    linear_bce = nn.functional.binary_cross_entropy(linear_p, y_xor)
    mlp_acc = ((mlp(X_xor).squeeze(-1) > 0).float() == y_xor).float().mean()

# [4]
print(f"trained linear: probabilities {linear_p.numpy().round(3)}, "
      f"BCE {linear_bce:.3f}")
print(f" MLP (4 hidden): accuracy {mlp_acc:.0%}")
```

```
trained linear: probabilities [0.5 0.5 0.5 0.5], BCE 0.693
MLP (4 hidden): accuracy 100%
```

3. Nonlinearity and the MLP

The cross-entropy optimum for this symmetric XOR dataset assigns probability 0.5 to all four points, with loss $\log 2 \approx 0.693$. Turning those exact ties into hard labels with a floating-point threshold can report 25%, 50%, or 75% because of tiny rounding differences; that number is not evidence about the classifier. The honest geometric statement is that a hard linear boundary can classify at most three of the four points (75%), while the little network with a bend is perfect.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Decision regions.
-

CODE

```
import numpy as np
import matplotlib.pyplot as plt

# [1]
gx, gy = np.meshgrid(np.linspace(-0.6, 1.6, 220), np.linspace(-0.6, 1.6, 220))
grid = torch.tensor(np.stack([gx.ravel(), gy.ravel()], 1), dtype=torch.float32)

fig, axes = plt.subplots(1, 2, figsize=(7.4, 3.4), sharey=True)
# [2]
for ax, model, title in [(axes[0], None, "best linear rule (75%)"),
                          (axes[1], mlp, "trained MLP (100%)")]:
    with torch.no_grad():
        if model is None:
            Z = (grid.sum(1) - 0.5 > 0).float().reshape(gx.shape)
        else:
            Z = (model(grid).squeeze(-1) > 0).float().reshape(gx.shape)
    ax.contourf(gx, gy, Z, levels=[-0.5, 0.5, 1.5],
                colors=["#DCE6F2", "#FDE3C8"], alpha=0.9)
    for cls, marker, color in [(0, "o", "#232D4B"), (1, "s", "#E57200")]:
        pts = X_xor[y_xor == cls]
        ax.scatter(pts[:, 0], pts[:, 1], marker=marker, s=120, c=color,
                    edgecolors="white", linewidths=1.5, zorder=3,
                    label=f"class {cls}")
    ax.set_title(title); ax.set_xlabel("$x_1$")
axes[0].set_ylabel("$x_2$"); axes[0].legend(loc="center right")
plt.tight_layout(); plt.show()
```

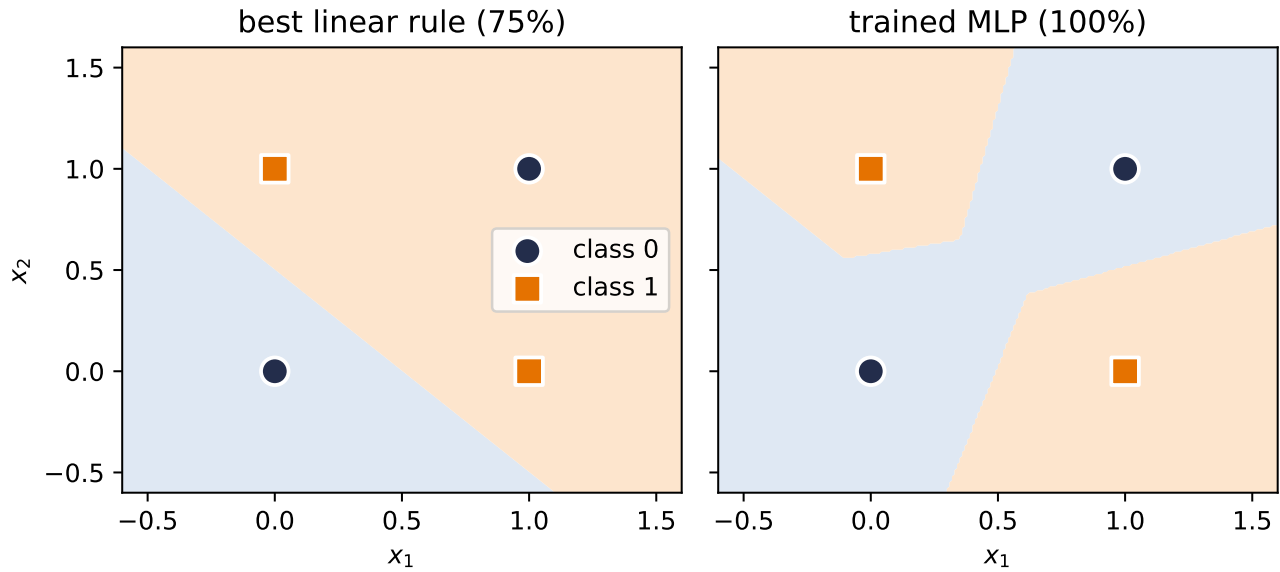


Figure 3.1.: Left: a representative best linear boundary gets three of four XOR points right, but one point must remain on the wrong side. Right: the trained 4-hidden-unit MLP carves separate regions for the two positive corners.

3.2. The neuron: a threshold with a bend

Return once more to Figure 1.7. A neuron keeps that weighted-sum circuit and places a bend after its output.

The artificial neuron takes its inspiration from biology: a neuron receives signals, and it *fires* only when the accumulated stimulus crosses a threshold. The modern, computationally friendly version of that thresholding is the **rectified linear unit**:

$$\text{ReLU}(z) = \max(0, z), \quad f(\mathbf{x}) = \text{ReLU}(\mathbf{w}^\top \mathbf{x} + b). \quad (3.2)$$

The weighted input $\mathbf{w}^\top \mathbf{x} + b$ is the stimulus — and recall from Chapter 1 that it is a *similarity score* against the template $\mathbf{w}$. Below threshold, the neuron is silent (output 0); above it, the neuron fires with intensity proportional to the stimulus.

PLAN

1. Evaluate one neuron's stimulus and rectified response.
 2. Evaluate the same neuron over the input plane.
-

CODE

```
# [1]
z = torch.linspace(-3, 3, 300)
response = torch.relu(z)
# [2]
axis = torch.linspace(-2.4, 2.4, 220)
x1, x2 = torch.meshgrid(axis, axis, indexing="xy")
w, b = torch.tensor([1.0, 0.65]), -0.25
surface = torch.relu(w[0] * x1 + w[1] * x2 + b)
```

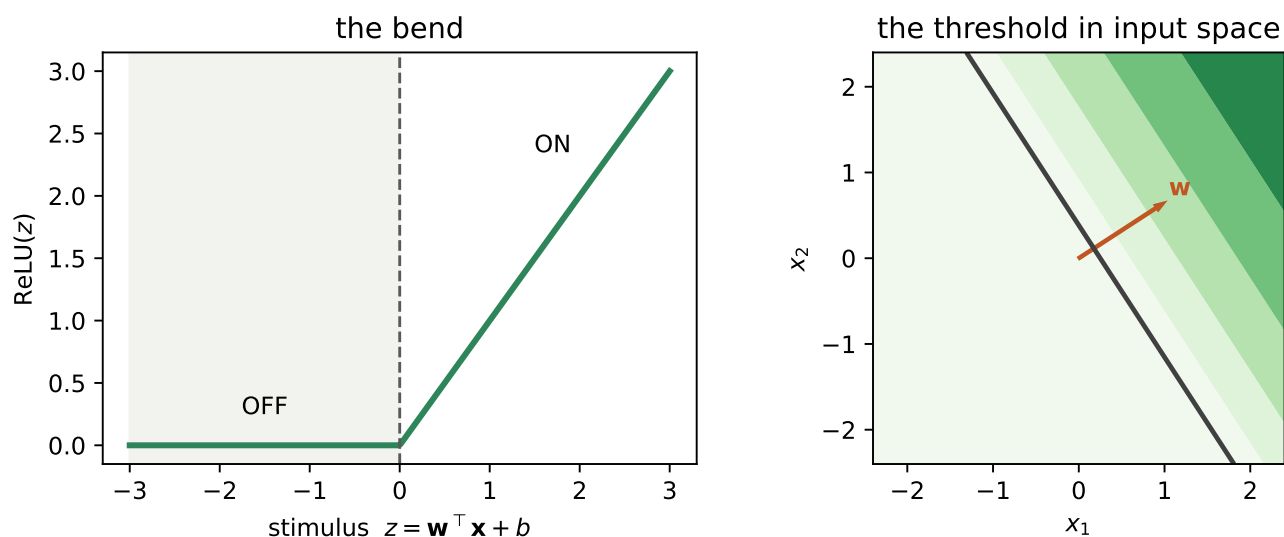


Figure 3.2.: One neuron, two views. Left: ReLU bends a scalar stimulus at zero—silent on one side, proportional response on the other. Right: the same bend creates a threshold line in the input plane; the weight vector is perpendicular to it, and activation grows as inputs move across the line.

In one dimension a single neuron turns a line into a **hinge**, and in two dimensions it draws a line through the plane: an OFF region and an ON region, with $\mathbf{w}$ perpendicular to the boundary. That boundary is the fundamental unit of computation in everything that follows.

(What about the sharp corner at $z = 0$, where the derivative is undefined? Libraries choose a subgradient convention, commonly 0. Exact zeros can occur systematically with zero biases, sparse activations, or symmetric inputs, so the convention is not merely academic. The gradient story of this open-or-shut gate — including when it helps and when a unit goes dead — is told in [Chapter 5](#).)

3.3. Layers of hinges: bumps and polytopes

One neuron makes one boundary. A layer of k neurons makes k boundaries, and their intersections partition the input space into convex **polyhedral regions** (bounded ones are polytopes), and within each cell the layer computes a plain linear function. For fixed input dimension d , the maximum region count of one generic hyperplane arrangement grows on the order of k^d .

3. *Nonlinearity and the MLP*

The simplest compact constructive example lives in 1D. Combine three hinges so the slopes cancel again after the third breakpoint:

 PLAN

1. Combine three shifted hinges into a compact bump.
-

CODE

```
# [1]
xs = torch.linspace(-3, 5, 400)
h1 = torch.relu(xs + 1.5)
h2 = torch.relu(xs - 0.5)
h3 = torch.relu(xs - 2.5)
bump = h1 - 2 * h2 + h3
```

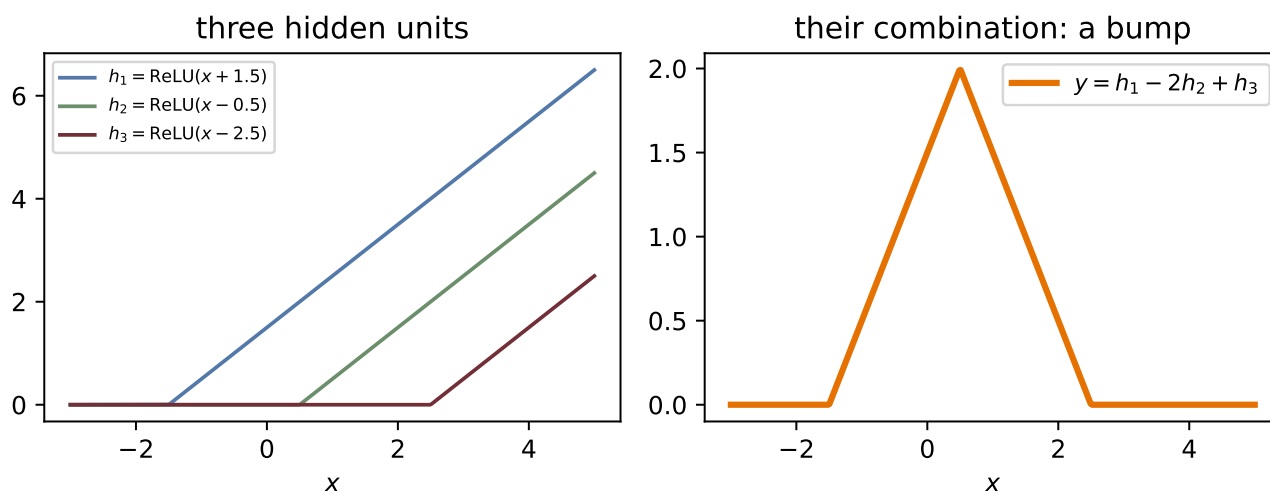


Figure 3.3.: Three hinges make a compact triangular bump: the first starts the rise, the second reverses the slope, and the third cancels it back to zero. Bumps are the brush strokes of function approximation.

Follow the pieces: before the first hinge, all units are silent and the output is zero. After the first breakpoint, h_1 starts the climb. The coefficient -2 on h_2 reverses the slope at the peak, and h_3 cancels that downward slope at the final breakpoint. Four linear pieces $\rightarrow$ one compact bump.

3.4. The universal approximation theorem

Bumps are the whole secret. If you can make one bump, you can make many, at any position and height $\rightarrow$ with enough hidden units you can *tile* any continuous function, the way an artist paints a shape with brush strokes. That is the intuition behind the **universal approximation theorem**: a network with a single hidden layer can, in principle, approximate any continuous function on a bounded region to any desired accuracy.

Watch the theorem materialize as width grows:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable `fit` helper.
 3. Implement the width sweep.
-

CODE

```
# [1]
xs1 = torch.linspace(-3, 3, 200)[: , None]
target = torch.sin(1.5 * xs1) + 0.08 * xs1 ** 2

# [2]
def fit(width: int, steps: int = 3000, lr: float = 0.05) -> torch.Tensor:
    torch.manual_seed(6050)
    net = nn.Sequential(nn.Linear(1, width), nn.ReLU(), nn.Linear(width, 1))
    opt = torch.optim.SGD(net.parameters(), lr=lr)
    for _ in range(steps):
        opt.zero_grad()
        ((net(xs1) - target) ** 2).mean().backward()
        opt.step()
    with torch.no_grad():
        return net(xs1).squeeze(-1)

plt.figure(figsize=(6.6, 3.6))
plt.plot(xs1, target, color="#232D4B", lw=2.5, label="target")

# [3]
for width, color in [(2, "#B8B8A8"), (8, "#5379AA"), (64, "#E57200")]:
    plt.plot(xs1, fit(width), color=color, lw=1.6, label=f"width {width}")
plt.xlabel("$x$"); plt.legend(); plt.tight_layout(); plt.show()
```

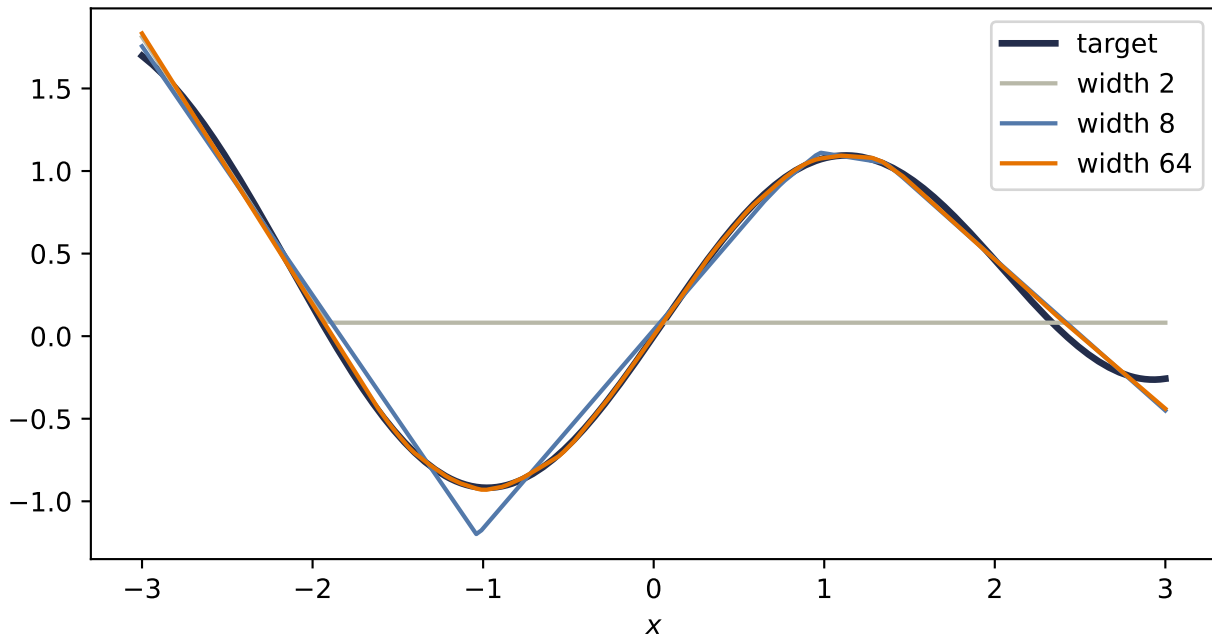



Figure 3.4.: One hidden ReLU layer approximating a continuous curved target. Width 2 is too rigid; width 8 catches the shape; width 64 follows it closely. Width buys more hinge locations, but the theorem itself does not tell the optimizer where to place them.

Now read the theorem’s fine print, because it is where beginners over-promise:

⚠ Trap: universal approximation is not a training guarantee

1. It does not tell you **how to find the weights** — existence of a good network is not a training algorithm. Finding the weights is the job of the loss and the optimizer, and nothing guarantees gradient descent lands on the theorem’s solution.
2. It does not say the network is **small**. Tiling a complicated function with one-layer bumps can require astronomically many units.
3. In its usual uniform-approximation form, it speaks of **continuous functions on bounded regions** and says nothing about how the fit behaves *between* or *beyond* your samples. A continuous ReLU network cannot uniformly approximate a jump to arbitrary accuracy: near the jump its worst-case error stays at least half the jump size. Under a mean-square (L_2) criterion, however, it can squeeze the transition into a narrow interval and make the *average* error arbitrarily small. The norm and the target class matter. Approximation is still not generalization; that distinction gets its own chapter ([Chapter 6](#)).

3.5. Why depth: boundaries that bend

If one hidden layer suffices in principle, why is the field called *deep* learning? Efficiency, and geometry. A second hidden layer receives, as its inputs, the piecewise-linear outputs of the first → its boundaries are no longer straight lines. They *bend* wherever the first layer’s boundaries cut the space. Boundaries composed of boundaries: each layer folds the space the previous layer drew.

Concretely: a ReLU network is exactly linear inside each region of input space where the on/off pattern of its units is constant, and those regions are polytopes whose walls are the units' boundaries. Deeper layers add walls that kink wherever they cross earlier ones. Exercise 4 has you draw this tiling and count its cells.

This bending can compound dramatically. For suitable ReLU constructions at fixed input dimension, achievable region counts grow exponentially with depth, while a single hidden layer's count grows polynomially with width. This is an existence and architecture-dependent comparison, not a promise that every trained deep network realizes all those regions. It gives the economic argument for depth: some partitions represented by a deep, narrow network require far more units in a shallow one.

3.6. Your first real MLP

Time to build the model properly as a reusable class.

💡 Coding hygiene: the house and its foundation

Every model you write from now on subclasses `nn.Module`, and the first line of your `__init__` is always `super().__init__()`. Think of it as pouring the foundation before building the house: it is what wires up parameter tracking, `.parameters()`, and everything the optimizer and autograd need. Forget it and PyTorch fails loudly — the one favor a framework can do you.

PLAN

1. Register the hidden and output layers, then compose them in `forward`.
-

CODE

```
# [1]
class MLP(nn.Module):
    """A multilayer perceptron: linear layers with bends between them."""

    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int):
        super().__init__() # the foundation
        self.hidden = nn.Linear(input_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, output_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.out(torch.relu(self.hidden(x)))
```

And a problem worthy of it: two interleaved moons, the classic shape no line can separate well. Same training loop as always — the model is the only thing that changed:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Fit a linear model and an MLP to the same moons.
3. Evaluate both decision rules on a dense grid.

CODE

```
# [1]
torch.manual_seed(6050)
n = 200
t = torch.rand(n) * torch.pi
moon1 = torch.stack([torch.cos(t), torch.sin(t)], 1) + 0.12 * torch.randn(n, 2)
moon2 = torch.stack([1 - torch.cos(t), 0.4 - torch.sin(t)], 1)
moon2 = moon2 + 0.12 * torch.randn(n, 2)
X_m = torch.cat([moon1, moon2])
y_m = torch.cat([torch.zeros(n), torch.ones(n)])

torch.manual_seed(6050)
# [2]
lin_m = train(nn.Linear(2, 1), X_m, y_m, steps=3000, lr=0.8)
torch.manual_seed(6050)
mlp_m = train(MLP(2, 16, 1), X_m, y_m, steps=3000, lr=0.8)

gx3, gy3 = np.meshgrid(np.linspace(-1.6, 2.6, 240), np.linspace(-1.1, 1.6, 240))
grid3 = torch.tensor(np.stack([gx3.ravel(), gy3.ravel()], 1), dtype=torch.float32)

# [3]
moon_panels = []
for model in (lin_m, mlp_m):
    with torch.no_grad():
        accuracy_value = (
            (model(X_m).squeeze(-1) > 0).float() == y_m
        ).float().mean()
        boundary_values = (
            model(grid3).squeeze(-1) > 0
        ).float().reshape(gx3.shape)
    moon_panels.append((accuracy_value, boundary_values))
```

Where did the curve go? Nowhere — and that is the chapter’s deepest point. The MLP did not learn a curved boundary; it learned a **space in which the boundary is straight**. Its readout is still a linear model, $\mathbf{w}^\top \phi_\theta(x) + b$: project the hidden activations onto the readout direction $\hat{\mathbf{w}} = \mathbf{w}/\|\mathbf{w}\|$ and the decision boundary sits at the exact coordinate $-b/\|\mathbf{w}\|$, a straight line. This is the preface’s central equation, drawn:

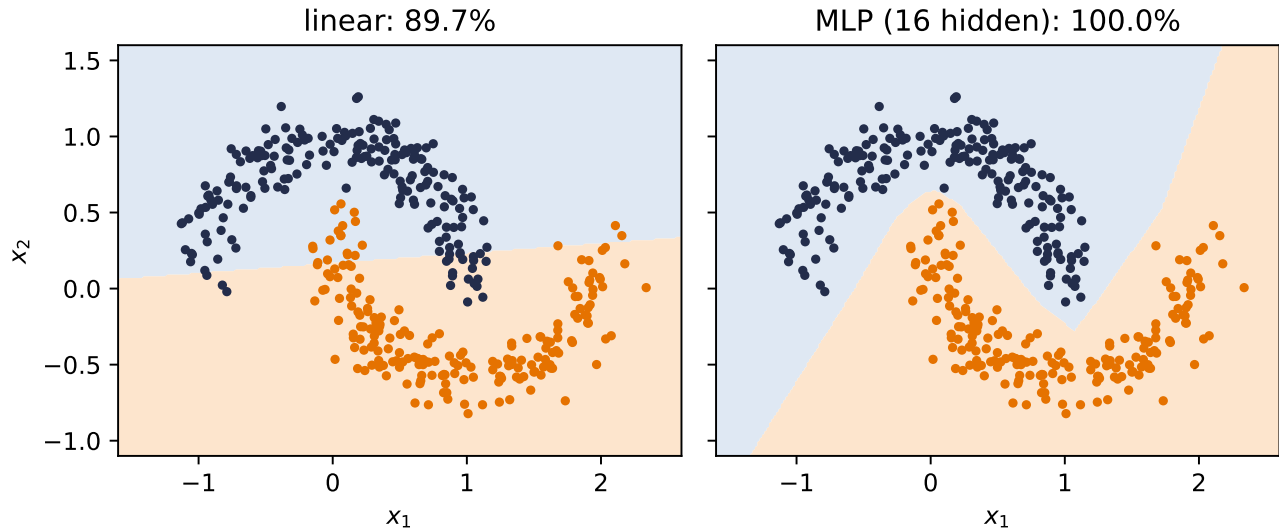


Figure 3.5.: Two moons: the linear model does what it can with one line (about 90% here); the MLP bends its boundary through the gap and separates the classes completely.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Project the moons into learned readout and orthogonal coordinates.
-

CODE

```
# [1]
with torch.no_grad():
    phi = torch.relu(mlp_m.hidden(X_m))
    w = mlp_m.out.weight.squeeze(0)
    b = mlp_m.out.bias.item()
    w_hat = w / w.norm()
    a1 = phi @ w_hat
    # phi_theta(x): (400, 16)
    # readout coordinate
# [2]
resid = phi - a1[:, None] * w_hat
resid = resid - resid.mean(0)
a2 = resid @ torch.linalg.svd(resid, full_matrices=False).Vh[0]
boundary = -b / w.norm()
# the line, exactly
field = (torch.relu(mlp_m.hidden(grid3))) @ w_hat).reshape(grid3.shape)
```

One honest footnote on the XOR network: in principle *two* hidden units suffice, but such minimal networks are temperamental; across random seeds they frequently get stuck at 50% accuracy with silent units (you will verify this in Exercise 3). A few spare units cost little and improve reliability. [Chapter 4](#) will examine when optimization noise helps exploration, without treating it as a guaranteed escape.

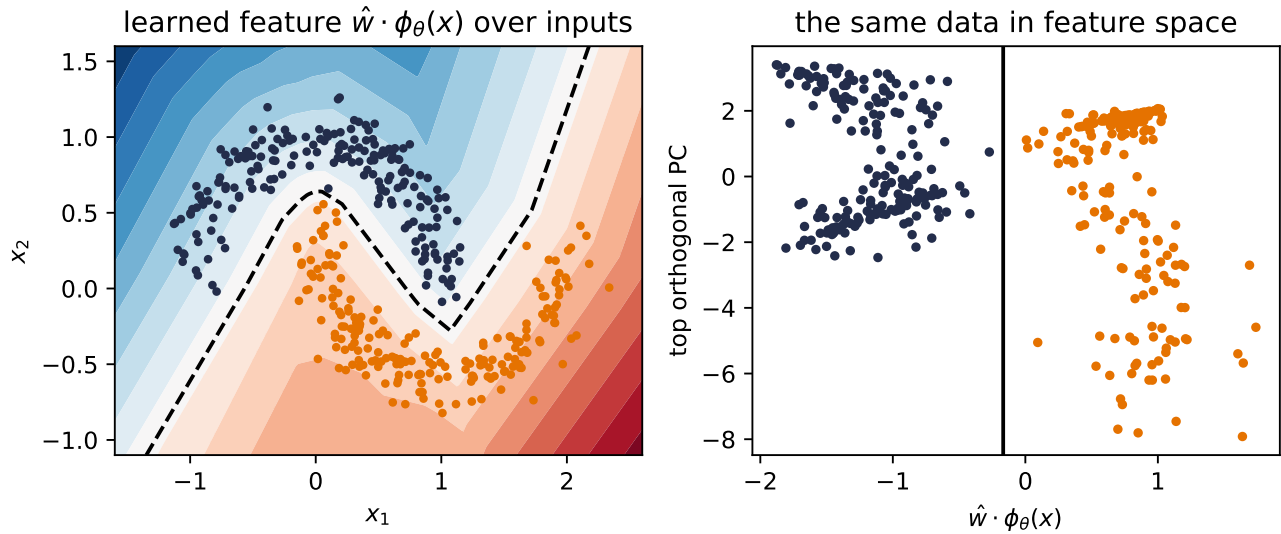


Figure 3.6.: The learned feature space, on the trained moons MLP from the previous figure. Left: the scalar feature $\hat{w} \cdot \phi_\theta(x)$ over the input plane; the dashed contour is the decision boundary, kinked wherever hidden units switch on or off. Right: the same points with coordinates (readout direction, top orthogonal principal component of ϕ_θ): the classes separate, and the boundary is the exact vertical line $\hat{w} \cdot \phi = -b/\|w\|$. Learning nonlinear features until a linear readout suffices is the book’s central move.

3.7. A cautionary tale, and a look ahead

There is a seductive story about what the layers of an MLP learn: the first layer detects edges, the next combines them into shapes, the next into objects — a tidy hierarchy of increasingly abstract features. It is the *idealized* story, and for fully-connected networks on raw images it is mostly **not what happens**. A fully-connected layer sees its input as one long, structureless vector; it has no notion that pixel potentially relates to its neighbor. Nothing in the architecture encourages the tidy hierarchy, and with every pixel connected to every unit, the parameter count explodes long before the hierarchy emerges.

We now hold real expressive power — networks that can, in principle, represent almost anything. [Chapter 4](#) trains them at scale and [Chapter 5](#) supplies their gradients. And then [Chapter 6](#) asks this chapter’s uncomfortable question: what happens when all that flexibility meets data it does not understand the *structure* of? The answer — watch an MLP fail, in pictures, then fix it by building knowledge into the architecture — is the turn the whole book pivots on.

i Check yourself

Close the book for one minute and rebuild the bend.

- Why does stacking linear layers without an activation still produce one linear map?
- What geometric object does one ReLU neuron introduce?
- How can a nonlinear feature map make a curved input-space problem linearly separable?

3.8. Okay, so — one bend changed everything

1. **Linear stacks collapse** (Equation 3.1); XOR proves four points can defeat any line.
2. **A neuron is a thresholded similarity score**: below the boundary silent, above it proportional. One neuron $\rightarrow$ one boundary; a layer $\rightarrow$ a polyhedral partition; three hinges $\rightarrow$ a compact bump.
3. **Bumps tile functions** (universal approximation) — but the theorem promises existence, not learnability, economy, or generalization.
4. **Depth bends boundaries**: suitable deep constructions can create region counts that grow exponentially with depth, yielding large efficiency gaps relative to comparable shallow constructions.
5. **The MLP is a class now** (`nn.Module`, foundation first), and it separates what no line can.

Sources and further reading

- Cybenko, “Approximation by Superpositions of a Sigmoidal Function” (1989) gives a foundational uniform universal-approximation result on compact domains.
- Hornik, “Approximation Capabilities of Multilayer Feedforward Networks” (1991) makes the distinction between uniform and L_p approximation explicit, which is exactly the jump-function nuance in this chapter.
- Nair and Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines” (2010) is an early empirical account of rectifiers preserving useful signal through multiple layers.
- Montúfar et al., “On the Number of Linear Regions of Deep Neural Networks” (2014) develops the architecture-dependent region-count argument behind the chapter’s case for depth.
- Lin, Tegmark, and Rolnick, “Why Does Deep and Cheap Learning Work So Well?” (2017) studies multiplication as a reusable compositional primitive and proves settings in which flattening that computation requires exponentially more units.

Exercises

1. **(Pencil.)** Generalize Equation 3.1: show by induction that any stack of L linear layers equals a single linear layer. Where exactly does inserting ReLU between layers break your proof?
2. **(Pencil.)** Using the bump construction, give explicit weights for a one-hidden-layer network that outputs approximately 1 on $[0, 1]$ and 0 outside $[-0.2, 1.2]$. How many hidden units did you need?
3. **(Code.)** Retrain the XOR network with `hidden_dim = 2` across ten different seeds. How often does it reach 100%? Inspect a failed run: what happened to its hidden units? (Hint: check how many are permanently silent.)
4. **(Pencil.)** For h lines in general position in two dimensions, derive the maximum region count $1 + h + \binom{h}{2} = \sum_{i=0}^2 \binom{h}{i}$. **(Code.)** Color the input plane by ReLU on/off pattern for a random $2 \rightarrow 8 \rightarrow 8$ network. Count connected constant-pattern cells on successively finer grids, then compare deep and wide networks at matched parameter count using seed 6050. Report the grid resolution with the count: a sampled count is a lower bound, not an exact theorem.
5. **(Code.)** Replace the moons MLP’s ReLU with `nn.Identity()` and retrain. Explain the resulting boundary in one sentence using Equation 3.1.

6. **(Pencil.)** Let σ be smooth and choose an offset c with $\sigma''(c) \neq 0$. Use Taylor expansion to show that

$$m_\lambda(u, v) = \frac{\sigma(c + \lambda(u + v)) + \sigma(c - \lambda(u + v)) - \sigma(c + \lambda(u - v)) - \sigma(c - \lambda(u - v))}{4\sigma''(c)\lambda^2} \rightarrow uv$$

as $\lambda \rightarrow 0$. **(Code.)** Use the logistic sigmoid with $c = 1$, sweep λ , and plot $|m_\lambda(u, v) - uv|$ in float64. Then compose the gate into a balanced tree for $x_1 \cdots x_8$. Explain both the approximation-error regime and the eventual roundoff regime. *Named wrong answer:* “expand the logistic sigmoid at $c = 0$ ” — that is its inflection point, where $\sigma''(0) = 0$ and the proposed denominator vanishes.

4. Training: Loss and Stochastic Gradient Descent

We now have models worth training: linear regressors ([Chapter 1](#)), classifiers ([Chapter 2](#)), and multilayer perceptrons ([Chapter 3](#)). We have losses that tell each of them how bad they are, and we know the direction of improvement is the negative gradient. What remains is the question every practitioner faces daily: *how, exactly, do you run the descent* when the dataset has a million examples, the model has millions of knobs, and the loss landscape is no longer a friendly bowl?

This chapter is about the workhorse answer, stochastic gradient descent, and the small set of refinements (learning rates, batch sizes, momentum, Adam) that turn it from a theoretical idea into the algorithm that trains essentially everything in this book.

4.1. Where losses come from, one more time

Before optimizing a loss, remember why it is *that* loss. In [Chapter 1](#) we saw that assuming Gaussian noise on a linear process makes maximum likelihood collapse into least squares; in [Chapter 2](#) the same argument with a categorical distribution produced cross-entropy. This is the general pattern: **a loss function is a noise model in disguise**. Choose how you believe the data deviates from your model, and maximum likelihood hands you the loss. The loss is how we tell the model it is doing badly $\rightarrow$ choosing it well is not a detail, it is the supervision itself.

With the loss fixed, training is one line:

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \arg \min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \ell_i(\mathbf{w}), \quad (4.1)$$

where ℓ_i is the loss on example i . Everything that follows is about minimizing this average when n is enormous.

4.2. The computational bottleneck

Gradient descent, as we left it in [Chapter 1](#), updates with the *full* gradient:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \frac{1}{n} \sum_{i=1}^n \nabla \ell_i(\mathbf{w}^{(t)}). \quad (4.2)$$

Look at the cost. One step touches every example; modern datasets have millions to billions of them, modern models have millions to billions of parameters, and training needs thousands of steps. One exact gradient per step is a luxury we cannot afford $\rightarrow$ we need to change strategy to scale up.

4.3. Stochastic gradient descent

The idea is almost cheeky: we do not need the exact gradient. A good approximation is enough. Instead of the expensive sum over all n examples, average over a small random **minibatch** $\mathcal{B}$ — think 32 examples against a million:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla \ell_i(\mathbf{w}^{(t)}). \quad (4.3)$$

Why does this work? At a *fixed* parameter vector, a uniformly sampled minibatch gradient is **unbiased**: the expected contribution of example i to a batch equals its contribution to the full average, so

$$\mathbb{E}[\nabla \mathcal{L}_{\mathcal{B}}(\mathbf{w})] = \nabla \mathcal{L}(\mathbf{w}). \quad (4.4)$$

This statement holds for independent draws with replacement and also for one uniformly chosen subset without replacement. In expectation, that cheap gradient points where the expensive gradient points. Each estimate is noisy; its average direction is right.

In practice the algorithm is:

1. Shuffle the training data.
2. Split it into minibatches of size B .
3. For each batch: compute the average gradient, update the parameters.
4. One pass through all batches is an **epoch**; repeat for multiple epochs.

i Honest fine print

There are two different facts hiding under the word “without replacement.” A single uniform subset, evaluated at a fixed $\mathbf{w}$, is still unbiased and has *less* variance than with-replacement sampling. Its variance includes the finite-population correction $(n - B)/(n - 1)$, which reaches zero at $B = n$. Random reshuffling is subtler. After the first batch changes $\mathbf{w}$, the next batch comes from the remaining examples and is statistically tied to the earlier path. It is not an independent unbiased draw of the full gradient at this *new* iterate. Dedicated random-reshuffling theory handles that dependence; the one-line proof in Equation 4.4 does not. Keep the caveat in mind; keep using the shuffle.

4.4. The two zones of SGD

Strang’s treatment of SGD describes **semi-convergence**: rapid progress early, then oscillation near the solution. His convergence discussion also isolates the assumption behind Equation 4.4: the stochastic gradient must be unbiased at the parameter vector where it is evaluated. Put those ideas together on our convex least-squares bowl. Write the batch gradient as

$$\nabla \mathcal{L}_{\mathcal{B}}(\mathbf{w}) = \nabla \mathcal{L}(\mathbf{w}) + \boldsymbol{\xi}_{\mathcal{B}}, \quad \mathbb{E}[\boldsymbol{\xi}_{\mathcal{B}}] = \mathbf{0}.$$

Far from the optimum, the full gradient is large relative to the batch noise, so most batch directions make useful progress and their mean is the full descent direction. Near the optimum, the full gradient

4. *Training: Loss and Stochastic Gradient Descent*

shrinks while batches can still disagree (“go left” — “go right”). A finite learning rate then makes the iterate bounce in what we will call the **region of confusion**:

PLAN

1. Generate one least-squares problem and solve for its empirical optimum.
2. Partition the examples into equal batches and define their gradients.
3. Compare batch descent directions far from and at the optimum.
4. Verify that their fixed-point mean equals the full descent direction, then plot both zones.

CODE

```
import torch
import numpy as np
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
x1 = torch.randn(80)
y1 = 2.5 * x1 - 1.0 + 0.4 * torch.randn(80)
A = torch.stack([x1, torch.ones_like(x1)], dim=1)
optimum = torch.linalg.lstsq(A, y1).solution

# [2]
batch_size = 4
batches = torch.randperm(len(x1)).reshape(-1, batch_size)

def gradient_at(theta: torch.Tensor, idx=slice(None)) -> torch.Tensor:
    residual = A[idx] @ theta - y1[idx]
    return 2 * A[idx].T @ residual / len(residual)

def batch_descent_directions(theta: torch.Tensor) -> torch.Tensor:
    return torch.stack([-gradient_at(theta, idx) for idx in batches])

# [3]
points = [torch.tensor([-0.5, 2.0]), optimum]
titles = ["far away: agreement in expectation",
          "at the optimum: batches disagree"]
arrow_lengths = [0.72, 0.46]

W, B = np.meshgrid(np.linspace(-1.5, 5.5, 100),
                   np.linspace(-4.0, 3.0, 100))
L = ((y1.numpy()[None, None, :] -
      (W[..., None] * x1.numpy() + B[..., None])) ** 2).mean(-1)

# [4]
fig, axes = plt.subplots(1, 2, figsize=(9.2, 3.7), sharex=True, sharey=True)
for ax, theta, title, length in zip(axes, points, titles, arrow_lengths):
    directions = batch_descent_directions(theta)
    full_direction = -gradient_at(theta)
```

```

assert torch.allclose(directions.mean(0), full_direction, atol=2e-6)

unit = directions / directions.norm(dim=1, keepdim=True).clamp_min(1e-12)
origin = theta.repeat(len(unit), 1)
ax.contour(W, B, L, levels=24, cmap="Blues", alpha=0.55)
ax.quiver(origin[:, 0], origin[:, 1], unit[:, 0], unit[:, 1],
          color="#E57200", alpha=0.32, angles="xy",
          scale_units="xy", scale=1 / length, width=0.008)

if full_direction.norm() > 1e-5:
    mean_unit = full_direction / full_direction.norm()
    end = theta + 1.15 * mean_unit
    ax.annotate("", xy=end, xytext=theta,
               arrowprops={"arrowstyle": "->", "lw": 3, "color": "#232D4B"})
else:
    ax.text(float(theta[0]), float(theta[1]) + 0.72,
           "mean direction $\approx 0$", ha="center", color="#232D4B")

ax.plot(*optimum, marker="*", ms=13, color="#2F855A")
ax.plot(*theta, marker="o", ms=5, color="#232D4B")
ax.set(title=title, xlabel="$w$", xlim=(-1.5, 5.5), ylim=(-4, 3))
ax.grid(alpha=0.12)

axes[0].set_ylabel("$b$")
axes[0].plot([], [], color="#E57200", lw=3, alpha=0.45,
             label="batch directions")
axes[0].plot([], [], color="#232D4B", lw=3,
             label="mean = full direction")
axes[0].plot([], [], marker="*", ms=10, color="#2F855A", lw=0,
             label="empirical optimum")
axes[0].legend(loc="upper right", fontsize=8)
axes[1].add_patch(plt.Circle(optimum.numpy(), 0.82, color="#722F37",
                             alpha=0.09, zorder=0))
axes[1].text(float(optimum[0]), float(optimum[1]) - 1.05,
             "region of confusion", ha="center", color="#722F37")
plt.tight_layout()
plt.show()

```

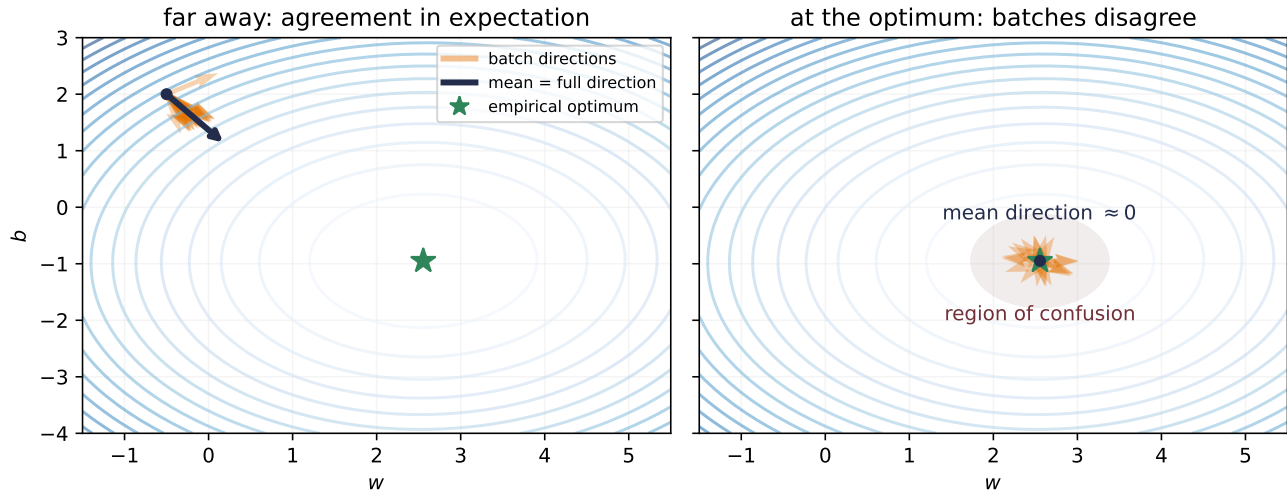


Figure 4.1.: The two zones at fixed parameter vectors. Left: equal-size batch directions vary, but their mean overlaps the full descent direction and points down the convex bowl. Right: at the empirical optimum the full gradient is essentially zero while individual batches still disagree, so finite steps create a region of confusion.

Here is the surprise: the bouncing is not merely tolerable; in some regimes it is a **feature**. A noisy step may perturb the iterate away from a saddle, a narrow basin, or a shallow trap that exact descent would follow more predictably. Minibatch noise can also bias training toward different solutions, and smaller batches sometimes improve generalization. None of these outcomes is guaranteed: the effect depends on the model, data, learning rate, and batch size. We will give that evidence its proper treatment in [Chapter 6](#). For now, treat noise as part of the algorithm, not automatically as a defect or a cure.

4.5. The learning rate

One knob dominates all others. The learning rate α scales every step, and its failure modes are asymmetric: too small wastes your compute budget crawling; too large overshoots the valley and diverges.

PLAN

1. Define the reusable `losses_for` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Implement the learning-rate triptych.
 4. Report or visualize the measured result.
-

CODE

```

# [1]
def losses_for(lr: float, steps: int = 60) -> list[float]:
    w, b, out = torch.tensor(-0.5), torch.tensor(2.0), []
    for _ in range(steps):
        err = (w * x1 + b) - y1
        out.append(float((err ** 2).mean()))
        w = w - lr * 2 * (err * x1).mean()
        b = b - lr * 2 * err.mean()
    return out

# [2]
plt.figure(figsize=(6.2, 3.4))
# [3]
for lr, style, color in [(0.005, "-", "#5379AA"), (0.12, "-", "#E57200"),
                        (1.1, "--", "#722F37")]:
    plt.semilogy(losses_for(lr), style, color=color, label=f"$\\alpha = {lr}$")
plt.xlabel("step"); plt.ylabel("loss (log scale)")
# [4]
plt.legend(); plt.tight_layout(); plt.show()

```

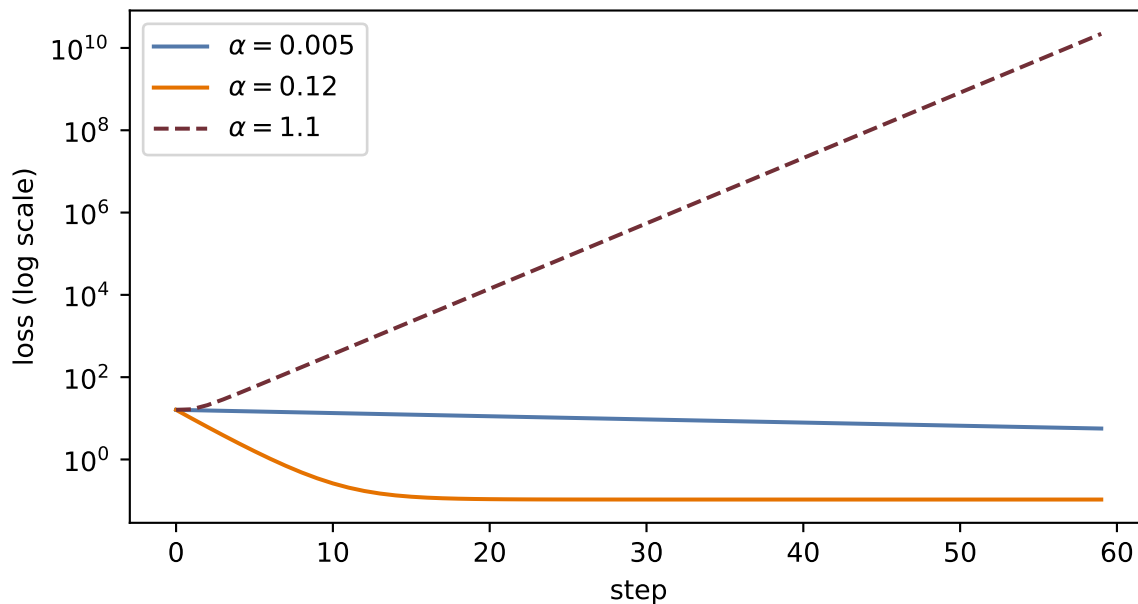


Figure 4.2.: Same problem, same steps, three learning rates. Too small crawls; too large overshoots back and forth and climbs; the middle one converges quickly.

💡 Practical rule of thumb

For the normalized toy problems here, plain SGD often starts around $\alpha = 0.1$. That is a scale-calibrated starting point, not a universal constant; Adam, unnormalized features, and very deep models can require very different values. Read the training curves: crawling $\rightarrow$ consider raising it;

oscillating or exploding $\rightarrow$ lower it. Later in training it often pays to *decay* the learning rate so the fine-tuning steps get smaller; that is a **learning-rate schedule**. One exotic-sounding relative, *warmup* (starting tiny and ramping up), will matter enormously when we train Transformers in [Chapter 14](#).

4.6. The batch size

The batch size B sets where you live on the noise–efficiency trade-off:

Batch size	Character	Trade-off
Small (1–32)	Noisy, exploratory	Sometimes a generalization benefit; slow and less hardware-efficient
Medium (32–256)	Balanced	The usual default; hardware-efficient
Large (256+)	Smooth, high throughput	Less gradient noise; often needs learning-rate and schedule retuning

With independent sampling, gradient standard deviation scales like $1/\sqrt{B}$: quadrupling the batch roughly halves the jitter. For a uniform subset without replacement, multiply by $\sqrt{(n-B)/(n-1)}$; when $B = n$, the noise is exactly zero. For $B \ll n$ the correction is near one, which is why the simpler rule is useful. There is no universal best setting here, only a dial linking statistics and hardware.

4.6.1. What a batch may estimate — and what it may not

The license behind everything above deserves one paragraph of fine print, because the rest of the book will lean on it in three different ways.

Case 1 — decomposable objectives. When the loss is a mean of per-example terms, $\mathcal{L}(\mathbf{w}) = \mathbb{E}_x[\ell(x; \mathbf{w})]$, a minibatch average is an unbiased estimator of the full objective, and its gradient an unbiased estimator of the full gradient. That is the license SGD runs on, and it is why B is a compute-and-noise dial, not a correctness dial. Cross-entropy, squared error, and every loss in Parts I–III are Case 1.

Case 2 — nonlinear functionals of aggregates. Some quantities are a *nonlinear function of an expectation*, $R(\mathbb{E}_x[s(x)])$. A batch plug-in estimate $R(\bar{s}_B)$ is generally **biased** (Jensen’s inequality gives the direction when R is convex or concave), and averaging per-batch values of R does not converge to the true quantity as training runs longer — only larger batches shrink the bias. Log-of-mean, KL between *aggregate* distributions, and correlation-style diagnostics live here. When the book meets one ([Chapter 19](#)’s aggregate-posterior trap is the flagship), it names the case and either restructures the estimand or declares the bias.

Case 3 — batch-defined objectives. Sometimes the batch is not estimating anything: it is *part of the objective’s definition*. A contrastive loss whose denominator ranges over the batch’s own candidates ([Chapter 20](#)) is a different objective at $B = 64$ than at $B = 256$ — neither is a biased version of the

other, and B becomes **protocol**, to be reported like any other design choice. Decoding-side analogues exist too: beam width in [Chapter 11](#) defines the search, it does not estimate it.

The audit habit, whenever a loss line reduces over anything: name the target, the estimator, the reduction axes, the case, and the boundary where the license fails. Chapters that meet cases 2 and 3 repeat this in one sentence; the full statement lives only here. [Appendix E](#) places these cases beside the population quantity, distribution, and uncertainty that complete the statistical contract.

4.7. Momentum: give the ball some mass

Vanilla SGD has exactly one control, α , and in narrow, curved valleys that is not enough: the iterate zigzags across the steep direction while inching along the shallow one. The fix is to give the update a memory. Keep a running **velocity** that accumulates gradients, and move along the velocity instead of the raw gradient:

$$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + \nabla \mathcal{L}(\mathbf{w}^{(t)}), \quad \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \mathbf{v}^{(t)}, \quad (4.5)$$

with $\beta \in [0.9, 0.99]$. This is the ball rolling downhill: in directions where gradients keep agreeing, speed builds; in directions where they flip sign every step, they cancel. The zigzag damps itself, and small bumps in the landscape get rolled through rather than obeyed.

4.8. Adam: adaptive steps per knob

Momentum treats every parameter alike, but some knobs sit on steep cliffs while others sit on plains. **Adam** (adaptive moment estimation) combines three ideas from the same optimization family:¹ momentum (accumulate past gradients), per-parameter adaptive learning rates (scale each knob's step by its own gradient history), and bias correction (account for both accumulators starting at zero). Formally, with elementwise operations:

$$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1 - \beta_1) \mathbf{g}^{(t)}, \quad \mathbf{s}^{(t)} = \beta_2 \mathbf{s}^{(t-1)} + (1 - \beta_2) \mathbf{g}^{(t)2}, \quad \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \frac{\hat{\mathbf{m}}^{(t)}}{\sqrt{\hat{\mathbf{s}}^{(t)} + \epsilon}}, \quad (4.6)$$

where $\hat{\mathbf{m}}, \hat{\mathbf{s}}$ are the bias-corrected accumulators. When to use what: **SGD with momentum** is simple, reliable, and strong for large models; **Adam** is the good default that works out of the box.

4.9. The race, on a problem where we know the truth

Claims about optimizers deserve a test you can rerun. We build a least-squares problem with a deliberately *ill-conditioned* landscape; the second feature is ten times the scale of the first, so the loss surface is a long narrow valley. We give all three optimizers the same 120-step, 16-example-batch budget, while using learning rates chosen for this toy: 0.003 for SGD and momentum, 0.1 for Adam. This is a mechanism comparison, not a controlled claim that one optimizer wins at a shared learning rate.

¹The per-parameter denominator is RMSProp. Adam adds bias-corrected first moments and names RMSProp directly.

4. Training: Loss and Stochastic Gradient Descent

The cell also uses `backward()` as the framework instrument previewed in Chapter 1. It supplies gradients for the race; [Chapter 5](#) opens that box next.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable `race` helper.
3. Implement the optimizer race.
4. Report or visualize the measured result.

CODE

```
# [1]
torch.manual_seed(6050)
n = 256
Xr = torch.randn(n, 2) * torch.tensor([1.0, 10.0]) # ill-conditioned by design
w_true = torch.tensor([3.0, 0.3])
yr = Xr @ w_true + 0.1 * torch.randn(n)

# [2]
def race(kind: str, steps: int = 120, B: int = 16) -> list[float]:
    torch.manual_seed(1) # same batches for everyone
    w = torch.zeros(2, requires_grad=True)
    opt = {"SGD": torch.optim.SGD([w], lr=3e-3),
          "momentum": torch.optim.SGD([w], lr=3e-3, momentum=0.9),
          "Adam": torch.optim.Adam([w], lr=0.1)}[kind]
    hist = []
    for _ in range(steps):
        idx = torch.randint(0, n, (B,))
        opt.zero_grad()
        ((Xr[idx] @ w - yr[idx]) ** 2).mean().backward()
        opt.step()
        hist.append(((Xr @ w.detach() - yr) ** 2).mean().item())
    return hist

plt.figure(figsize=(6.2, 3.6))
# [3]
for kind, color in [("SGD", "#5379AA"), ("momentum", "#232D4B"),
                   ("Adam", "#E57200")]:
    plt.semilogy(race(kind), color=color, label=kind)
plt.xlabel("step"); plt.ylabel("full-batch loss (log scale)")
# [4]
plt.legend(); plt.tight_layout(); plt.show()
```

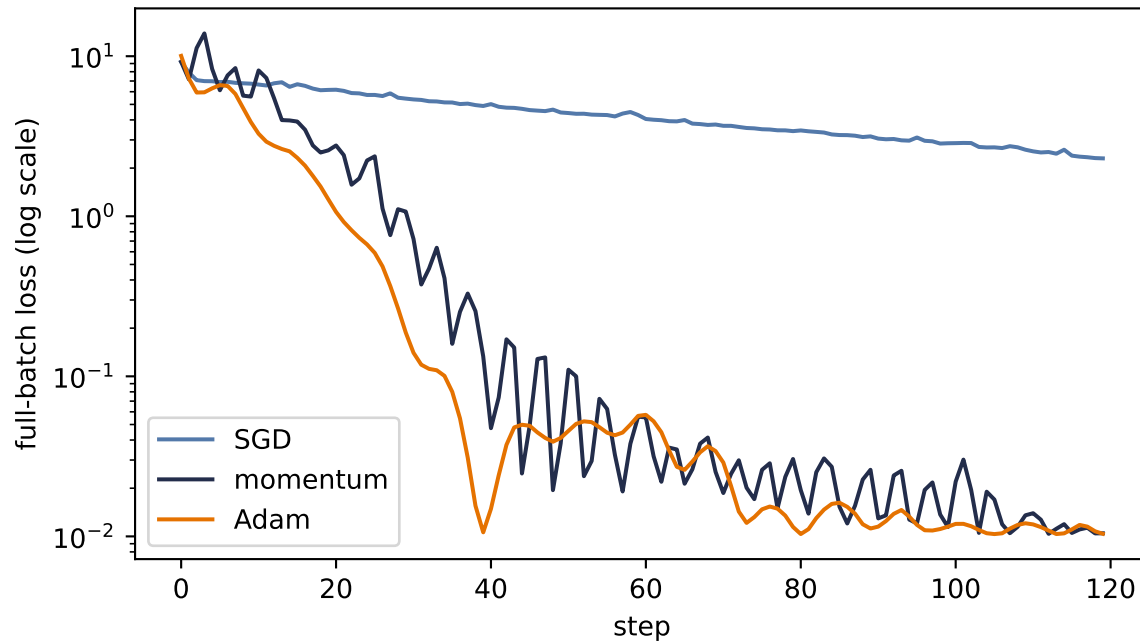


Figure 4.3.: The optimizer race on an ill-conditioned valley (feature scales 1 and 10), with an equal step budget and optimizer-appropriate learning rates (0.003 for SGD/momentum, 0.1 for Adam). Plain SGD crawls along the shallow direction; momentum damps the zigzag; Adam’s per-parameter scaling largely neutralizes the conditioning.

Read the result honestly. This is one problem, chosen to expose conditioning; it is not proof that Adam always wins (it does not — plain SGD with momentum, well tuned, still trains many of the best vision models). What the race does show is *mechanism*: when different directions of the landscape have wildly different steepness, per-direction memory (momentum) and per-parameter scaling (Adam) buy you orders of magnitude. It also whispers a practical lesson: much of this valley’s narrowness came from unscaled features $\rightarrow$ *normalize your inputs* and you fight the optimizer less (Exercise 5).

4.10. Three regularizers that live inside training

Regularization can act on the update, the hidden computation, or the duration of training. Ridge from [Chapter 1](#) returns as a training-loop citizen; the other two mechanisms make their contracts explicit here.

Weight decay. Under plain SGD, adding an L_2 penalty is equivalent (up to whether the coefficient is written as λ or $\lambda/2$) to one extra shrinkage term: $\mathbf{w} \leftarrow (1 - \alpha\lambda)\mathbf{w} - \alpha\nabla\mathcal{L}$. This is the ridge connection from [Chapter 1](#). Many PyTorch optimizers expose a `weight_decay` argument, but the equivalence needs a boundary: coupled L_2 inside an adaptive optimizer such as Adam is rescaled by that optimizer and is not the same trajectory as multiplicative shrinkage. **AdamW** decouples the decay step explicitly. Also, training recipes often exempt bias and normalization parameters rather than shrinking every parameter indiscriminately.

Dropout. Weight decay changes the update. Dropout changes the network that receives the update. Let h_i be one hidden activation, let p be its drop probability, and write $q = 1 - p$ for the probability

that it survives. During training, **inverted dropout** draws an independent mask $r_i \sim \text{Bernoulli}(q)$ and sends

$$\tilde{h}_i = \frac{r_i}{q} h_i \quad (4.7)$$

to the next layer. A dropped activation becomes zero; a surviving activation is scaled by $1/q$. The scaling is not cosmetic. Conditional on the unmasked activation,

$$\mathbb{E}[\tilde{h}_i \mid h_i] = \frac{\mathbb{E}[r_i]}{q} h_i = h_i. \quad (4.8)$$

The training-time signal therefore has the same first moment as the full signal, while each step sees a different thinned network. At evaluation time we remove the masks and use h_i itself. PyTorch's `nn.Dropout(p)` implements exactly this convention; `model.train()` turns masking on and `model.eval()` turns it off throughout the model. This train/evaluation asymmetry is part of dropout's definition, not an optional speed setting.

Here is the mechanism written directly, followed by a small seeded check. The empirical training mean is close to the input, while evaluation is deterministic and leaves the input unchanged:

PLAN

1. Define the reusable `inverted_dropout` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Implement inverted dropout and verify its scale.
-

CODE

```
# [1]
def inverted_dropout(
    h: torch.Tensor, p: float, training: bool = True
) -> torch.Tensor:
    if not training:
        return h
    if not 0.0 <= p < 1.0:
        raise ValueError("p must satisfy 0 <= p < 1")
    q = 1.0 - p
    mask = (torch.rand_like(h) < q).to(h.dtype)
    return mask * h / q

# [2]
torch.manual_seed(6050)
h = torch.tensor([1.0, -2.0, 4.0, 0.5])
draws = torch.stack([inverted_dropout(h, p=0.25) for _ in range(20_000)])
mean_output = draws.mean(0)

# [3]
```

```
print("training mean:", [round(v, 4) for v in mean_output.tolist()])
print(f"max |mean - h|: {(mean_output - h).abs().max():.4f}")
print("evaluation:", inverted_dropout(h, p=0.25, training=False).tolist())
```

```
training mean: [1.0025, -2.0057, 4.0064, 0.5024]
max |mean - h|: 0.0064
evaluation: [1.0, -2.0, 4.0, 0.5]
```

There is a useful **ensemble intuition** here, with an important calibration. Training shares weights across many masked subnetworks; evaluation uses one full network whose activation scale matches their mean. That can resemble averaging many related predictors. It is not an exact ensemble identity, because nonlinear layers and shared weights prevent the output of the full network from being the literal average of all masked-network outputs.

Early stopping. Track the validation loss during training and stop when it turns upward, even though the training loss is still falling:

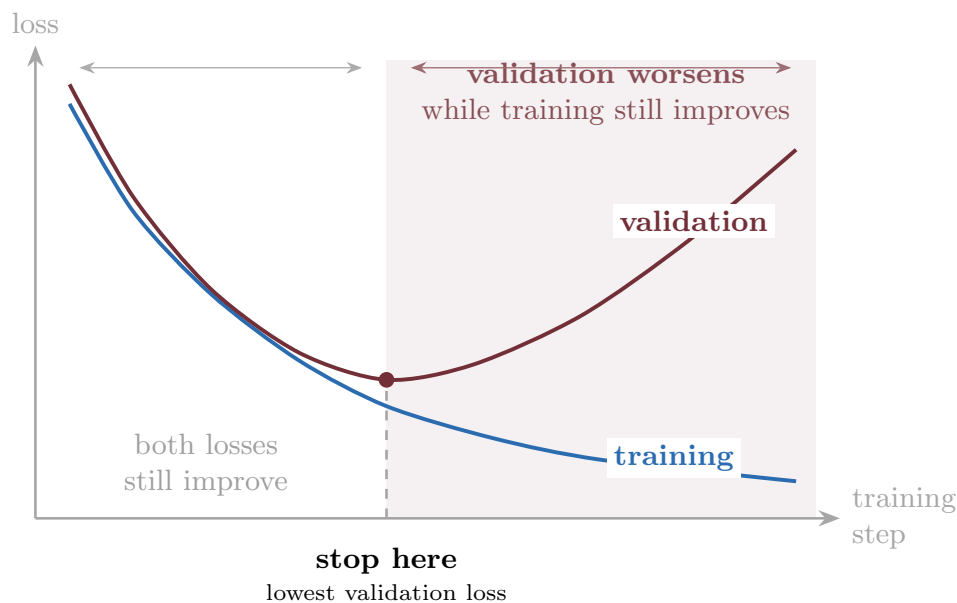


Figure 4.4.: Early stopping as model selection along one training trajectory. In this schematic, both losses initially improve; after the validation minimum, training loss keeps falling while validation loss worsens. Stop at the turn and retain that checkpoint.

All three are common, inexpensive relative to training the model, and previews: the full story of *why* constraining or randomizing training can improve generalization is the business of [Chapter 6](#).

i Check yourself

Close the book for one minute and reconstruct one update.

- Why is a minibatch gradient correct in expectation but noisy in any one draw?
- What do momentum and Adam retain from earlier gradients?
- Which operations belong inside the training loop, and which belong only at evaluation?

4.11. Okay, so — the training loop

Assemble the chapter into the loop you will run, in some form, for the rest of the book:

1. Start from (well-initialized) random parameters.
2. For each epoch: shuffle, split into minibatches of size B .
3. For each batch: training-mode forward pass (including dropout, if used) $\rightarrow$ loss (your noise model in disguise) $\rightarrow$ backward pass for the gradients $\rightarrow$ optimizer step (α , momentum or Adam, weight decay).
4. Switch to evaluation mode for validation; watch the curve, schedule the learning rate, and stop early when it turns.

That recipe, as code. The book keeps it as a tested function — **Listing 4.1** — printed here once and imported by later chapters, which then show only what they change (their model builder and their budget):

PLAN

1. Declare one reusable interface for the model factory, data, and training budget.
 2. Seed before constructing the model and optimizer.
 3. Reshuffle the examples and visit every minibatch each epoch.
 4. Predict, measure cross-entropy, backpropagate, and update.
 5. Return the trained model as the experiment artifact.
-

CODE

```
"""Listing 4.1 - the supervised training loop, importable.

Chapter 4 derives this loop and prints it; Chapters 6, 8, and 9 import it and
print only their deltas (the model builder and the budget). The loop is the
book's canonical minibatch recipe: seed, build, then repeat
predict -> measure -> step over reshuffled minibatches.
"""
from collections.abc import Callable

import torch
import torch.nn.functional as F
from torch import nn

# [1]
def fit_supervised(
    model_fn: Callable[[], nn.Module],
    X: torch.Tensor,
    y: torch.Tensor,
    *,
    epochs: int,
```

```

batch: int = 64,
lr: float = 1e-3,
seed: int = 6050,
) -> nn.Module:
    """Train a fresh model on (X, y) with cross-entropy and Adam.

    Seeding precedes construction, so a given (model_fn, seed) pair always
    starts from the same tensors; each epoch reshuffles example order.
    """
    # [2]
    torch.manual_seed(seed)
    model = model_fn()
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    # [3]
    for _ in range(epochs):
        perm = torch.randperm(len(X))
        for i in range(0, len(X), batch):
            idx = perm[i : i + batch]
            # [4]
            loss = F.cross_entropy(model(X[idx]), y[idx])
            opt.zero_grad()
            loss.backward()
            opt.step()
    # [5]
    return model

```

Two reading notes. Seeding *precedes* construction, so one (model_fn, seed) pair always trains from the same starting tensors — later chapters’ paired comparisons lean on that. And the loss line is the estimator licensed earlier in this chapter: a per-example objective averaged over a shuffled minibatch is an unbiased estimate of the full-data loss, which is precisely why the minibatch size is a compute knob rather than a correctness knob (Chapter 4).

One box in that loop is still magic: “backward pass for the gradients.” Computing millions of partial derivatives at the cost of roughly one extra forward pass is the subject of Chapter 5.

Sources and further reading

- Tieleman and Hinton, “Lecture 6.5 — RMSProp,” *COURSERA: Neural Networks for Machine Learning* (2012): the original public provenance for RMSProp’s running average of squared gradients.
- Gilbert Strang, *Linear Algebra and Learning from Data*, §VI.5, especially pp. 361–365, develops SGD’s fast start, later oscillation, unbiased-gradient assumption, and convergence-in-expectation view. The assigned excerpt is [Resources/Gil Strang/SGD.pdf](#).
- [Robbins and Monro](#), “A Stochastic Approximation Method” (1951) is the classical starting point for replacing exact quantities with noisy observations in an iterative procedure.
- [Sutskever et al.](#), “On the Importance of Initialization and Momentum in Deep Learning” (2013) shows how initialization and momentum interact with curvature in deep-network training.

- Kingma and Ba, “Adam: A Method for Stochastic Optimization” (2015) gives the algorithm, bias corrections, and motivation for adaptive moment estimates.
- Loshchilov and Hutter, “Decoupled Weight Decay Regularization” (2019) explains precisely why coupled L_2 regularization and weight decay separate under adaptive optimizers.
- Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” (2014) introduces dropout and its model-averaging motivation.

Exercises

1. **(Pencil.)** Prove Equation 4.4 for sampling with replacement: if i is drawn uniformly from $\{1, \dots, n\}$, show $\mathbb{E}[\nabla \ell_i(\mathbf{w})] = \frac{1}{n} \sum_j \nabla \ell_j(\mathbf{w})$. Where exactly does the argument use uniformity?
2. **(Code.)** In the two-zones figure, sweep $B \in \{1, 4, 16, 80\}$ and plot the final 30 steps of each path. How does the radius of the region of confusion scale with B ? Compare against $1/\sqrt{B}$ first, then include the finite-population correction. Why must the noise vanish at $B = 80$?
3. **(Code.)** Add a learning-rate schedule to the race: halve α every 30 steps for plain SGD. How much of the gap to momentum does scheduling close? What does that tell you about what momentum is *really* fixing here?
4. **(Code.)** Sweep momentum’s $\beta \in \{0, 0.5, 0.9, 0.99\}$ in the race. Explain the failure mode at the top end using the ball analogy.
5. **(Code.)** Standardize the race’s features (divide each column of $\mathbf{Xr}$ by its standard deviation) and rerun all three optimizers. How much of Adam’s advantage evaporates? State the practical moral in one sentence.
6. **(Pencil.)** (a) From Equation 4.7, show that $\text{Var}(\tilde{h}_i | h_i) = p h_i^2 / (1 - p)$. **(Code.)** (b) Estimate that variance for $p \in \{0.1, 0.5, 0.9\}$ using the seeded check. What happens to signal noise as p approaches one, and what evaluation bug appears if masking is left on?

5. Backpropagation

Recall the update rule that ended [Chapter 1](#): new parameters come from old parameters by stepping against the gradient, $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}$. For linear regression we could write that gradient down by hand. But a modern network has millions, sometimes billions, of knobs, tangled through layer after layer of composed functions $\rightarrow$ computing each partial derivative naively, one at a time, is hopeless.

Backpropagation is the algorithm that computes *all* of them, exactly, in one backward sweep whose cost is a small constant multiple of the graph’s elementary operations. It is not a new kind of calculus. It is the chain rule, *organized*. By the end of this chapter you will have derived it, implemented it by hand, rebuilt it as a tiny scalar autograd engine, and then checked that `torch.autograd` agrees with you to machine precision.

This chapter owns the mechanics. For reverse accumulation as a memory and numerical diagnostic—including fan-out accumulation, vector–Jacobian products, retained state, and checkpoint replay—continue in [The Gradient Has a Memory](#).

5.1. One neuron, one chain

Start with the smallest possible case: one neuron on one training example. The previous activation $a^{(l-1)}$ is multiplied by a weight w , giving the pre-activation $z = w a^{(l-1)} + b$; the activation function produces $a = \sigma(z)$; and a feeds a loss L .

Now turn the knob w a little. How much does L change? Follow the chain: turning w changes z ; the change in z changes a ; the change in a changes L . The chain rule multiplies these local sensitivities:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}. \quad (5.1)$$

Two things about this picture generalize to any network, however deep:

1. **The forward pass caches.** Computing L produces the intermediate values (z, a) along the way; we keep them, because the backward pass will need them.
2. **The backward pass reuses.** Each edge contributes one *local* derivative, and the derivative of L with respect to anything is the product of local derivatives along the path back to it. Nothing is recomputed $\rightarrow$ the whole sweep costs about as much as one forward pass.

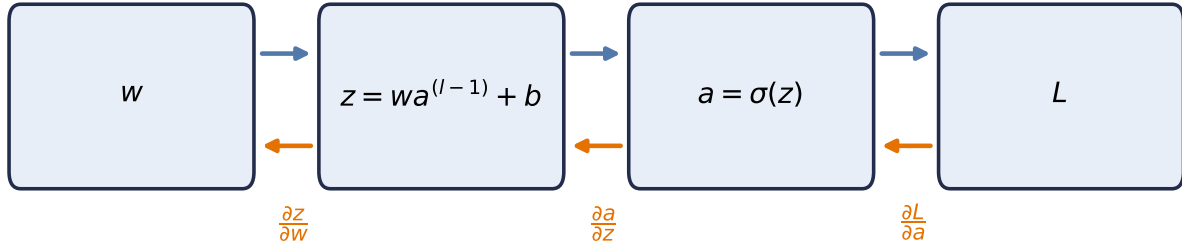


Figure 5.1.: Computation as a graph: the forward pass (top, blue) computes and caches values; the backward pass (bottom, orange) multiplies local derivatives along the same edges, in reverse.

5.2. The game of blame

For a full layer of neurons the bookkeeping threatens to become a tangled mess of sums. The cure is to pick the right intermediate quantity and track only that. At the output of the network the error is obvious: it is just the gap to the target. What is *not* obvious is how much of that error is the fault of a weight buried ten layers deep. Backpropagation sends this blame backward, from where it is obvious to where it is not.

i The blame signal

Define, for each layer l , the **blame signal**

$$\delta^{(l)} := \frac{\partial L}{\partial \mathbf{z}^{(l)}},$$

the sensitivity of the loss to that layer's *pre-activations*. It answers: how much did each neuron's z contribute to the final loss?

Everything now unfolds in four short equations. With $\odot$ denoting elementwise (Hadamard) product:

1. Start where blame is obvious (output layer L):

$$\delta^{(L)} = \frac{\partial L}{\partial \mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)}). \quad (5.2)$$

For the half-squared-error convention $L = \frac{1}{2} \|\mathbf{a}^{(L)} - \mathbf{y}\|^2$, $\partial L / \partial \mathbf{a}^{(L)}$ is exactly the prediction error $\mathbf{a}^{(L)} - \mathbf{y}$. Other conventions contribute a known scale: mean squared error, for example, contributes $2/n$ for n scalar outputs. In every case the output error is then gated by how sensitive the activation was to its input.

2. Propagate blame one layer back:

$$\delta^{(l)} = \left((\mathbf{W}^{(l+1)})^\top \delta^{(l+1)} \right) \odot \sigma'(\mathbf{z}^{(l)}). \quad (5.3)$$

5. Backpropagation

Blame flows backward through the same weights the signal flowed forward through, hence the transpose; then the local gate $\sigma'(\mathbf{z}^{(l)})$ scales it. This recursion runs from the last layer to the first.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Compute the backward blame signal through every layer.
-

CODE

```
import torch
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
sizes = [3, 4, 4, 1]
Ws = [torch.randn(sizes[i + 1], sizes[i]) for i in range(3)]

x = torch.randn(3)                                # forward pass, cached
zs, a = [], x
# [2]
for W in Ws[::-1]:                                # hidden layers: sigmoid
    zs.append(W @ a)
    a = torch.sigmoid(zs[-1])
zs.append(Ws[-1] @ a)                             # identity output, as in our 2-layer build

delta = [None] * 3                                # backward pass: eqs. 1 and 2
sp = lambda z: torch.sigmoid(z) * (1 - torch.sigmoid(z))
delta[2] = zs[2] - 1.0                             # eq. 1 (identity output, y = 1)
delta[1] = (Ws[2].T @ delta[2]) * sp(zs[1])         # eq. 2
delta[0] = (Ws[1].T @ delta[1]) * sp(zs[0])         # eq. 2 again
```

3. Convert blame into weight gradients:

$$\frac{\partial L}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^\top. \quad (5.4)$$

4. And into bias gradients:

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}. \quad (5.5)$$

Equation 5.4 deserves a pause, because its shapes tell the story. With m neurons in layer l and n in layer $l-1$: $\delta^{(l)}$ is $(m \times 1)$, and $(\mathbf{a}^{(l-1)})^\top$ is $(1 \times n)$. Their **outer product** is an $m \times n$ matrix, exactly the shape of $\mathbf{W}^{(l)}$, and entry (i, j) is the blame of neuron i times the activation of neuron j that fed it. The update for a weight is *blame out times signal in*:

5. Backpropagation

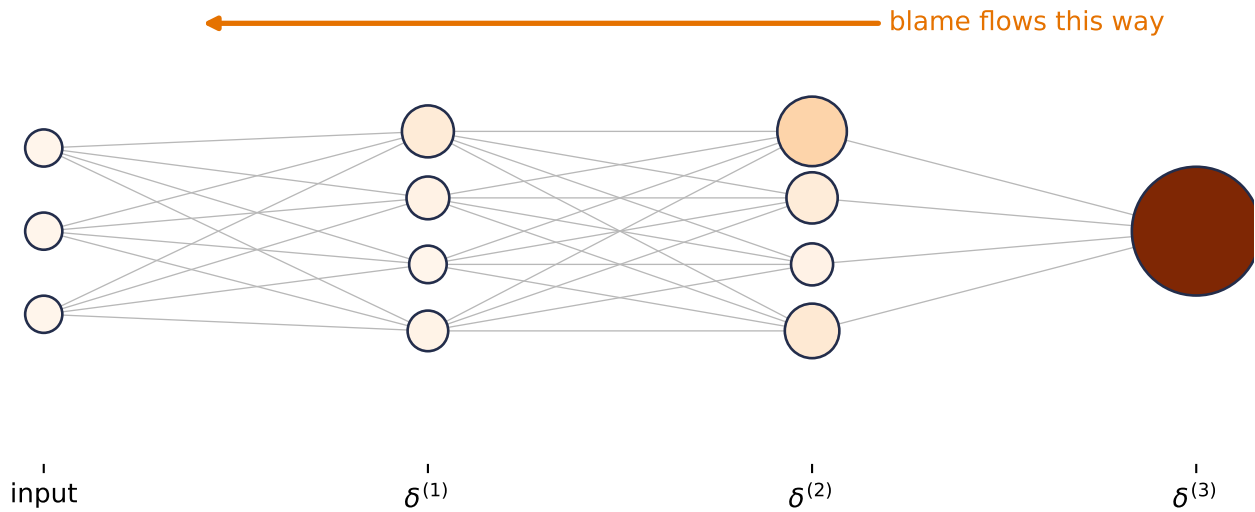


Figure 5.2.: Equations 1 and 2 in action on a real network (sizes 3–4–4–1, random weights, one example). Node size and color show each neuron’s blame $|\delta_j|$: it starts at the output, where the error is obvious, and flows backward through $W^\top$, shrinking wherever a sigmoid gate is nearly closed.

PLAN

1. Form the weight-gradient outer product.
-

CODE

```
# [1]
d2 = delta[1]                # layer-2 blame      (4,)
a1 = torch.sigmoid(zs[0])    # layer-1 activations (4,)
G = torch.outer(d2, a1)      # eq. 3
```

No messy summations: the matrix form performs them all at once, and it maps directly onto the parallel matrix multiplies GPUs are built for. The same four equations govern a toy network and a billion-parameter model; only the matrix sizes change.

5.3. Build it, then check it against autograd

Let us implement the four equations on a two-layer network, with no autograd anywhere, and then ask `torch.autograd` to differentiate the same network. If our blame algebra is right, the two answers must agree to machine precision.

5. Backpropagation

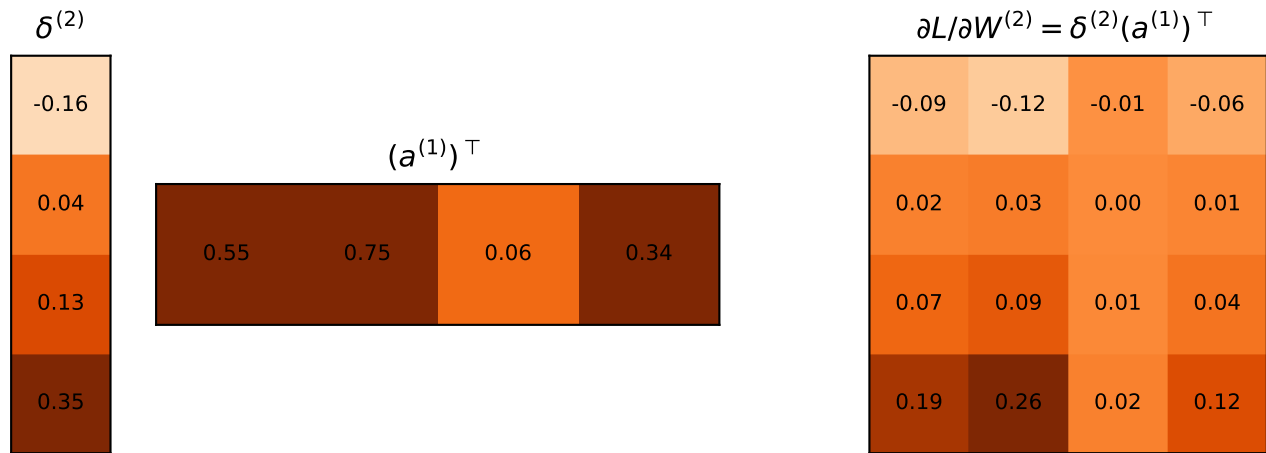


Figure 5.3.: Equation 3, pictured with the blame and activations from the network above: a (4×1) blame column times a (1×4) activation row yields the full (4×4) gradient matrix in one stroke — entry (i, j) is blame out $\times$ signal in.

PLAN

1. Load the chapter dependencies and establish reproducible state.

CODE

```
import torch

# [1]
_ = torch.manual_seed(6050)
```

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Cache the forward pass, propagate blame backward, and form both gradients.
3. Report or visualize the measured result.

CODE

```
# [1]
n, d, h = 32, 5, 8 # batch, input dim, hidden dim
X = torch.randn(n, d)
y = torch.randn(n)

W1, b1 = torch.randn(h, d) * 0.3, torch.zeros(h)
```

```

W2, b2 = torch.randn(1, h) * 0.3, torch.zeros(1)

# forward pass -- cache everything the backward pass will need
Z1 = X @ W1.T + b1                                # (n, h)
A1 = torch.sigmoid(Z1)                             # (n, h)
Z2 = A1 @ W2.T + b2                                # (n, 1)
L = ((Z2.squeeze() - y) ** 2).mean()

# backward pass -- the four equations (batch-averaged)
sig_prime = A1 * (1 - A1)                          # sigma'(z) = sigma(z)(1 - sigma(z))
delta2 = 2 * (Z2.squeeze() - y)[: , None] / n      # eq. 1 (identity output: no gate)
delta1 = (delta2 @ W2) * sig_prime                  # eq. 2: blame through W^T, gated
grads = {                                           # eqs. 3 & 4: blame out x signal in
    "W2": delta2.T @ A1, "b2": delta2.sum(0),
    "W1": delta1.T @ X,  "b1": delta1.sum(0),
}

# same network, autograd's turn
params = {k: v.clone().requires_grad_(True) for k, v in
           {"W1": W1, "b1": b1, "W2": W2, "b2": b2}.items()}
A1_ = torch.sigmoid(X @ params["W1"].T + params["b1"])
L_ = (((A1_ @ params["W2"].T + params["b2"])).squeeze() - y) ** 2).mean()
# [2]
L_.backward()

# [3]
for k in grads:
    gap = (grads[k] - params[k].grad).abs().max()
    print(f"{k}: max |ours - autograd| = {gap:.2e}")

```

```

W2: max |ours - autograd| = 0.00e+00
b2: max |ours - autograd| = 0.00e+00
W1: max |ours - autograd| = 3.73e-09
b1: max |ours - autograd| = 1.86e-09

```

The largest discrepancy is 3.73×10^{-9} , consistent with float32 rounding. The blame equations *are* what autograd computes; the framework simply builds the computation graph for us as we go and then walks it backward, caching and reusing exactly as we did.

5.4. A tiny scalar autograd engine

To remove the last trace of magic, let us build the graph ourselves. Each `Value` node remembers how it was made (its parents and the local derivative rule); calling `backward()` walks the graph in reverse topological order, accumulating blame into every node's `grad`.

This teaching implementation is adapted from Andrej Karpathy's [micrograd](#), released under the MIT License. We use only the scalar reverse-mode pattern needed for this derivation; the upstream copyright and license notice are preserved in the repository's [third-party notices](#).

PLAN

1. Implement scalar reverse-mode autodiff with a topological backward traversal.

CODE

```
# [1]
class Value:
    """A scalar that remembers its history -- a node in the computation graph."""

    def __init__(self, data: float, parents=(), backward_rule=lambda: None):
        self.data = data
        self.grad = 0.0
        self._parents = parents
        self._backward_rule = backward_rule

    def __add__(self, other: "Value | float") -> "Value":
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other))
        def rule():
            # d(out)/d(self) = d(out)/d(other) = 1
            self.grad += out.grad
            other.grad += out.grad
        out._backward_rule = rule
        return out

    def __mul__(self, other: "Value | float") -> "Value":
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data * other.data, (self, other))
        def rule():
            # product rule, one side at a time
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward_rule = rule
        return out

    def sigmoid(self) -> "Value":
        s = 1 / (1 + 2.718281828459045 ** (-self.data))
        out = Value(s, (self,))
        def rule():
            # sigma' = s (1 - s), the gate
            self.grad += s * (1 - s) * out.grad
        out._backward_rule = rule
        return out

    def backward(self) -> None:
        order, seen = [], set()
        def topo(v):
            # creators land in order first
            if v not in seen:
                seen.add(v)
```



```

        for p in v._parents:
            topo(p)
        order.append(v)
    topo(self)
    self.grad = 1.0          # dL/dL = 1: blame starts here
    for v in reversed(order):
        v._backward_rule()

```

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify micro autograd.
 3. Report or visualize the measured result.
-

CODE

```

# [1]
w, x, b = Value(0.7), Value(2.0), Value(-0.5)
a = ((w * x) + b).sigmoid()          # one neuron, forward
loss = (a + (-0.3)) * (a + (-0.3))   # squared error vs target 0.3
# [2]
loss.backward()

# analytic check: dL/dw = 2(a - y) * sigma'(z) * x
z = 0.7 * 2.0 - 0.5
s = 1 / (1 + 2.718281828459045 ** (-z))
# [3]
print(f"micro-autograd: dL/dw = {w.grad:.6f}")
print(f"by hand:          dL/dw = {2 * (s - 0.3) * s * (1 - s) * 2.0:.6f}")

```

```

micro-autograd: dL/dw = 0.337801
by hand:        dL/dw = 0.337801

```

In a few dozen lines, it is the real thing: PyTorch's autograd is this idea with tensors instead of scalars, a compiled graph walker, and thousands of derivative rules.

5.5. torch.autograd in practice: five rules

Autograd is doing exactly what we just built by hand, and knowing that mechanics tells you why each of its rules exists. These five cover nearly every autograd bug you will meet; the worked cases below isolate each path before the full network combines them.

Rule 1 — by default, only leaf tensors retain gradients. A *leaf* is a tensor you created directly (your parameters); a *non-leaf* is the result of an operation (activations, losses). After `backward()`,

5. Backpropagation

leaves have `.grad` populated while intermediate blame is used and released. Call `.retain_grad()` on a non-leaf before `backward()` only when you explicitly need its gradient for inspection.

Rule 2 — gradients accumulate. `backward()` *adds* into `.grad`. Zero them before each step (`optimizer.zero_grad()`), or every update uses the sum of all past gradients.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Backpropagate twice without clearing the leaf gradient.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
w = torch.randn(3, requires_grad=True)      # leaf: our parameter
y_hat = (w * 2).sum()                       # non-leaf: result of an operation

# [2]
y_hat.backward()

# [3]
print(f"leaf grad:      {w.grad.tolist()}")      # populated
print(f"non-leaf grad: {y_hat.grad}")           # None -- rule 1

(w * 2).sum().backward()                    # backward again, WITHOUT zeroing
print(f"after 2nd pass: {w.grad.tolist()}")      # doubled -- rule 2
```

```
leaf grad:      [2.0, 2.0, 2.0]
non-leaf grad:  None
after 2nd pass: [4.0, 4.0, 4.0]
```

Rule 3 — parameter updates happen outside the graph. For a raw leaf tensor, the update `w = w - lr * w.grad` inside autograd's view records the update and rebinds `w` to a non-leaf, so its `.grad` will not be retained by default on the next pass. Assigning a plain tensor over a registered `nn.Parameter` may instead fail loudly. The safe idiom below covers both cases:

PLAN

1. Save the leaf parameter and choose a learning rate.
 2. Pause graph recording and update the parameter in place.
 3. Verify that the update changed the value without changing leaf status.
-

CODE

```
# [1]
before = w.detach().clone()
lr = 0.1

# [2]
with torch.no_grad():
    w -= lr * w.grad

# [3]
assert not torch.equal(w, before)
assert w.is_leaf
```

Rule 4 — in-place operations can corrupt the tape. Methods ending in `_` (`add_`, `relu_`) overwrite values that the backward pass may still need from the forward cache. Inside `no_grad()` parameter updates they are safe; anywhere else, prefer creating a new tensor.

Rule 5 — the graph lives until you let it go. Every forward pass builds a fresh graph, freed after `backward()`. But any tensor you keep (say, appending `loss` to a list for plotting) keeps its whole graph alive with it. Store `loss.item()` (a plain float) or `loss.detach()` instead: `detach()` returns the same value with the history cut.

 Keep updates outside the graph

Rules 1 and 3 combine into a classic raw-tensor bug: rebinding `w = w - lr * w.grad` creates a **new non-leaf**. Its `.grad` then comes back `None` by default on the next pass. A registered `nn.Parameter` assignment can raise an error instead. In either case, update the existing parameter under `torch.no_grad()`.

5.6. The gate, and the problem with deep chains

Look again at Equation 5.3. Every backward step multiplies by $\sigma'(\mathbf{z}^{(l)})$. That derivative acts as a **gate** on the blame: wide open, and the signal passes; nearly closed, and it dies.

Now examine the sigmoid's gate. From $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, its maximum value, at $z = 0$, is exactly 0.25; away from zero the neuron *saturates* and the gate approaches 0. The sigmoid gate is at best a quarter open, and often nearly closed.

 PLAN

1. Evaluate the sigmoid and ReLU derivative gates.
-

CODE

```
import matplotlib.pyplot as plt

# [1]
z = torch.linspace(-6, 6, 300)
sig = torch.sigmoid(z)
sigmoid_gate = sig * (1 - sig)
relu_gate = (z > 0).float()
```

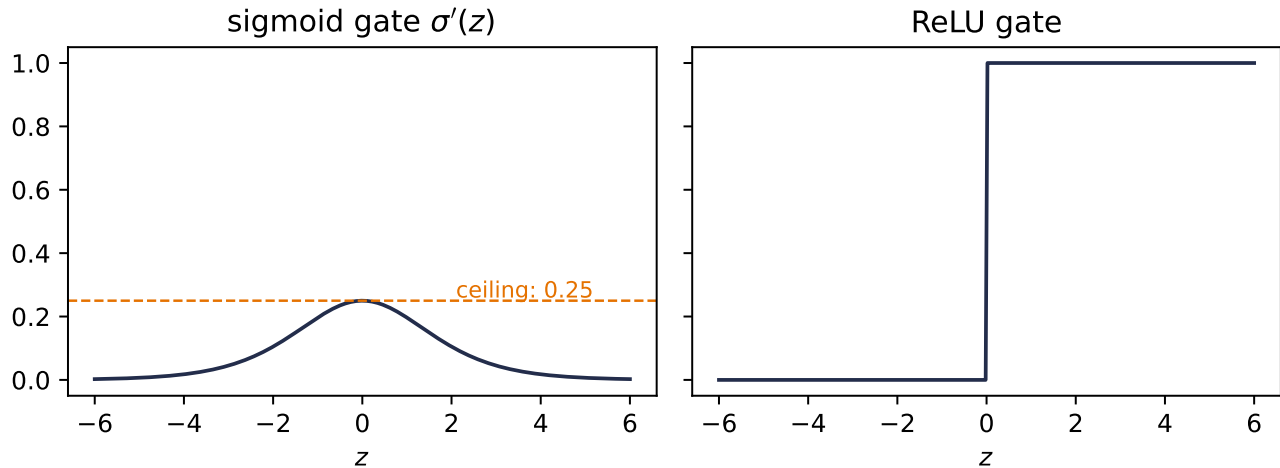


Figure 5.4.: Two gates. The sigmoid’s derivative never exceeds 0.25 and vanishes under saturation; ReLU’s is fully open (exactly 1) for every active neuron.

Here is the consequence, and it is one of deep learning’s most important failure modes. The chain rule *multiplies* local Jacobians across layers. The activation-derivative part of a sigmoid path contributes at most 0.25 per layer, so that part is bounded by 0.25^k after k gates. Ten gates in, its ceiling is below one in a million. The weight matrices and branching paths also scale the full network Jacobian, so 0.25^k is not a bound on the complete gradient. It isolates why repeated sigmoid gates strongly favor vanishing signals.

The fix is almost embarrassingly simple. $\text{ReLU}(z) = \max(0, z)$ has derivative 1 for $z > 0$ and 0 for $z < 0$. At the kink $z = 0$, PyTorch uses derivative 0; exact zeros can occur with zero biases or sparse activations. The gate is either fully open or fully closed. For every neuron that was active in the forward pass, the activation derivative contributes a factor of exactly one. The surrounding weight matrices can still scale the full signal, but the active ReLU does not shrink it: a **gradient superhighway** through the network. This, more than anything, is why ReLU and its variants are the default in deep networks.

Do not take the argument on faith; measure it. We build the same 30-layer network twice (sigmoid versus ReLU), give both copies the same He-scaled weights and input batch, and record how much gradient reaches each layer. We compute the diagnostic in float64 so the norm itself does not underflow; then we ask the practical float32 question: would a learning-rate-0.1 update actually change the first layer’s stored weights?

PLAN

1. Define the reusable `gradient_diagnostic` helper.
2. Prepare the inputs and fixed settings for the example.
3. Implement the depth experiment.
4. Report or visualize the measured result.

CODE

```

from torch import nn

# [1]
def gradient_diagnostic(
    activation: type[nn.Module], depth: int = 30, width: int = 64
) -> dict[str, object]:
    torch.manual_seed(6050)
    layers = []
    for _ in range(depth):
        layers += [nn.Linear(width, width), activation()]
    net = nn.Sequential(*layers, nn.Linear(width, 1)).double()
    with torch.no_grad():
        for layer in net:
            if isinstance(layer, nn.Linear):
                nn.init.normal_(layer.weight, std=(2 / layer.in_features) ** 0.5)
                nn.init.zeros_(layer.bias)

    out = net(torch.randn(128, width, dtype=torch.float64)).mean()
    out.backward()
    linears = [layer for layer in net if isinstance(layer, nn.Linear)][:-1]
    first = linears[0]
    grad32 = first.weight.grad.float()
    weight32 = first.weight.detach().float()
    changed = ((weight32 - 0.1 * grad32) != weight32).sum().item()
    return {
        "norms": [layer.weight.grad.norm().item() for layer in linears],
        "first_max": first.weight.grad.abs().max().item(),
        "first_nonzero": grad32.count_nonzero().item(),
        "first_changed": changed,
        "first_size": grad32.numel(),
    }

# [2]
diagnostics = {}
plt.figure(figsize=(6.2, 3.4))

# [3]
for act, color in [(nn.Sigmoid, "#232D4B"), (nn.ReLU, "#E57200")]:
    diagnostics[act.__name__] = gradient_diagnostic(act)

```

5. Backpropagation

```
plt.semilogy(diagnostics[act.__name__]["norms"], "o-", ms=3,
             color=color, label=act.__name__)
plt.xlabel("layer (0 = closest to input)")
plt.ylabel(r"$\|\partial L / \partial W^{(1)}\|$")
# [4]
plt.legend(); plt.tight_layout(); plt.show()

for name, result in diagnostics.items():
    print(f"{name:7s}: max|g0|={result['first_max']:.2e}; "
          f"nonzero={result['first_nonzero']}/{result['first_size']}; "
          f"changed={result['first_changed']}/{result['first_size']}")
```

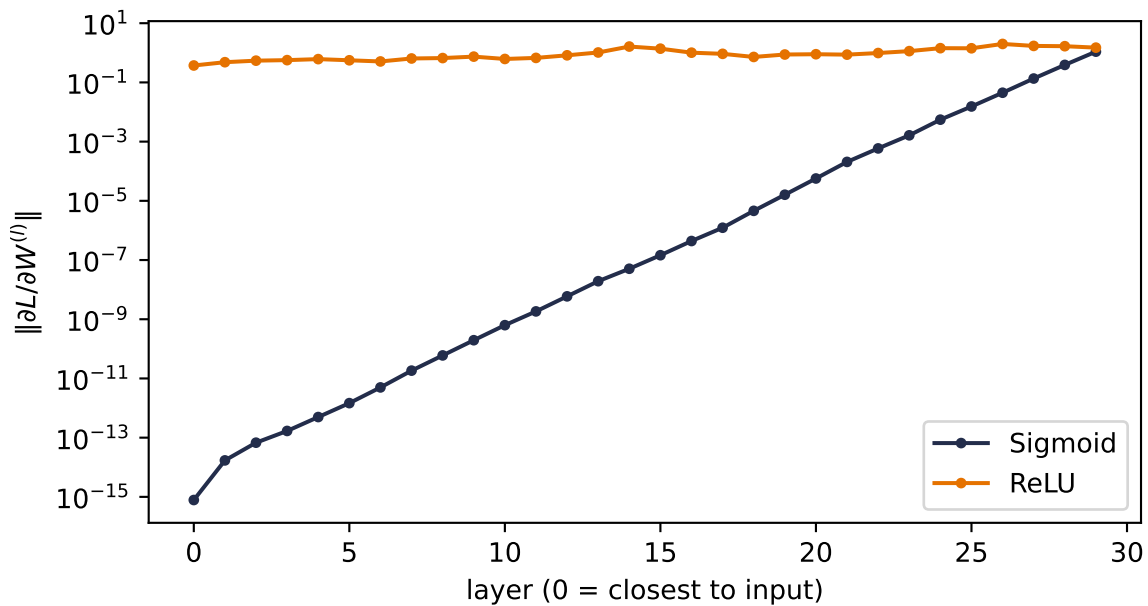


Figure 5.5.: Gradient reaching each layer of two matched 30-layer MLPs (float64 norm, log scale). With the same He-scaled weights and inputs, repeated sigmoid gates leave a nonzero but unusably small first-layer signal; ReLU preserves enough signal for a float32 update to change every first-layer weight.

```
Sigmoid: max|g0|=9.47e-17; nonzero=4096/4096; changed=0/4096
ReLU    : max|g0|=2.54e-02; nonzero=4096/4096; changed=4096/4096
```

Read the slopes, then read the update audit. The sigmoid first-layer entries are not mathematically or representationally zero: all 4,096 survive the float32 cast. They are so small, however, that subtracting $0.1g$ changes none of the stored float32 weights. The ReLU copy changes all 4,096. That distinction matters: a displayed float32 norm can underflow while individual entries remain nonzero, and nonzero gradients can still be too small to move a parameter at its current magnitude. ReLU is not magic, but its open gates make the signal usable in this matched experiment. Appendix C names these as separate representation failures: a nonzero value can survive near zero while the same update rounds away against a larger stored weight.

5.7. Initialization: setting the signal's energy

One more lever lives in this chapter, because it falls out of the same variance bookkeeping. Think of the network as a pipeline and the variance of the signal as its *energy*. If each layer multiplies the variance by more than one, activations explode after enough layers; by less than one, they vanish, and by Equation 5.3 the gradients follow. A good initialization keeps the energy roughly constant in both directions.

The analysis is one line. For $z = \sum_{i=1}^{n_{\text{in}}} w_i x_i$ with independent, zero-mean inputs and weights, $\text{Var}(z) = n_{\text{in}} \text{Var}(w) \text{Var}(x)$. Keeping $\text{Var}(z) = \text{Var}(x)$ demands

$$\text{Var}(w) = \frac{1}{n_{\text{in}}} \quad (\text{forward}), \quad \text{Var}(w) = \frac{1}{n_{\text{out}}} \quad (\text{backward}).$$

Xavier initialization splits the difference for symmetric activations like tanh and sigmoid, $\text{Var}(w) = 2/(n_{\text{in}} + n_{\text{out}})$. **He initialization** accounts for ReLU discarding half its input distribution by doubling the forward recipe, $\text{Var}(w) = 2/n_{\text{in}}$. PyTorch applies sensible defaults for you; the difference between a good and a careless choice is smooth training versus a dead or exploding network:

PLAN

1. Define the reusable `signal_std` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Signal energy vs depth.
 4. Report or visualize the measured result.
-

CODE

```
from collections.abc import Callable

# [1]
def signal_std(
    scale_fn: Callable[[torch.Tensor, int], torch.Tensor],
    depth: int = 40, width: int = 256
) -> list[float]:
    x = torch.randn(512, width)
    stds = []
    for _ in range(depth):
        W = scale_fn(torch.randn(width, width), width)
        x = torch.relu(x @ W)
        stds.append(x.std().item())
    return stds

# [2]
torch.manual_seed(6050)
schemes = {
    "naive (std = 0.05)": lambda W, n: W * 0.05,
```

5. Backpropagation

```
"naive (std = 0.12)": lambda W, n: W * 0.12,
"He (std = sqrt(2/n))": lambda W, n: W * (2 / n) ** 0.5,
}
plt.figure(figsize=(6.2, 3.4))
# [3]
for (name, fn), color in zip(schemes.items(), ["#5379AA", "#722F37", "#E57200"]):
    plt.semilogy(signal_std(fn), color=color, label=name)
plt.xlabel("layer"); plt.ylabel("std of activations")
# [4]
plt.legend(); plt.tight_layout(); plt.show()
```

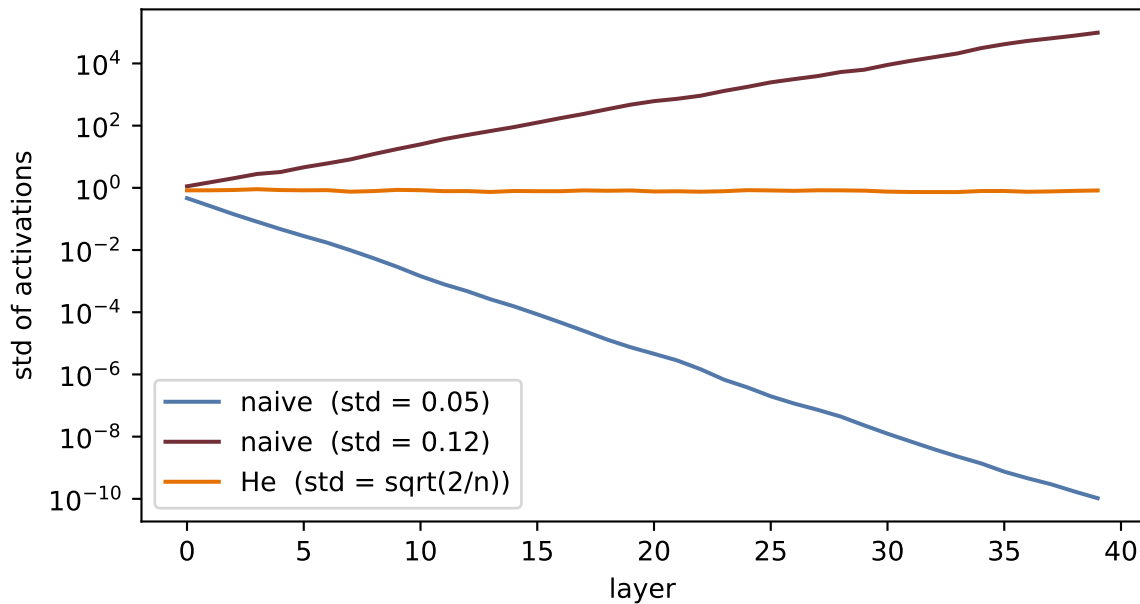


Figure 5.6.: Signal energy through 40 ReLU layers. Careless scales lose or amplify the signal exponentially; He initialization holds it steady.

Initialization gives a stable *start*. Keeping the signal stable *during* training, as the weights move, is the job of normalization layers; and for very deep networks even that is not enough, and gradients get an architectural bypass: skip connections that add a block's input straight to its output. Both belong to later chapters (normalization and the residual idea in [Chapter 9](#), and layer normalization where it becomes essential, inside the Transformer of [Chapter 14](#)). What you should carry from here is the invariant they all serve: *keep the signal, forward and backward, at constant energy*.

Check yourself

Close the book for one minute and run the chain rule backward.

- What does each layer cache on the forward pass?
- Why does the blame signal pass through a transposed weight matrix?
- How do activation derivatives and initialization together control gradient scale with depth?

5.8. Okay, so — the chain rule, organized

1. **The graph.** Any network is a computation graph; the forward pass computes and caches, the backward pass multiplies local derivatives in reverse.
2. **The blame signal.** $\delta^{(l)} = \partial L / \partial \mathbf{z}^{(l)}$ turns the tangle into four equations: start at the output (Equation 5.2), propagate through $\mathbf{W}^\top$ and the gate (Equation 5.3), then read off weight and bias gradients (Equation 5.4, Equation 5.5). The backward sweep costs a small constant multiple of the forward graph’s operations, for *every* gradient at once.
3. **We verified it:** our hand-built equations match `torch.autograd` to 10^{-8} , and a tiny scalar `Value` class reproduces the machinery end to end.
4. **The gate rules the depth.** The sigmoid-derivative product along a path is bounded above by 0.25^k after k gates; ReLU’s open gate is a gradient superhighway. Initialization sets the signal’s energy so neither direction explodes or dies at the start.

Backpropagation is the engine under every later chapter, through Chapter 20. The next time a deep model fails to learn, your first questions now have names: *is the gradient reaching the early layers? what is gating it? what is its energy?*

Sources and further reading

- Rumelhart, Hinton, and Williams, “[Learning representations by back-propagating errors](#)” (1986): the landmark short account of learning internal representations by reverse error propagation.
- Baydin et al., “[Automatic Differentiation in Machine Learning: a Survey](#)” (2018): separates reverse-mode automatic differentiation from symbolic and numerical differentiation.
- Glorot and Bengio, “[Understanding the difficulty of training deep feedforward neural networks](#)” (2010): the variance argument behind Xavier initialization.
- He et al., “[Delving Deep into Rectifiers](#)” (2015): derives the rectifier-aware initialization used in the matched depth experiment.
- CS231n, “[Gradient Checks](#)”: the centered-difference and relative-error safeguards used in Exercise 6.
- Ioffe and Szegedy, “[Batch Normalization](#)” (2015): the coupled minibatch normalization whose backward pass is audited in Exercise 7.

Exercises

1. **(Pencil.)** Derive the four blame equations for a network whose output layer uses the identity activation and squared-error loss (the network of our verification cell). Confirm that your $\delta^{(2)}$ matches the `delta2` line in the code.
2. **(Pencil.)** In Equation 5.3, why does the blame flow through $(\mathbf{W}^{(l+1)})^\top$ rather than $\mathbf{W}^{(l+1)}$? Argue from shapes: what are the dimensions of $\delta^{(l)}$, $\delta^{(l+1)}$, and $\mathbf{W}^{(l+1)}$?
3. **(Code.)** Extend the `Value` class with `tanh` and use it to train the one-neuron model from the check cell for 50 steps (write the update loop yourself). Verify the loss decreases.
4. **(Code.)** In the depth experiment, give the sigmoid network Xavier initialization (`nn.init.xavier_normal_` on each linear layer). How much of the vanishing does good initialization recover at depth 30? What does that tell you about why both fixes, initialization *and* ReLU, became standard?

5. **(Code.)** Comment out `optimizer.zero_grad()` in any training loop from [Chapter 1](#) and describe what happens to the loss curve. Explain it with one sentence about `.grad` accumulation.
6. **(Pencil.)** Derive why a forward finite difference has truncation error $O(\varepsilon)$ while a centered difference has error $O(\varepsilon^2)$. **(Audit.)** Gradient-check this chapter’s two-layer manual backward pass with $|g_a - g_n| / \max(10^{-12}, |g_a|, |g_n|)$ for $\varepsilon = 10^{-2}, \dots, 10^{-10}$ in float32 and float64. Plot relative error, explain its U-shape as truncation meets rounding, and require agreement across several randomly selected parameters under seed 6050. *Named wrong answer:* “absolute error below 10^{-4} proves the gradient” — absolute error ignores the scale of the quantity being checked.
7. **(Pencil.)** Derive $\partial\mathcal{L}/\partial x$ through BatchNorm’s batch mean and variance. Mark the direct normalized-input term and the two terms created by coupling every example through those statistics. **(Code.)** Verify all three terms against autograd on the experimentation interlude’s BatchNorm module. Then delete the two coupling terms deliberately, compare the gradient error, and rerun the interlude’s training curve under the same data, initialization, minibatch schedule, and seed panel.

6. When It Fails: Generalization in Pictures, and Inductive Bias

Part I has given us everything a course could promise: models with universal expressive power ([Chapter 3](#)), losses grounded in maximum likelihood ([Chapter 2](#)), an optimizer that scales ([Chapter 4](#)), and gradients for anything ([Chapter 5](#)). This chapter takes all of it, aims it at real images, and succeeds completely.

Then we will look a little closer, and the wheels will come off. Watching *how* they come off — in pictures rather than theorems — is the most important lesson of Part I, because the cure is the idea the entire rest of this book is built on.

6.1. Victory

The data is a subset of Fashion-MNIST, a classic benchmark: 28×28 grayscale images of clothing in ten classes. The model and training loop hold no surprises: an MLP from [Chapter 3](#), trained exactly as in [Chapter 4](#):

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable helpers: `train_mlp` and `accuracy`.
3. Fit the Chapter 4 recipe on the fixed Fashion-MNIST development split.
4. Report or visualize the measured result.

CODE

```
import torch
from torch import nn

# [1]
torch.manual_seed(6050)
development = torch.load("../data/fashion-train.pt")
holdout = torch.load("../data/fashion-test.pt")
X_dev = development["X"].float().reshape(-1, 784) / 255.0 # (1200, 784)
y_dev, classes = development["y"], development["classes"]
X_test = holdout["X"].float().reshape(-1, 784) / 255.0 # (600, 784)
y_test = holdout["y"]

split = torch.randperm(len(X_dev), generator=torch.Generator().manual_seed(6050))
```

```

fit_idx, val_idx = split[:1000], split[1000:]
X_tr, y_tr = X_dev[fit_idx], y_dev[fit_idx]           # (1000, 784), (1000,)
X_val, y_val = X_dev[val_idx], y_dev[val_idx]         # (200, 784), (200,)

from dlbook.supervised import fit_supervised          # Listing 4.1

# [2]
def train_mlp(
    X: torch.Tensor, y: torch.Tensor, hidden: int = 256,
    epochs: int = 40, seed: int = 6050
) -> nn.Module:
    # The delta from Listing 4.1 is one line: this chapter's model.
    return fit_supervised(
        lambda: nn.Sequential(
            nn.Linear(784, hidden), nn.ReLU(), nn.Linear(hidden, 10)
        ),
        X, y, epochs=epochs, seed=seed,
    )

def accuracy(net: nn.Module, X: torch.Tensor, y: torch.Tensor) -> float:
    net.eval()
    with torch.no_grad():
        return (net(X).argmax(1) == y).float().mean().item()

# [3]
net = train_mlp(X_tr, y_tr)
# [4]
print(f"train accuracy: {accuracy(net, X_tr, y_tr):.1%}")
print(f"validation accuracy: {accuracy(net, X_val, y_val):.1%}")

```

```

train accuracy: 95.0%
validation accuracy: 76.0%

```

The present MLP has no mode-dependent layer, but keep the `net.eval()` line: it disables training-only behavior such as the dropout mechanism from [Chapter 4](#) before evaluation.

Seventy-six percent on held-out validation images, from one thousand fitting examples and a few seconds of CPU. The split was fixed before any experiment, and the separate test set remains sealed until the end of the chapter. By every metric we have learned to compute, we might be done.

The rest of this chapter is about why we are not.

Before the autopsy, separate three questions that are often collapsed into “the model worked.” Each question needs different evidence.

Table 6.1.: Three claims that require three kinds of evidence. Low training error is evidence about one optimization run and, indirectly, the available capacity. It is not a generalization result.

Question	Evidence that answers it	What that evidence does not settle
Representation: can the model family express a useful rule?	A construction, approximation argument, or fitted witness	Whether the optimizer will find that rule
Optimization: did this training procedure reach low empirical risk?	The loss trace, gradients, restarts, and a final training-set check	Behavior on fresh or shifted data
Generalization: does the fitted procedure work on the distribution we care about?	Independent evaluation drawn from that declared distribution	Performance under an undeclared deployment shift

6.2. Two experiments that should worry you

6.2.1. Experiment 1: move everything two pixels

Take the validation images and shift each one two pixels to the right. Nothing about any garment changes: a coat is a coat, wherever it hangs in the frame. Any human would call the original and shifted batches equivalent for classification. Before you run the cell, commit to a number: the model scores 76% in place — what do you expect at two pixels?

PLAN

1. Define the reusable `shift_right` helper.
 2. Shift validation images while keeping the trained MLP fixed.
-

CODE

```
# [1]
def shift_right(X: torch.Tensor, px: int) -> torch.Tensor:
    img = X.reshape(-1, 28, 28)
    out = torch.zeros_like(img)
    if px > 0:
        out[:, :, px:] = img[:, :, :-px]
    else:
        out = img.clone()
    return out.reshape(-1, 784)

# [2]
for px in [0, 2]:
    print(f"shift {px}px: validation accuracy "
          f"{accuracy(net, shift_right(X_val, px), y_val):.1%}")
```

shift 0px: validation accuracy 76.0%
 shift 2px: validation accuracy 46.5%

Two pixels, and the accuracy is nearly cut in half. Push further and the collapse completes:

This is the distributional distinction prepared in Chapter 1. If T shifts an image while preserving its label, then

$$R_{T_{\#}P}(\theta) = \mathbb{E}_{(X,Y) \sim P} [\ell(f_{\theta}(T(X)), Y)], \quad (6.1)$$

where $T_{\#}P$ is the distribution obtained by transforming inputs drawn from P . The MLP generalized moderately from its fitting sample to fresh clean examples from P , but it failed the desired robustness contract under $T_{\#}P$. There is no contradiction. Clean validation and shift robustness answer different questions. The model did not violate its literal empirical objective; that objective was insufficient for the semantic and deployment contract we later imposed. Appendix E gathers the P , $T_{\#}P$, and deployment-distribution notation in one place.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Plot accuracy vs. shift.
-

CODE

```
import matplotlib.pyplot as plt

# [1]
shifts = list(range(5))
accs = [accuracy(net, shift_right(X_val, s), y_val) for s in shifts]
plt.figure(figsize=(5.6, 3.3))
# [2]
plt.plot(shifts, accs, "o-", color="#E57200", lw=2, ms=7)
plt.axhline(0.1, ls=":", color="#B8B8A8")
plt.text(0.1, 0.115, "chance (10 classes)", color="#8A8A7A", fontsize=9)
plt.xlabel("shift (pixels right)"); plt.ylabel("validation accuracy")
plt.ylim(0, 0.9); plt.xticks(shifts)
plt.tight_layout(); plt.show()
```

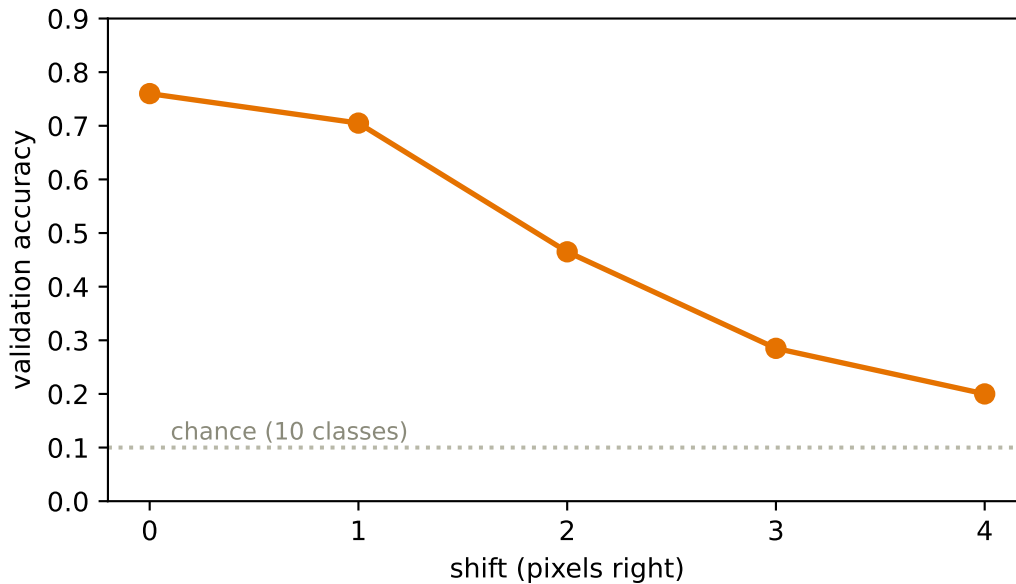


Figure 6.1.: The validation cliff. Each pixel of horizontal shift destroys accuracy although the garments themselves never changed. By four pixels the model is approaching guesswork.

6.2.2. Experiment 2: destroy the images entirely

Now the stranger experiment, which exposes a fatal flaw. Choose one fixed random permutation of the 784 pixel positions and apply it to *every* image, fit and validation alike. The displayed arrays are not recognizable pictures anymore — the ordinary grid adjacency has been destroyed. Retrain the same MLP on this scrambled world — and predict first: does the destroyed geometry cost this model five points, twenty, or nothing?

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Retrain on one fixed permutation of the pixel coordinates.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
torch.manual_seed(0)
pixel_perm = torch.randperm(784)          # one fixed scrambling for all images

# [2]
net_shuffled = train_mlp(X_tr[:, pixel_perm], y_tr)

# [3]
print(f"original images:  validation accuracy {accuracy(net, X_val, y_val):.1%}")
print(f"shuffled pixels:  validation accuracy "
```

```
f"{accuracy(net_shuffled, X_val[:, pixel_perm], y_val):.1%}"
```

```
original images:  validation accuracy 76.0%
shuffled pixels:  validation accuracy 77.0%
```

The two validation accuracies are close in this one deterministic run. That is not an uncertainty estimate, but it is enough for the structural point: after retraining, an invertible relabeling of pixel coordinates costs this MLP no visible accuracy.

Hold the two results side by side and the diagnosis writes itself:

The MLP hypothesis class and training procedure show no preference for the structure that matters, including pixel adjacency and image geometry, yet the fitted model is **hypersensitive** to a nuisance that should not matter: where the garment happens to sit. It learned the dataset without needing an image-aware representation.

6.3. The autopsy, in pictures

Why exactly does shifting hurt a model that shuffling cannot? Look at what the model actually learned. Each row of the first layer's weight matrix is a 784-vector — which means we can reshape it into a 28×28 image and look at it.¹ And we know from [Chapter 1](#) what such a row *is*: a **template**, scored against the input by dot product.

PLAN

1. Select the largest-norm first-layer templates.
-

CODE

```
# [1]
W1 = net[0].weight.detach()          # (256, 784)
order = W1.norm(dim=1).argsort(descending=True)[:16]
scale = float(W1[order].abs().max())
```

There is the whole story in one figure. Every template is **global** — a pattern stretched across the entire frame, tuned to where garments sat in the training images. Shift the input two pixels and every pixel of every template now scores against the wrong part of the garment; hundreds of learned template scores change at once. Conversely, the fixed permutation is merely an invertible relabeling of coordinates. Retraining can permute the first-layer weights in the opposite direction, so the same functions remain available even though adjacency has been destroyed. A model trained on ordinary images is not itself invariant to arbitrary pixel permutations.

Capacity was never the issue. By [Chapter 3](#)'s universal approximation theorem, this network *can represent* a perfectly shift-tolerant classifier. It has no reason to find one: nothing in its architecture, loss, or data ever told it that position is a nuisance and adjacency is meaningful.

¹Looking at first-layer weights as images is a long-standing diagnostic; the autopsy below applies it to our own model.

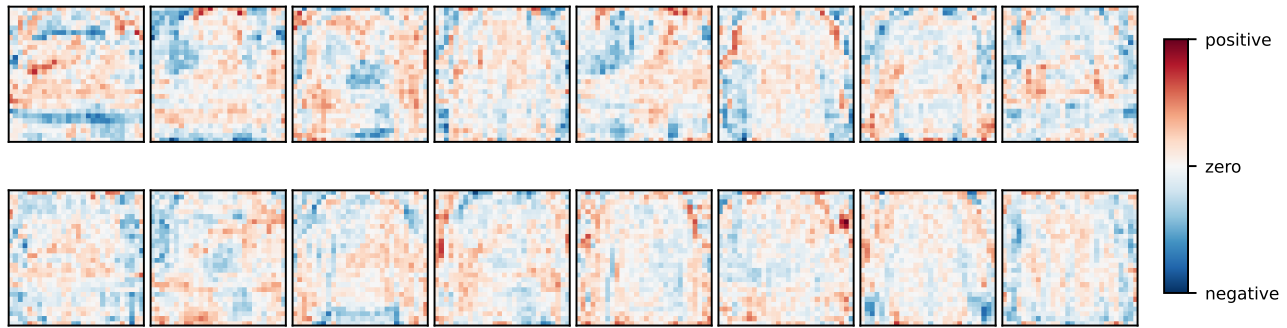


Figure 6.2.: Sixteen first-layer templates, reshaped into images. The labeled scale runs from negative weights at the blue end through pale zero to positive weights at the red end; magnitude grows darker away from zero. Each template is a global, full-frame pattern, and many are diffuse rather than recognizable garment detectors. A template that spans the whole frame changes its score when the input pattern moves.

6.4. Naming the disease

With the failure in hand, the classical concepts of Part I snap into focus.

6.4.1. Hunting the U-curve

The textbook story ([Chapter 1](#)'s cartoon) predicts that as capacity grows, validation error should fall and then *rise* as the model overfits. We now have everything needed to test the cartoon on real data: a width sweep at two dataset sizes:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Measure validation error across widths and fitting-set sizes.
-

CODE

```
# [1]
widths = [2, 4, 8, 16, 64, 256]
# [2]
width_errors = {
    n: [1 - accuracy(train_mlp(X_tr[:n], y_tr[:n], hidden=h, epochs=60),
                           X_val, y_val)
        for h in widths]
    for n in (300, 1000)
}
```

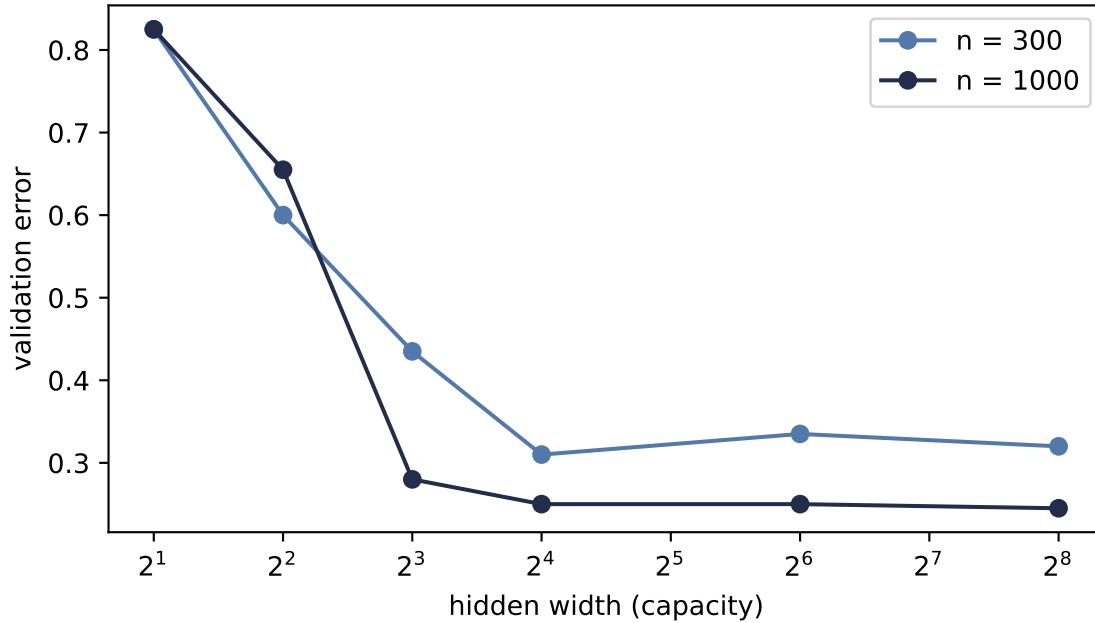


Figure 6.3.: Hunting the U-curve: validation error vs. hidden width at two fitting-set sizes. Error falls steeply as capacity grows, then flattens. In this one seeded sweep, $n=300$ shows a small late upturn while $n=1000$ does not. The result describes this regime; it does not estimate a scaling law.

Read it honestly. The left side of the cartoon is emphatically real: too little capacity underfits badly. But the dreaded right side barely materializes, with a whisper of an upturn at the small dataset and nothing at the larger one. This is exactly the caveat planted in [Chapter 1](#): the U is a helpful cartoon, not a law of nature. The optimizer, sample size, and data structure can all shape the curve. This one seeded Adam sweep does not identify their separate effects, and it does not establish a $1/n$ variance law.

Which sharpens our problem instead of solving it: **our model does not fail for having too much capacity. It fails for having no knowledge.** No point on either curve survives a two-pixel shift.

6.4.2. The curse of dimensionality

Could we fix everything with data alone, just by showing the model garments at every position? Count what that costs. Our images live in 784 dimensions; a small color image ($32 \times 32 \times 3$) needs 3,072 numbers, a one-megapixel photo three million, an ImageNet input 150,528. And covering a d -dimensional space densely requires exponentially many examples: if 9 points cover a 1-D interval at some resolution, matching that density takes 9^2 points in 2-D and 9^d in d dimensions. For $d = 784$, there are not enough atoms in the universe, at any exchange rate.

High dimensionality also breaks the very idea of “similar examples”:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Measure the nearest-to-farthest distance ratio by dimension.

CODE

```
# [1]
torch.manual_seed(6050)
dims = [2, 10, 100, 784, 5000]
ratios = []
# [2]
for d in dims:
    P = torch.rand(300, d)
    D = torch.cdist(P, P)
    ratios.append(((D + torch.eye(300) * 1e9).min(1).values /
    ↪ D.max(1).values).mean())
```

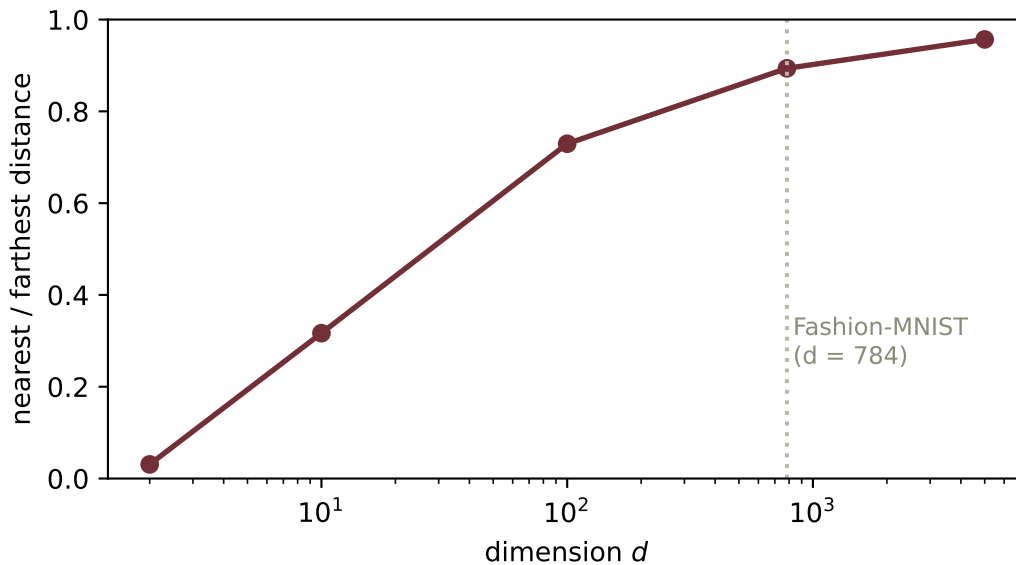


Figure 6.4.: Distance concentration for 300 points sampled uniformly from a unit cube: the average ratio of nearest-neighbor to farthest-neighbor distance, by ambient dimension. At $d=784$ the ratio reaches 0.89. This is a controlled geometry example, not a claim that natural images fill the 784-dimensional cube uniformly.

In the uniform cube, brute-force interpolation loses genuinely near neighbors as the ambient dimension grows. Natural images are highly structured and may concentrate near a much lower-dimensional set, so this experiment is not their empirical distance distribution. It shows the price of trying to cover an *unconstrained* 784-dimensional space and why useful structure must enter through data, representation, or architecture.

6.5. The cure has a name

Return to the word planted on the very first pages of this book ([Chapter 1](#)): **inductive bias** — the nudge that steers a learner toward the functions we want, among the infinitely many that fit. Our MLP’s failure is not a capacity problem or (fixably) a data problem. It is a *knowledge* problem: we knew things about images that we never told the model.

Say the two principles out loud:

1. **Locality.** Nearby pixels are related; a pixel and its neighbor jointly form edges, textures, parts. Distant pixels, far less so. (The shuffle experiment proved our MLP holds no such belief.)
2. **Translation structure.** A local detector’s response should move with the feature (**equivariance**), while a classifier’s final verdict should tolerate or ignore small changes in position (**invariance**). The shift experiment showed that our fitted MLP did neither reliably.

6.5.1. Data augmentation: put the belief in the examples

The shift cliff suggests another route. If moving a garment a little should not change its class, let T be a random label-preserving transformation drawn from a declared distribution $\mathcal{A}$. The population objective becomes

$$R_{\mathcal{A}}(\theta) = \mathbb{E}_{(X,Y) \sim P} \mathbb{E}_{T \sim \mathcal{A}} [\ell(f_{\theta}(T(X)), Y)] . \quad (6.2)$$

This is **data augmentation**, and its role is larger than making the dataset look bigger. It is a **data-side inductive bias**: the transformation distribution states which changes we believe should preserve the target. Sampling shifts from -3 to $+3$ pixels says that location within that window is a nuisance. The empirical augmented objective used in training replaces the outer expectation with the sample average. Drawing one fresh T when an example enters a batch then gives a Monte Carlo estimate of the inner expectation. Exercise 2 asks you to put that belief into the training batches and measure both what it buys on the shift curve and what it costs on clean images. The corresponding estimator and distribution contracts are summarized in [Appendix E](#).

We can now place three familiar levers side by side:

1. **Architecture-side bias** changes which functions are available or easy to learn. Local connectivity and shared convolutional filters are the route Part II will take.
2. **Objective- or training-side bias** changes which fitted solution is preferred. Weight penalties, dropout, and early stopping from [Chapter 4](#) live here.
3. **Data-side bias** changes the examples the learner is asked to agree on. Augmentation encodes a chosen family of label-preserving transformations.

These routes can work together, but they do not make identical promises. A finite set of augmented shifts encourages consistency; it does not prove exact invariance. An architectural constraint can make a symmetry exact in a layer, but may exclude useful functions. A penalty prefers some solutions without specifying which input changes should preserve a label. And augmentation is only as honest as its transformation: flipping a garment may be harmless, while flipping a digit or a directional sign may change the answer.

For Part II we begin with the architecture-side move: **build locality and translation structure into the network itself** as constraints. Constrain each template to look at a small local patch instead of

the whole frame. Then *share* that template across every position, so the detection rule itself does not depend on where. Two constraints, and both pathologies are attacked at their root. The parameter count falls too: our MLP has 203,530 parameters, while the LeNet model in [Chapter 8](#) will have 61,706, less than a third as many. Sharing is a massive economy.

💡 Constraints are knowledge

A constraint looks like a *reduction* in power: the constrained model is a strict subset of the MLP. That is exactly the point. By [Chapter 1](#)'s bias–variance lens, we are spending bias — assumptions we are confident of — to collapse the search space onto functions that respect the structure of images, slashing variance and sample complexity. Inductive bias is not merely a limitation of a model; done right, it is what you know about the world, cast into architecture, objective, training procedure, or data. And it is a dial, not a dogma: give a weakly-biased model enough data and it can win again — a trade we will meet at the frontier in [Chapter 16](#).

6.6. The pivot

Look once more at the templates figure. The fix is almost visible in it: the templates should not span the frame. They should be **small and local** — and they should **slide**, scoring every neighborhood of the image with the same little pattern, so that a sleeve found anywhere is the same sleeve.

A local template, slid across the image, scoring each position by dot product. That machine has a name; it predates deep learning by decades, and Part II opens by building it with our bare hands — then asking this book's favorite question about it:

What if the template were learnable?

6.7. One sealed test check

The architecture, optimizer, shift size, and reported metrics are now fixed. Only now do we fit the same MLP on all 1,200 development examples and open the 600-example test set once:

PLAN

1. Run one final test-set check.
 2. Report or visualize the measured result.
-

CODE

```
# [1]
net_final = train_mlp(X_dev, y_dev)
# [2]
print(f"clean test accuracy:  {accuracy(net_final, X_test, y_test):.1%}")
print(f"shift-2 test accuracy: "
      f"{accuracy(net_final, shift_right(X_test, 2), y_test):.1%}")
```

clean test accuracy: 80.8%
shift-2 test accuracy: 42.0%

The sealed check confirms the diagnostic rather than creating it: 80.8% on clean test images becomes 42.0% after the predeclared two-pixel shift.

i Check yourself

Close the book for one minute and retrieve the argument before reading the recap.

- What two experiments reveal that the MLP learned coordinates rather than image structure?
- Where can inductive bias enter: architecture, objective, or data? Give one example of each.
- Why can high clean validation accuracy coexist with collapse after a two-pixel shift?

6.8. Okay, so — the lesson of Part I

1. **Every metric can say yes while the model has learned the wrong thing.** Our MLP scored well on clean held-out data while its hypothesis class showed no adjacency preference (fixed shuffle) and its fitted predictions were hypersensitive to a nuisance shift.
2. **The autopsy is visual:** global templates, tuned to position, blind to adjacency.
3. **Capacity is not the culprit in this experiment.** The U-curve barely appears in one seeded sweep, while the uniform-cube example shows why unconstrained ambient coverage becomes expensive.
4. **The cure is inductive bias:** encode knowledge through architecture, objective, training procedure, or data. Part II chooses locality and translation structure as architectural constraints; augmentation is the data-side alternative and companion.

Deeper dive

i For the curious: how deep this rabbit hole goes

Networks can memorize random labels. Zhang et al. reported 100% training accuracy on randomly labeled CIFAR-10 and 95.20% on randomly labeled ImageNet under their finite training budget, not 100% on ImageNet. Capacity is abundant enough that fitting noise is easy (Exercise 4 reproduces the phenomenon on our subset). Capacity alone therefore does not explain generalization; the data, optimizer, and architecture must enter the account.

The bias–variance picture, updated. Our flattened U-curve is a mild case of a broader phenomenon: with modern training, test error can even *descend again* beyond the interpolation point — “double descent.” The cartoon survives as intuition for the underfit regime; the overparameterized regime plays by subtler rules.

Further reading

- Zhang, Bengio, Hardt, Recht, Vinyals, “[Understanding deep learning requires rethinking generalization](#)” (ICLR 2017), the primary source for the random-label experiment.
- Neal, “[On the Bias-Variance Tradeoff: Textbooks Need an Update](#)” (2020), the source of this book’s “cartoon, not a law” stance.

- Belkin et al., “Reconciling modern machine-learning practice and the classical bias–variance trade-off” (PNAS 2019), a primary account of double descent.
- 3Blue1Brown, *Gradient descent, how neural networks learn*, the visual companion to this chapter’s pacing, including the demonstration that a trained MLP can classify pure noise confidently.

Sources and further reading

- Xiao, Rasul, and Vollgraf, *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*: the benchmark dataset used throughout this chapter and several later mechanism studies.
- Geirhos et al., *Shortcut Learning in Deep Neural Networks*: a framework for recognizing decision rules that succeed on the benchmark but fail the intended semantic contract.
- Koh et al., *WILDS: A Benchmark of in-the-Wild Distribution Shifts*: datasets and evaluation protocols that name the deployment distribution rather than treating clean validation as a robustness test.
- Shorten and Khoshgoftaar, *A Survey on Image Data Augmentation for Deep Learning*: a broad survey of data-side transformations and the invariances they encourage.
- Bach, *Learning Theory from First Principles*: develops the grown-up capacity and generalization tools behind Exercise 6; the exercise itself stops at counting.

Exercises

1. **(Pencil.)** Explain algebraically why the pixel shuffle costs the MLP nothing: if $\mathbf{P}$ is a permutation matrix and we train on $\mathbf{P}\mathbf{x}$ instead of $\mathbf{x}$, exhibit the weight matrix that makes the two networks identical. Why does no analogous argument work for the shift?
2. **(Code.)** Augment the training set with random shifts of ± 3 pixels and retrain. Re-plot the cliff. How much robustness did augmentation buy, what did it cost in clean accuracy and compute, and — the real question — what data-side inductive bias did augmentation encode, compared to building invariance into the architecture?
3. **(Code.)** The U-curve hunt used widths up to 256. Push to 1024 and 4096 at $n = 300$, repeating each width across several seeds. Then compare Adam with minibatch SGD from Chapter 4. Which patterns are stable enough to interpret?
4. **(Code.)** Random-label audit. (a) Randomly permute the training *labels* and train a wide MLP for long enough. Report train and test accuracy. **(Audit.)** (b) Use Table 6.1 to state exactly what successful fitting supports. Explain why one successful run proves neither that the objective is convex, nor that bad local minima are absent, nor that the fitted rule will generalize.
5. **(Pencil.)** Estimate the first-layer parameter count of an MLP with 1,000 hidden units on an ImageNet-scale input ($224 \times 224 \times 3$). At one float32 per parameter, how much memory is that — and how does the number change if each unit instead sees only a $7 \times 7 \times 3$ patch?
6. **(Pencil.)** Count the labelings of N ordered points realizable by
 - (a) positive rays on a line, (b) intervals on a line, and (c) affine linear separators for N points in general position in two dimensions. Obtain $N + 1$, $N(N + 1)/2 + 1$, and $2 \sum_{i=0}^{N-1} \binom{N-1}{i}$, respectively. Compare each with 2^N . Do not invoke VC dimension: the point is to see, with arithmetic, how restricting the hypothesis menu changes its growth.

Interlude: Who Trains the Trainer? Learning by Experiment

Chapter 6 ended with a model whose code was correct, whose loss fell, and whose validation accuracy looked respectable. Then two carefully chosen experiments showed that it had learned the wrong structure. This creates a new problem. Gradient descent can train the weights, but who trains the trainer? Who chooses the learning rate, the architecture, the regularization, the stopping point, and even the comparison that we will call fair?

The answer is not another optimizer. It is **experimentation**. We make a claim, design a comparison that can answer it, spend a declared budget, and preserve enough evidence that someone else can audit our reasoning. Deep learning is mathematical, but much of its progress is also empirical — the code can run perfectly while the scientific question remains badly posed.

This interlude gives names to the discipline that the rest of the book will use. Its central distinction is simple:

Weights learn inside a run; we learn about designs across runs.

That distinction — inner training versus outer inquiry — keeps hyperparameter optimization and ablation from collapsing into one large sweep. They cooperate, but they answer different questions.

Two levels of learning

Recall the image of a weight as a knob. During one training run, the optimizer turns millions of those knobs using gradients. A **parameter** — a weight, a bias, or another model quantity updated by the training algorithm — is learned this way.

Hyperparameters — learning rate, batch size, weight decay, and training schedule, for example — control the learning process from outside that inner loop. An architectural choice, such as whether to include BatchNorm, can also sit outside the weight update. Whether we call it a hyperparameter or an experimental factor depends on the question we ask of it.

i Run, experiment, and study

- A **run** trains one design — under one configuration, seed, data split, and budget — once.
- An **experiment** compares runs to answer a predeclared question.
- A **study** is the larger collection of experiments from which we draw a scientific conclusion.

Ten runs are not automatically an experiment. Without a question and a controlled comparison, they are ten observations looking for a story.

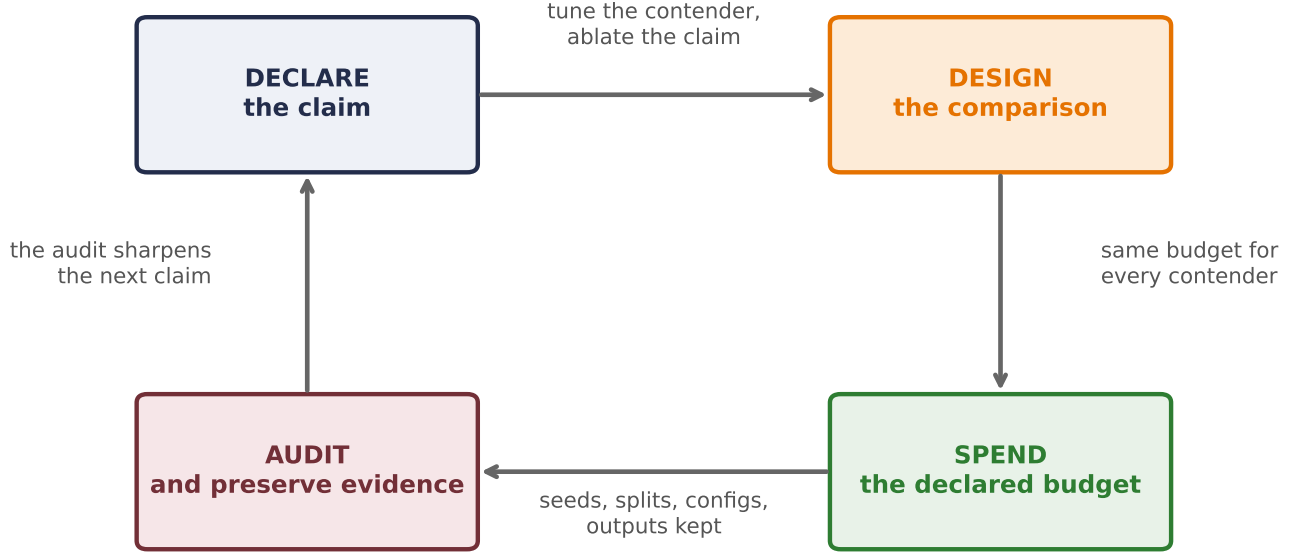


Figure EX.1: The outer loop this interlude teaches. Declare the claim before any run, design a comparison that tests the claim, spend the same declared budget, and preserve enough evidence for someone else to retrace the reasoning. Gradient descent lives inside one box; science is the whole cycle.

Let us formalize the two levels only after seeing why they are needed. Write a for a design choice — BatchNorm on or off, for example — and λ for the training configuration. Let s be the random seed and b the budget. Training returns weights

$$\mathbf{w}^*(a, \lambda, s, b) = \text{Train}(a, \lambda, s, b).$$

If J_D is a metric on dataset D , one run produces

$$J_D(a, \lambda, s, b) = \text{Metric}(f_{\mathbf{w}^*(a, \lambda, s, b)}, D).$$

The optimizer learns $\mathbf{w}^*$. We choose a , search over λ , decide which seeds and budget are credible, and interpret the result. Those outer choices are part of the claim — they do not disappear when a library automates them.

Start with the claim, not the sweep

Suppose a new normalization layer improves accuracy by two points. What did the experiment establish? The answer depends on what stayed fixed and what was optimized. Four common claim types should not be mixed; Table EX.1 separates them:

A single seeded run can be entirely legitimate — sometimes ideal — for a mechanism demonstration. It is not enough to estimate an average effect of a stochastic training procedure. Likewise, a benchmark table can rank complete systems without isolating which component caused the difference. Before opening a notebook, finish the sentence: **the claim I want this experiment to support is ...**

Neither estimand is universally fair. A shared recipe is fair to the question of a controlled intervention; separate tuning is fair to the question of attainable performance. Trouble begins when we run one and write the conclusion of the other.

i Framework preview: BatchNorm is an opaque factor here

Chapter 9 opens BatchNorm and explains its train/evaluation behavior. This interlude uses the framework layer only as a binary experimental factor—we need not credit a mechanism to compare protocols. Treat it as a labeled measuring instrument, not as a tool the reading order has introduced for general use.

A worked experiment: BatchNorm changes the learning-rate story

Let us make the distinction visible on the same Fashion-MNIST subset used in Chapter 6. The research question is intentionally small: **does adding BatchNorm help a tiny MLP?** Before running anything, we pin the contract in Table EX.2:

Notice what has not happened: we have not chosen a result and then invented a protocol around it. The split, search space, seeds, metric, budget, and selection rule are fixed first. That pre-testing discipline is what lets the prose report numbers rather than negotiate with them.

The 600-image file is **not a globally sealed test set**: Chapter 6 has already used the same book benchmark, and Chapters 8 and 9 will reuse it later. Within this worked study, its values remain decision-inert—we do not inspect them until the designs and learning rates are locked, and we make no change afterward. This is a demonstration of the protocol, not a claim that reused public data became untouched again. A new project should reserve a truly unopened audit set.

The next cell loads only the development data and defines one typed training function. The tensor shapes are explicit because experimental rigor does not excuse coding ambiguity.

PLAN

1. Implement the paired Fashion-MNIST experiment.
 2. Define the reusable helpers: `TinyMLP`, `accuracy`, and `run_once`.
-

CODE

```
from __future__ import annotations

import itertools
from copy import deepcopy

import matplotlib.pyplot as plt
import torch
from torch import Tensor, nn
import torch.nn.functional as F
```

Claim	What is held fixed?	What it can support
Mechanism demonstration	A small, transparent regime	“This behavior can occur, and here is why.”
Fixed-protocol ablation	One training recipe, paired seeds and data	“At this recipe, changing this component changed the outcome.”
Tuned design comparison	Search space and search budget, not one recipe	“Under equal tuning opportunity, this design achieved this result.”
Benchmark estimate	Locked method, metric, split, and reporting protocol	“This complete procedure attained this performance in the declared regime.”

Table EX.1: Four claim types require different controls and support different conclusions.

Item	Predeclared choice
Development data	1,200 committed images; 1,000 fit and 200 validation
Locked endpoint data	600 committed test images, already used by earlier book studies
Design factor	BatchNorm after each hidden affine layer: absent or present
Shared architecture	784 $\rightarrow$ 128 $\rightarrow$ 64 $\rightarrow$ 10, ReLU
Optimizer and budget	SGD with momentum 0.9, batch 100, at most 20 epochs
Learning-rate search	{0.003, 0.01, 0.03, 0.1}
Search seeds	6050–6054, paired across all configurations
Audit seeds	6060–6064, fixed before endpoint results are evaluated
Selection rule	Highest mean validation accuracy; restore each run’s best checkpoint

Table EX.2: The predeclared contract for the paired BatchNorm study.

```
# [1]
torch.set_num_threads(4)

development = torch.load("../data/fashion-train.pt")
X_dev = development["X"].float().unsqueeze(1) / 255.0 # (1200, 1, 28, 28)
y_dev = development["y"] # (1200,)

split = torch.randperm(
    len(X_dev), generator=torch.Generator().manual_seed(6050)
)
fit_idx, val_idx = split[:1000], split[1000:]
X_fit, y_fit = X_dev[fit_idx], y_dev[fit_idx]
X_val, y_val = X_dev[val_idx], y_dev[val_idx]

# [2]
class TinyMLP(nn.Module):
    def __init__(self, use_batchnorm: bool) -> None:
        super().__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.bn1 = nn.BatchNorm1d(128) if use_batchnorm else nn.Identity()
        self.fc2 = nn.Linear(128, 64)
        self.bn2 = nn.BatchNorm1d(64) if use_batchnorm else nn.Identity()
        self.out = nn.Linear(64, 10)

    def forward(self, x: Tensor) -> Tensor:
        x = x.flatten(1) # (B, 1, 28, 28) -> (B, 784)
        x = F.relu(self.bn1(self.fc1(x))) # (B, 784) -> (B, 128)
        x = F.relu(self.bn2(self.fc2(x))) # (B, 128) -> (B, 64)
        return self.out(x) # (B, 64) -> (B, 10)

@torch.no_grad()
def accuracy(model: nn.Module, x: Tensor, target: Tensor) -> float:
    model.eval() # freeze BatchNorm statistics
    return (model(x).argmax(1) == target).float().mean().item()

def run_once(
    seed: int,
    use_batchnorm: bool,
    learning_rate: float,
    endpoint: tuple[Tensor, Tensor] | None = None,
    epochs: int = 20,
) -> tuple[float, float]:
    torch.manual_seed(seed)
    model = TinyMLP(use_batchnorm)
    optimizer = torch.optim.SGD(
```

```
    model.parameters(), lr=learning_rate, momentum=0.9
)
batch_rng = torch.Generator().manual_seed(seed + 100_000)
best_validation = 0.0
best_state = deepcopy(model.state_dict())

for _ in range(epochs):
    model.train()
    order = torch.randperm(len(X_fit), generator=batch_rng)
    for idx in order.split(100):
        loss = F.cross_entropy(model(X_fit[idx]), y_fit[idx])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    validation = accuracy(model, X_val, y_val)
    if validation > best_validation:
        best_validation = validation
        best_state = deepcopy(model.state_dict())

    model.load_state_dict(best_state)                # evaluate the selected
↪ checkpoint
    endpoint_score = (
        accuracy(model, *endpoint) if endpoint is not None else best_validation
    )
    return best_validation, endpoint_score
```

There are two easy-to-miss hygiene details. Evaluation switches the model to `eval()` so BatchNorm stops updating its running statistics. More subtly, each run restores the checkpoint that actually won on validation. Reporting the last epoch after saying “best validation” would be a mismatch between the selection rule and the code.

Now we run the complete validation search. Five seeds — no magic threshold — form a small panel that makes run-to-run variation visible while keeping this example comfortably on a CPU.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Search four learning rates with paired seeds.
 3. Report or visualize the measured result.
-

CODE

```

# [1]
learning_rates = (0.003, 0.01, 0.03, 0.1)
search_seeds = tuple(range(6050, 6055))
search_rows: list[dict[str, float | bool | Tensor]] = []

# [2]
for use_batchnorm, learning_rate in itertools.product(
    (False, True), learning_rates
):
    scores = torch.tensor([
        run_once(seed, use_batchnorm, learning_rate)[0]
        for seed in search_seeds
    ])
    search_rows.append({
        "batchnorm": use_batchnorm,
        "learning_rate": learning_rate,
        "scores": scores,
        "mean": scores.mean().item(),
        "sd": scores.std(unbiased=True).item(),
    })

# [3]
for row in search_rows:
    label = "with BN" if row["batchnorm"] else "no BN "
    print(
        f"{label} lr={row['learning_rate']:<5} "
        f"validation={100 * row['mean']:.1f}% +/- {100 * row['sd']:.1f}%"
    )

best_row = {
    use_batchnorm: max(
        (row for row in search_rows if row["batchnorm"] == use_batchnorm),
        key=lambda row: row["mean"],
    )
    for use_batchnorm in (False, True)
}

```

no BN	lr=0.003	validation=54.5% +/- 1.5%
no BN	lr=0.01	validation=68.9% +/- 0.5%
no BN	lr=0.03	validation=74.5% +/- 1.4%
no BN	lr=0.1	validation=78.8% +/- 1.9%
with BN	lr=0.003	validation=77.3% +/- 1.3%
with BN	lr=0.01	validation=79.1% +/- 1.2%
with BN	lr=0.03	validation=78.7% +/- 0.6%
with BN	lr=0.1	validation=77.6% +/- 1.7%

Holding the recipe fixed does not produce one universal BatchNorm effect; it produces one estimate at each of the four predeclared learning rates. The paired BatchNorm-minus-no-BatchNorm validation

differences are +22.800 points at 0.003 (sample SD 1.643), +10.200 at 0.01 (SD 1.681), +4.200 at 0.03 (SD 1.681), and −1.200 at 0.1 (SD 3.402). The dramatic low-rate contrast is one end of an interaction curve, not a privileged architecture-wide effect.

The learning-rate sweep changes the conclusion. The unnormalized MLP rises from 54.5% to a best mean of 78.8% at learning rate 0.1. The BatchNorm MLP peaks at 79.1% at learning rate 0.01. After equal, design-specific tuning, the validation gap is only 0.3 points — much smaller than the seed-to-seed standard deviations of 1.9 and 1.2 points. We have learned that BatchNorm changes which learning rates are usable. We have not learned that one design wins.

Only after those configurations are selected do we evaluate the shared test file. The endpoint check uses the predeclared fresh seed panel and makes no further design or hyperparameter choices; the checkpoint rule remains the one declared above.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Evaluate the locked configurations on the shared endpoint.
-

CODE

```
# [1]
holdout = torch.load("../data/fashion-test.pt")
X_test = holdout["X"].float().unsqueeze(1) / 255.0 # (600, 1, 28, 28)
y_test = holdout["y"] # (600,)
audit_seeds = tuple(range(6060, 6065))
audit: dict[bool, dict[str, float | Tensor]] = {}

# [2]
for use_batchnorm in (False, True):
    learning_rate = float(best_row[use_batchnorm]["learning_rate"])
    test_scores = torch.tensor([
        run_once(
            seed, use_batchnorm, learning_rate, endpoint=(X_test, y_test)
        )[1]
        for seed in audit_seeds
    ])
    audit[use_batchnorm] = {
        "learning_rate": learning_rate,
        "scores": test_scores,
        "mean": test_scores.mean().item(),
        "sd": test_scores.std(unbiased=True).item(),
    }
    label = "with BN" if use_batchnorm else "no BN"
    print(
        f"{label} locked lr={learning_rate:<4} "
        f"test={100 * audit[use_batchnorm]['mean']:.1f}% "
        f"+/- {100 * audit[use_batchnorm]['sd']:.1f}%"
    )
```

```

    )

scores_by_rate = {
    (bool(row["batchnorm"]), float(row["learning_rate"])): row["scores"]
    for row in search_rows
}
fixed_rate_contrasts = {
    learning_rate: (
        scores_by_rate[(True, learning_rate)]
        - scores_by_rate[(False, learning_rate)]
    )
    for learning_rate in learning_rates
}
paired_contrasts = {
    **{"shared lr={rate:g}": values
        for rate, values in fixed_rate_contrasts.items()},
    "tuned validation": best_row[True]["scores"] - best_row[False]["scores"],
    "locked endpoint": audit[True]["scores"] - audit[False]["scores"],
}
print("paired BN - no-BN contrasts")
for label, contrasts in paired_contrasts.items():
    print(
        f"{label:<20} mean={100 * contrasts.mean():+.3f} pp  "
        f"SD={100 * contrasts.std(unbiased=True):.3f} pp"
    )

```

```

no BN      locked lr=0.1   test=78.1% +/- 0.7%
with BN    locked lr=0.01  test=76.9% +/- 0.9%
paired BN - no-BN contrasts
shared lr=0.003      mean=+22.800 pp  SD=1.643 pp
shared lr=0.01       mean=+10.200 pp  SD=1.681 pp
shared lr=0.03       mean=+4.200 pp   SD=1.681 pp
shared lr=0.1        mean=-1.200 pp   SD=3.402 pp
tuned validation     mean=+0.300 pp   SD=1.095 pp
locked endpoint      mean=-1.267 pp   SD=1.146 pp

```

The endpoint means are 78.1% without BatchNorm and 76.9% with it. The ordering—not only the gap—reversed. That reversal does not prove that removing BatchNorm is better. It tells us the honest thing: a 0.3-point validation separation in this small search was not sufficient to support a winner. The strongest conclusion is about **interaction** — BatchNorm and learning rate must be interpreted together — and about the limits of the experiment.

Because the arms share seeds, the paired contrasts are the sharper summaries. Across the four shared learning rates, the contrast slides from +22.800 to −1.200 points. After each design is tuned separately it is +0.300 points (paired sample SD 1.095), and at the locked endpoint it is −1.267 (SD 1.146). Marginal arm spreads answer a different question; the code preserves both.

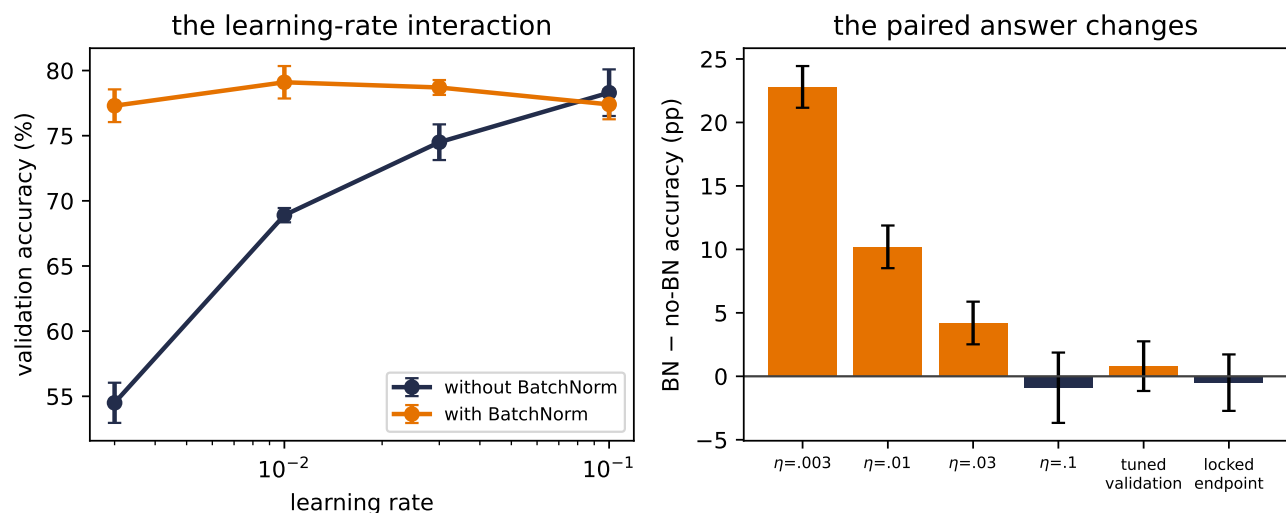


Figure EX.2: One experiment, several estimands. Left: mean validation accuracy across four predeclared learning rates. Right: paired BatchNorm-minus-no-BatchNorm contrasts at every shared rate, after separate validation tuning, and on the locked endpoint. Error bars are sample standard deviations across seeds, not confidence intervals.

Ablation is an argument about cause

An **ablation** — a deliberate removal or substitution — asks whether a component mattered. A useful ablation has five pieces:

1. **Baseline:** the complete system against which we compare.
2. **Hypothesis:** the mechanism we believe the component provides.
3. **Intervention:** the smallest change that tests that mechanism.
4. **Outcome:** a predeclared metric, not whichever curve looks favorable afterward.
5. **Scope:** the data, training protocol, seeds, and budget within which the claim holds.

“One change at a time” is the right opening discipline because it preserves attribution. If we remove BatchNorm and change the learning rate simultaneously, the raw difference cannot tell us which change produced it. Our worked example also shows why one-at-a-time is not the end of the story: components can interact.

Suppose Y_{bd} is the mean score with BatchNorm indicator $b \in \{0, 1\}$ and dropout indicator $d \in \{0, 1\}$. The BatchNorm effect without dropout is $Y_{10} - Y_{00}$; with dropout it is $Y_{11} - Y_{01}$. Their difference is the interaction:

$$I = (Y_{11} - Y_{01}) - (Y_{10} - Y_{00}).$$

If I is far from zero relative to run-to-run variation, the effect of one component depends on the other. A 2×2 design reveals that dependence; two isolated removals do not. This is why an ablation table should be read as a small factorial experiment, not as a shopping list of individually useful parts.

 Do not ablate a broken contender

If a variant cannot overfit one small batch, produces chance accuracy, or diverges under every reasonable learning rate, diagnose it before interpreting the ablation. A wiring error is not evidence against an idea. First make each contender train; then decide whether the claim calls for a shared recipe or equal tuning opportunity.

Hyperparameter optimization is a budgeted outer loop

Hyperparameter optimization (HPO) chooses the training configuration using validation performance. It is a nested problem:

$$\hat{\lambda}(a) = \arg \max_{\lambda \in \Lambda(a)} \frac{1}{|\mathcal{S}_{\text{search}}|} \sum_{s \in \mathcal{S}_{\text{search}}} J_{\text{val}}(a, \lambda, s, b),$$

while each term still requires the inner training defined earlier. There is usually no useful gradient through that whole procedure. Every evaluation — one point in the outer search — costs a training run, observations are noisy, and the search space mixes continuous, discrete, and categorical choices. HPO is therefore not “turn every knob.” It is experimental design under a budget.

Choose the space before choosing the algorithm

Three habits buy more than a fashionable search package:

- Search multiplicative quantities such as learning rate and weight decay on a **logarithmic scale**. The meaningful move is often $10^{-3} \rightarrow 10^{-2}$, not $0.001 \rightarrow 0.002$.
- Bound the space using the model, data scale, and sanity runs. A search over values known to diverge is not thorough; it is waste.
- Record conditional choices. Momentum has meaning for an optimizer that uses it; the number should not silently survive when the optimizer family changes.

For a small discrete space, a grid is transparent. In a higher-dimensional space where only a few knobs strongly affect the score, random search explores more distinct values along each important dimension for the same number of trials. Adaptive search uses earlier outcomes to choose later trials. None rescues a badly designed space.

Spend small budgets before large ones

When partial learning curves are informative, **successive halving** starts many configurations cheaply, keeps a fraction, and increases the survivors’ budgets. For example, 27 configurations run for one epoch, nine continue to a cumulative three, three continue to nine, and one continues to 27. With resumed checkpoints, the cost is $27 + 9(2) + 3(6) + 1(18) = 81$ epoch-units. Training all 27 for 27 epochs would cost 729.

The saving comes with an assumption: early rank predicts late rank well enough. A slow-starting configuration can be pruned before its advantage appears. Pruning policy is therefore part of the method

and must be logged. In sequence models later in the book, early curves can cross; “losing at epoch three” and “inferior after convergence” are not synonyms.

💡 A practical search order

Sanity check → broad, short search → narrow, longer search → lock the configuration → fresh-seed audit. If the broad search ends on a boundary, expand that boundary before declaring an optimum.

Search can also have more than one objective. Accuracy, latency, memory, and model size may trade against one another. In that case there may be no single winner, only a **Pareto frontier**: designs for which improving one objective requires worsening another. State the operating constraint before selecting a point from that frontier.

Train, validation, and test have different jobs

The three-way split is not ceremonial:

- The **training set** supplies gradients and fits $\mathbf{w}$.
- The **validation set** supplies decisions: learning rate, architecture, checkpoint, early stopping, and search-space revisions.
- The **test set** audits the locked procedure after those decisions are complete.

Every look at validation is information. After hundreds of trials, the winning configuration may partly fit quirks of that finite validation set — this is **validation overtuning**. A larger search budget can improve the chosen validation score while making the score a more optimistic estimate of future performance. Keep a test set sealed, and for high-stakes comparisons consider nested resampling or a fresh external audit.

The test set — the only split not used for decisions — is not a second validation set. If the test result disappoints and we return to change the model, that test set has joined the development process. It can still be useful, but it is no longer an untouched audit for the revised procedure.

Seeds are a panel, not decoration

Random initialization, batch order, dropout masks, and other stochastic choices can change the outcome. A seed helps make one run reproducible; it does not make the training procedure deterministic in meaning. The scientific target is usually an expectation over plausible runs, so we need a panel.

For a controlled comparison, use the same seed set and data split in both arms. Then report the per-seed differences as well as the means. This **paired-seed** design often removes background variation: if seed 6052 is difficult for both models, that shared difficulty does not masquerade as a design effect.

Be precise about the uncertainty summary:

- A **sample standard deviation across seeds** describes run-to-run spread in the chosen seed panel.
- It is not automatically a confidence interval for a population mean.
- Five seeds can expose a fragile ranking; they do not turn a toy subset into a broad benchmark.

- Never infer an average stochastic effect from one seed. A labeled one-seed mechanism demonstration is still useful when the claim stays that narrow.

Our Fashion-MNIST figure follows this discipline. Its error bars are sample standard deviations, and its conclusion is deliberately calibrated to a CPU-scale teaching regime.

The experiment ledger

If a result cannot be reconstructed, it is an anecdote. At minimum, each run should preserve:

Record	Why it matters
Question and hypothesis	Prevents the metric from inventing the story afterward
Code revision	Distinguishes an algorithmic change from a software change
Data version and split seed	Makes leakage and split drift auditable
Design and full configuration	Reconstructs both the outer and inner choices
Training seed and budget	Defines the stochastic run and resources spent
Metric definition and checkpoint rule	Prevents silent changes in what “best” means
Curves, final values, and failures	Keeps inconvenient trials in the evidence
Software and execution environment	Explains numerical and performance differences

Do not overwrite failed runs. A divergence at learning rate 0.1 defines the boundary of a search space; deleting it makes the next search less informed. Give every run a stable identifier, save the resolved configuration rather than only the defaults, and attach plots to data rather than to memory.

This is also good coding hygiene. Separate data loading, model construction, training, and evaluation so a sweep changes configuration rather than copying a notebook. Assert tensor shapes, evaluate under the correct model mode, and stop immediately on a NaN. The experiment manager can automate launches and logging later; it cannot decide what the comparison means.

A protocol for the rest of this book

When the book claims that one idea matters, read the claim through this sequence:

1. **Name the question.** Use Table EX.1: mechanism, fixed-protocol effect, tuned performance, or benchmark?
2. **Debug before searching.** Check shapes, chance baselines, and whether the model can overfit one small batch.
3. **Declare the contract.** Split, metric, seed panel, budget, search space, and selection rule.
4. **Run the smallest decisive comparison.** A good experiment isolates the question before scaling it.
5. **Inspect interactions.** If the result changes across learning rates or companion components, say so and test the relevant grid.
6. **Lock, then audit.** Select on validation; evaluate the locked procedure with fresh seeds on the sealed test set.

7. **Report the scope.** Means, spread, failures, resources, and what the experiment did not establish.

This protocol will return repeatedly. The optimizer race in Chapter 4 is a fixed recipe, so it describes that recipe. Chapter 6 uses one seeded sweep as a mechanism demonstration, not a scaling law. The CNN rematch asks whether architectural bias survives a paired shift test. Later, attention and positional encoding receive explicit ablations. The point is not to make every chapter a publication-scale benchmark. It is to make every conclusion no larger than its evidence.

For the graduate continuation, the companion volume turns this protocol into a [conditional-estimator and regime diagnostic](#) and a reusable [Incident Card](#). The first asks what one batch estimates at the current state; the second prevents a training symptom from becoming a remedy-first story.

i Check yourself

Close the book and audit one comparison you have run.

- What estimand did the comparison actually identify?
- Which choices were tuned on validation, and which result remained sealed?
- What interaction or failure could reverse the conclusion?

Okay, so — tune the contender, ablate the claim

1. **Weights and hyperparameters live in different loops.** The optimizer learns weights inside a run; we learn about designs across runs.
2. **Begin with an estimand.** A fixed recipe and a tuned-per-design comparison answer different questions, so “fair” must be defined rather than assumed.
3. **Ablation tests a claim.** Baseline, hypothesis, intervention, outcome, and scope turn a removal into an argument; factorial comparisons reveal interactions.
4. **HPO spends a declared budget.** Search space, sampling rule, pruning, seeds, and per-run resources are part of the result.
5. **Validation chooses; test audits.** Repeated validation queries can overfit, and a disappointing test result cannot be tuned away while keeping the same test set sealed.
6. **Seed spread calibrates the prose.** Report the panel honestly, distinguish standard deviation from a confidence interval, and keep conclusions inside the tested regime.

Sources and further reading

- Bergstra and Bengio, “[Random Search for Hyper-Parameter Optimization](#)” (JMLR 2012), the primary source for random search’s marginal-coverage argument.
- Li et al., “[Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization](#)” (JMLR 2018), the primary source for budgeted successive-halving brackets.
- Cawley and Talbot, “[On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation](#)” (JMLR 2010), a primary treatment of selection-induced optimism.
- Hardt and Recht, *[Patterns, Predictions, and Actions: Datasets](#)* (2022), quantifies nonadaptive best-of- k selection and shows why adaptive holdout reuse can lose its guarantee much faster.

Exercises

1. **(Pencil.)** A paper compares two optimizers under one learning rate and claims that optimizer A is intrinsically better. Write the fixed-protocol estimand it actually estimated, then write a tuned estimand that would support the broader claim. Which additional budget must the paper report?
2. **(Code.)** Extend the worked experiment to learning rates 0.001 and 0.3 without changing any other choice in Table EX.2. Do the selected rates land on a search boundary? Report the new validation table, but do not reinterpret the already-opened test set as a fresh audit.
3. **(Code.)** Add dropout as a second binary factor and run the 2×2 fixed-recipe design over paired seeds. Compute the displayed interaction contrast. Then tune the learning rate for each cell and explain why the two interaction estimates answer different questions.
4. **(Code.)** On a synthetic classification problem, compare a nine-point grid with nine random configurations over learning rate and weight decay. Repeat the entire search across several search seeds. Plot both the best-so-far curve and its spread; do not decide the winner from one search seed.
5. **(Audit.)** Choose one ablation table from a deep-learning paper. For each row, record the hypothesis, intervention, training recipe, tuning opportunity, seed count, and uncertainty summary. State the narrowest conclusion the evidence supports and one interaction the table cannot reveal.
6. **(Audit.)** With seed 6050, evaluate $k = 1, \dots, 200$ independently generated random binary classifiers on one fixed $n = 1000$ holdout set. Plot the best reported accuracy against k and compare its optimism with a $\sqrt{\log(k)/n}$ envelope. Then implement an adaptive analyst whose next classifier hill-climbs using earlier holdout feedback, and compare the gap with the nonadaptive envelope over at least 100 simulation repeats. Preserve a fresh test set that neither analyst queries. *Named wrong answer:* “the validation set remains unbiased because no gradient was trained on it” — one predetermined estimate may be unbiased, but selection by its reported values changes the chosen model.

Part II.

Vision: Learning the Filters

7. Filters and Convolution (Fixed Kernels)

Part I ended with a diagnosis and a prescription. The diagnosis: our MLP’s templates span the whole frame, so they are blind to adjacency and brittle to position ([Chapter 6](#)). The prescription, in one sentence: *make the template local, and slide it*.

This chapter builds that machine with our bare hands — no learning anywhere. It is a piece of classical engineering, decades older than deep learning, and computer-vision experts spent careers designing its templates manually. Understanding the machine in its fixed, hand-crafted form is the whole point of the chapter, because Part II’s revolution ([Chapter 8](#)) will consist of exactly one change to it.

Recall the primitive from [Chapter 1](#): a dot product scores an input against a template. When their norms are controlled, its angle component measures similarity. Everything below is that one primitive, applied locally, everywhere.

7.1. Two principles, one strategy

The failures of Chapter 6 tell us what knowledge to build in, the two principles from the mechanism:

1. **Spatial locality.** Nearby pixels are more related than distant ones. Edges, textures, and patterns emerge from *local neighborhoods*, not global pixel arrangements.
2. **Translation equivariance.** The same feature detector should work regardless of position — a cat’s ear is a cat’s ear whether it appears top-left or bottom-right.

The strategy they suggest: instead of one massive network staring at the entire image, use a *small* detector that analyzes one patch at a time, and **slide it across the image** to look for its feature everywhere. To understand what such detectors do, we first look at how vision experts used to design them by hand.

7.2. Start in 1D: the moving average

The core operation appears in its friendliest form when smoothing a noisy signal. A moving average replaces each point by the mean of its neighborhood: locality, with *uniform* weights:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Apply a one-dimensional moving-average filter.
-

CODE

```

import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
t = torch.linspace(0, 4 * torch.pi, 300)
noisy = torch.sin(t) + 0.35 * torch.randn(300)
kernel = torch.full((9,), 1 / 9)  # uniform local weights
# [2]
smooth = F.conv1d(noisy[None, None], kernel[None, None]).squeeze()

```

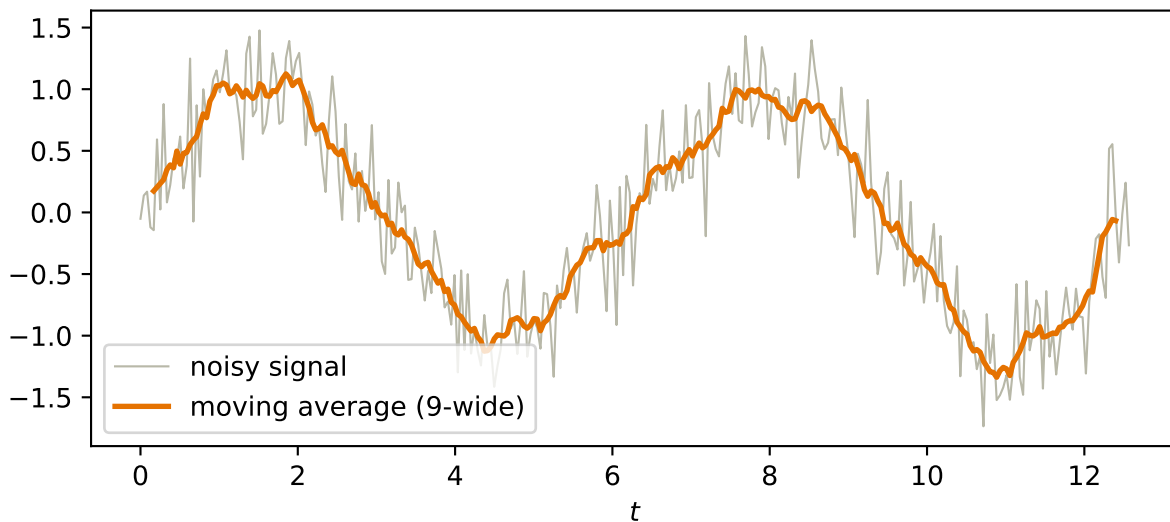


Figure 7.1.: A window of uniform weights, slid along a noisy signal, averaging as it goes: the simplest sliding-window operation, and already recognizably a denoiser.

7.3. To 2D: the recipe

The image version is the same idea with a small 2-D window, and it is worth stating as the five-step recipe, set beside the three lines that carry it:

PLAN

1. Define a small window of weights, the **kernel** (for a blur: all positive, summing to 1).
2. Place the kernel over a patch of the image.
3. Multiply elementwise: kernel weights against the pixels underneath.
4. Sum the products into one output pixel. (*Steps 3–4 are one dot product: the patch, scored against a small template.*)
5. **Slide the window** to the next location and repeat, producing a new image.

CODE

```
def manual_conv2d(x: torch.Tensor, k: torch.Tensor) -> torch.Tensor:
    """Cross-correlate an image and kernel over a view of sliding patches."""
    kh, kw = k.shape # [1]
    patches = x.unfold(0, kh, 1).unfold(1, kw, 1) # [2] [5]
    return (patches * k).sum(dim=(-1, -2)) # [3] [4]

def conv2d(x: torch.Tensor, k: torch.Tensor) -> torch.Tensor:
    return F.conv2d(x[None, None], k[None, None]).squeeze()

torch.manual_seed(6050)
x_test = torch.rand(10, 12)
k_test = torch.randn(3, 3)
gap = (manual_conv2d(x_test, k_test) - conv2d(x_test, k_test)).abs().max()
print(f"recipe vs. torch.conv2d: max |diff| = {gap:.1e}")
```

```
recipe vs. torch.conv2d: max |diff| = 4.8e-07
```

Read the markers and one thing stands out: steps 2 and 5 share a line. The plan moves one window at a time; `unfold` places *every* window at once, and the next line scores them all in parallel. That gap between a sequential plan and a vectorized kernel is where most tensor bugs live, which is why the last line checks this implementation against the framework's own — the build-then-verify habit of this book.

Also worth noticing before we move on: the output is smaller than the input. A $k \times k$ kernel on an $n \times n$ image produces $(n - k + 1) \times (n - k + 1)$ outputs, because the window must stay inside the frame. What to do about that (and about sliding in bigger steps) is [Chapter 8](#)'s business.

7.4. The filter zoo

Here is the remarkable part: the recipe never changes — *only the numbers in the kernel do*. Different numbers produce completely different specialists. Three classics, applied to a synthetic test scene and to a boot from our Fashion-MNIST subset:

- **Blur**: uniform positive weights summing to 1, the 2-D moving average.
- **Sharpen**: amplify each pixel's difference from its neighbors.
- **Edge detection** (Sobel): positive and negative weights summing to *zero* — over any flat region the products cancel and the output is silent; over an edge, a sharp change in values, the sum swings large. A detector that answers one question: *does intensity change here, in my direction?*

PLAN

1. Define the reusable `make_shapes` helper.
2. Prepare the inputs and fixed settings for the example.
3. Implement the zoo.

CODE

```
# [1]
def make_shapes(n: int = 64) -> torch.Tensor:
    img = torch.zeros(n, n)
    img[10:26, 8:30] = 0.9 # rectangle
    yy, xx = torch.meshgrid(torch.arange(n), torch.arange(n), indexing="ij")
    img[((yy - 44) ** 2 + (xx - 20) ** 2) < 100] = 0.6 # disk
    stripes = ((xx + yy) % 12 < 5).float() * 0.8
    img[8:56, 40:60] = stripes[8:56, 40:60] # diagonal stripes
    return img

# [2]
development = torch.load("../data/fashion-train.pt")
boot = development["X"][(development["y"] == 9).nonzero()[0, 0]].float() / 255.0

kernels = {
    "original": torch.tensor([[0., 0., 0.], [0., 1., 0.], [0., 0., 0.]]),
    "blur": torch.full((3, 3), 1 / 9),
    "sharpen": torch.tensor([[0., -1., 0.], [-1., 5., -1.], [0., -1., 0.]]),
    "Sobel (vert.)": torch.tensor([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]]),
    "Sobel (horiz.)": torch.tensor([[-1., -2., -1.], [0., 0., 0.], [1., 2., 1.]]),
}

fig, axes = plt.subplots(2, 5, figsize=(7.8, 3.4))
# [3]
for row, image in enumerate([make_shapes(), boot]):
    for col, (name, k) in enumerate(kernels.items()):
        out = conv2d(image, k)
        axes[row, col].imshow(out, cmap="gray")
        axes[row, col].set_xticks([]); axes[row, col].set_yticks([])
        if row == 0:
            axes[row, col].set_title(name, fontsize=9)
plt.tight_layout(); plt.show()
```

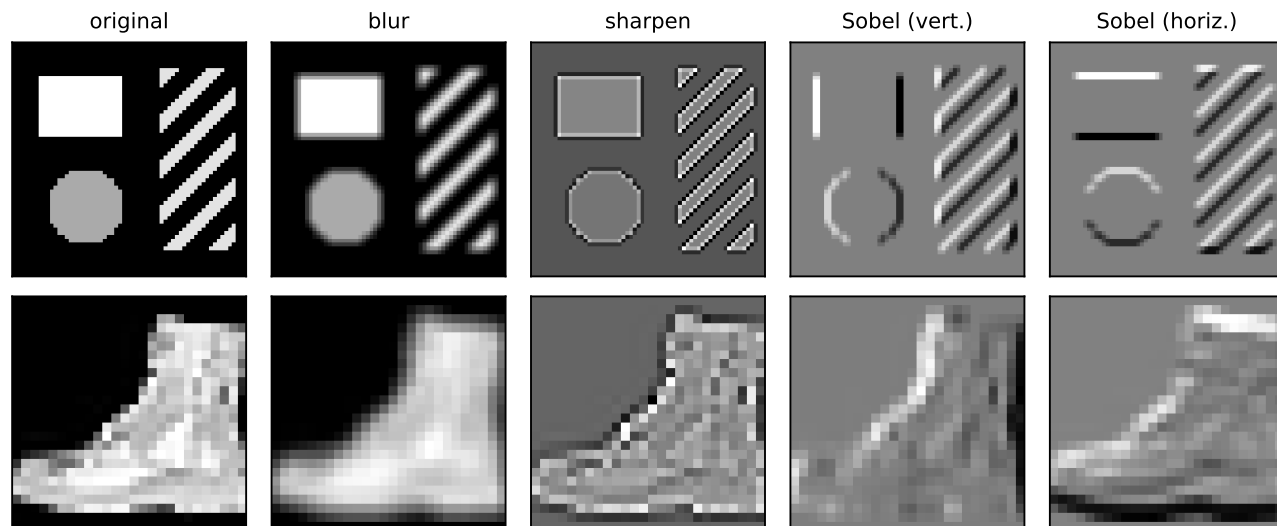


Figure 7.2.: One recipe, five kernels. Each column applies a different 3×3 kernel to the same inputs: identity, blur (local average), sharpen, and two Sobel edge detectors. The vertical-edge kernel fires on vertical boundaries; the horizontal one fires on horizontal boundaries.

Study the two Sobel columns for a moment. The vertical-edge detector lights up on the rectangle’s sides and stays silent on its top and bottom; the horizontal detector does the reverse; the diagonal stripes excite both. Nine numbers, and we have built a primitive *feature detector* — precisely the “small network analyzing one patch” of our strategy, except nobody has learned anything yet.

7.5. Naming the operation

This sliding sum-of-products has a proper name: **cross-correlation**. With kernel V of half-width Δ and image X :

$$[H]_{i,j} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [V]_{a,b} [X]_{i+a,j+b}. \quad (7.1)$$

i “Convolution”, with a small asterisk

The deep learning community universally calls this operation a **convolution**, and so will we. A mathematician’s convolution flips the kernel before sliding; since our kernels will soon be *learned* parameters, a flipped kernel is just another kernel, and the distinction is inconsequential in practice (Exercise 4 makes you confirm this).

7.6. The property we paid for: equivariance

Now collect the reward that Chapter 6’s MLP could never have. Away from boundaries, slide-and-score treats every position identically $\rightarrow$ if the input shifts, the output shifts *with it*, unchanged in content. That is **translation equivariance**, and it need not be taken on faith:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Compare filtering after a shift with shifting after filtering.

CODE

```
# [1]
img = make_shapes()
sobel = kernels["Sobel (vert.)"]
s = 5

shifted = torch.zeros_like(img)
shifted[:, s:] = img[:, :-s]           # input shifted right by 5

A = conv2d(shifted, sobel)              # filter the shifted image
B = conv2d(img, sobel)                  # filter, THEN shift the result
B_shifted = torch.zeros_like(B)
B_shifted[:, s:] = B[:, :-s]

interior = (A - B_shifted)[:, s + 2:]   # compare away from the blank border
# [2]
print(f"filter(shift(x)) vs shift(filter(x)): max |diff| =
      ↪ {interior.abs().max():.1f}")
```

```
filter(shift(x)) vs shift(filter(x)): max |diff| = 0.0
```

Exactly zero on the compared interior. The detector's report moves with the garment. Contrast Chapter 6: the MLP's global templates degraded under a 2-pixel shift because position was baked into every weight. Here, position cannot matter to *what* is detected on the interior, only to *where* the detection lands. At the frame boundary, cropping, padding, or fill rules can break exact equality; that is why the code excludes the blank-border region.

One precision worth keeping sharp, because the two words will both matter: **equivariant** means the output moves along with the input (what convolution gives us); **invariant** means the output does not change at all (what a classifier ultimately wants — “boot”, regardless of position). Equivariance is the raw material; in [Chapter 8](#), pooling will spend some position information to buy local shift tolerance. Exact invariance is a stronger property and must be tested, not presumed.

7.7. The matrix view: a structured, weight-shared linear map

One more pair of glasses. Every output pixel is a dot product $\rightarrow$ the whole operation is *linear* $\rightarrow$ cross-correlation could be written as one enormous matrix multiplying the flattened image. But it is a matrix with a rigid structure: almost everywhere zero (locality: each row touches only one patch), and the same nine numbers repeating along its diagonals (sharing: every row is the same template, relocated).

7. Filters and Convolution (Fixed Kernels)

That view makes the economics explicit. A dense layer from our 28×28 images to an equal-sized output would own $784 \times 784 \approx 615,000$ weights. The Sobel detector owns **nine** — reused at every position. This is Chapter 6’s constraints-are-knowledge callout made concrete: locality zeroes out most of the matrix, equivariance ties the survivors together, and what remains is a linear map with almost nothing left to specify.

PLAN

1. Construct the sparse weight-shared matrix.
-

CODE

```
from matplotlib.colors import ListedColormap

# [1]
weight_ids = torch.zeros(4, 6)
for row in range(4):
    weight_ids[row, row:row + 3] = torch.tensor([1, 2, 3])
```

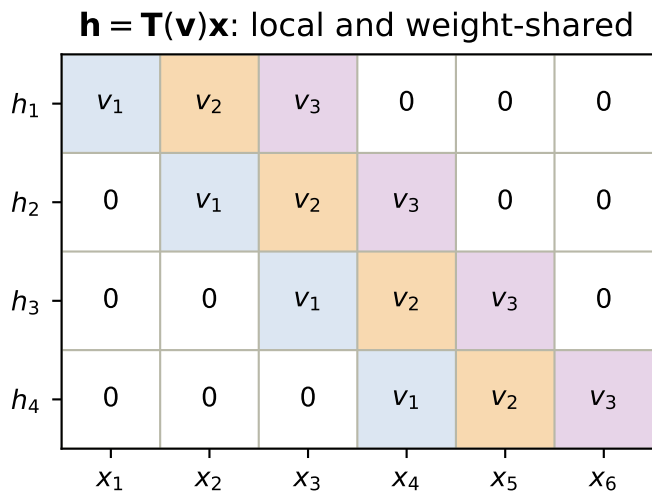


Figure 7.3.: The 1-D version of convolution as a structured matrix (shown in 1-D only so the pattern is visible). Each output row touches one local three-value window, so most entries are zero. The same three kernel weights repeat along diagonals: locality creates sparsity; sharing ties the nonzero entries together. A 2-D image produces the analogous block-structured matrix.

7.8. The bottleneck: someone has to design the kernel

So why did this classical machinery not solve vision decades ago? Because the kernels are only as good as their designers, and hand-crafted kernels have three limitations the operator exposes:

1. **Limited expressiveness.** They detect simple, predefined patterns — gradients, blobs, corners. Nobody can write down nine numbers for “cat ear” or “car wheel.”
2. **Manual feature engineering.** Choosing which kernels to use, and how to combine them, takes deep domain expertise and endless experimentation — for every new task.
3. **No adaptation.** A fixed kernel cannot adjust to a dataset. What works for one task fails for another.

An entire era of computer vision lived inside these constraints, stacking hand-designed detectors into elaborate pipelines. The machine was right; the templates were the bottleneck.

You can hear the question coming. We have a device that is local, weight-shared, and equivariant by construction — with a handful of numbers sitting inside it, currently chosen by hand. Nine numbers. We have spent four chapters building tools that tune numbers against a loss.

What if the template were learnable? That question is [Chapter 8](#), and its answer has a name you already know.

Check yourself

Close the book for one minute and reconstruct the sliding operation.

- Which two design commitments create locality and translation equivariance?
- Why does weight sharing turn a dense linear map into a small kernel?
- What remains hand designed at the end of this chapter?

7.9. Okay, so — the machine before the learning

1. **One operation, many specialists:** slide a small template, take a dot product at every position (Equation 7.1). Blur, sharpen, and edge detection differ only in the kernel’s numbers.
2. **Locality and sharing are built in:** as a matrix, convolution is almost-all-zero with nine numbers repeating — 615,000 weights collapsed to 9.
3. **Equivariance is exact on the interior**, not approximate: filter-then-shift equals shift-then-filter wherever both operations see the same pixels. Boundary handling must be specified separately. The next chapter asks how much local shift tolerance pooling can purchase from that equivariance.
4. **Hand-crafting is the bottleneck** — expressiveness, engineering cost, no adaptation — and it is exactly the kind of bottleneck this book knows how to break.

Sources and further reading

- Fukushima, “[Neocognitron: A Self-organizing Neural Network Model](#)” (1980): an early hierarchy of local, shift-tolerant feature detectors.
- LeCun et al., “[Gradient-Based Learning Applied to Document Recognition](#)” (1998): connects shared local kernels to the trainable CNN built next.
- Dumoulin and Visin, “[A Guide to Convolution Arithmetic for Deep Learning](#)” (2016): a visual reference for padding, stride, and output-shape arithmetic.

Exercises

1. **(Pencil.)** Compute by hand the full output of cross-correlating with the vertical Sobel kernel the 4×4 image whose rows are all $(0, 0, 1, 1)$. Before computing: where should the detector fire?
2. **(Pencil.)** (a) Design a 3×3 kernel that detects *diagonal* edges (bottom-left to top-right). State your design rule: what must the weights sum to, and why? **(Code.)** (b) Run it on `make_shapes()` and check that it fires on the stripes.
3. **(Pencil.)** Prove translation equivariance from Equation 7.1: substitute the shifted image $X'_{i,j} = X_{i-s,j}$ and show $H'_{i,j} = H_{i-s,j}$. Why does the same argument fail for the MLP of Chapter 6? State where the proof applies at a finite image boundary.
4. **(Code.)** True convolution flips the kernel: run `conv2d` with `k` and with `k.flip(0, 1)` for the blur and for Sobel. Which outputs differ, which do not, and what property of the kernel decides it?
5. **(Code.)** The box blur is *separable*: show that convolving with the 3×3 uniform kernel equals convolving with a 3×1 column of $\frac{1}{3}$ s and then a 1×3 row of $\frac{1}{3}$ s. Count multiplications per output pixel for each route — this trick mattered enormously in the hand-crafted era.

8. CNNs: Making the Filters Learnable

Chapter 7 closed on a question it had spent a whole chapter earning: we built a machine that is local, weight-shared, and translation-equivariant away from boundaries, with nine hand-picked numbers sitting inside it. Nine numbers → nine knobs → we have spent half a book tuning knobs against a loss.

So: declare the kernel a **parameter**. That single change is this chapter, and it is the whole revolution. Everything else here — channels, padding, stride, pooling — is the supporting cast that turns one learnable filter into a working network. By the end we will have assembled LeNet, the first great convolutional architecture, trained it on our garments, and re-run the cruelest experiment of [Chapter 6](#) to see what the inductive bias actually bought.

PLAN

1. Load imports and the Fashion-MNIST subset.
-

CODE

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
development = torch.load("../data/fashion-train.pt")
holdout = torch.load("../data/fashion-test.pt")
X_dev = development["X"].float().unsqueeze(1) / 255.0 # (1200, 1, 28, 28)
y_dev, classes = development["y"], development["classes"]
X_test = holdout["X"].float().unsqueeze(1) / 255.0 # (600, 1, 28, 28)
y_test = holdout["y"]

split = torch.randperm(len(X_dev), generator=torch.Generator().manual_seed(6050))
fit_idx, val_idx = split[:1000], split[1000:]
X_tr, y_tr = X_dev[fit_idx], y_dev[fit_idx] # (1000,1,28,28), (1000,)
X_val, y_val = X_dev[val_idx], y_dev[val_idx] # (200,1,28,28), (200,)
```

Note the shapes. In Part I we flattened every image into a 784-vector before the model ever saw it. That stops now: the images stay rectangular, and they pick up a *channel* dimension, for reasons that will occupy us shortly.

 The book's batch convention: NCHW

Throughout this book, every image **batch** uses PyTorch's **NCHW** order: (batch, channels, height, width). A single grayscale batch is therefore shaped (1, 1, 28, 28). `Conv2d` can also accept an unbatched (C, H, W) input, but keeping the batch dimension explicit makes downstream shapes predictable. The kernels of a conv layer live in a 4-D tensor too: (out-channels, in-channels, height, width). This bookkeeping is the single most common source of errors in convolutional code: when a shape error greets you, count dimensions before touching anything else. Chapter 7's `conv2d` helper hid this with `x[None, None]`; from here on we handle it in the open.

8.1. Nine numbers, learned

Here is a game. I have a feature detector — one of Chapter 7's zoo, but I won't tell you which. I will only show you what it does: you hand it images, it hands back feature maps. Your job is to build a detector that matches it.

With the tools of Part I, this is not even hard. A kernel applied by Equation 7.1 is just nine numbers entering a dot product → the output is differentiable with respect to those numbers → gradient descent applies. Start from a random kernel and run the loop we have run since Chapter 1: predict → measure → step.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Recover a mystery kernel by gradient descent.
-

CODE

```
# [1]
mystery = torch.tensor([[-1., 0., 1.],
                        [-2., 0., 2.],
                        [-1., 0., 1.]])      # (don't peek)
img = X_tr[:256]                          # (256, 1, 28, 28)
with torch.no_grad():
    target = F.conv2d(img, mystery[None, None]) # what the detector does

torch.manual_seed(0)
K = 0.1 * torch.randn(1, 1, 3, 3)
K_init = K.clone().squeeze()
K.requires_grad_(True)

lr = 0.5
# [2]
for step in range(601):
    pred = F.conv2d(img, K)                # predict
```

```

loss = F.mse_loss(pred, target)                # measure
loss.backward()
with torch.no_grad():                          # step
    K -= lr * K.grad
    K.grad.zero_()
if step % 200 == 0:
    print(f"step {step:3d}    loss {loss.item():.2e}")

print(f"\nlearned kernel, rounded:\n{K.detach().squeeze().round(decimals=2)}")
print(f"max |learned - mystery| = {(K.detach().squeeze() -
    ↪ mystery).abs().max():.3f}")

```

```

step  0    loss 1.37e+00
step 200   loss 5.53e-04
step 400   loss 4.80e-05
step 600   loss 4.42e-06

```

```

learned kernel, rounded:
tensor([[ -1.0100, -0.0000,  1.0100],
        [-1.9800, -0.0100,  1.9900],
        [-1.0100,  0.0100,  1.0000]])
max |learned - mystery| = 0.018

```

The loss falls more than five orders of magnitude, and the learned kernel is the vertical Sobel detector to within 0.02 in every entry. Gradient descent, given nothing but input–output pairs, has *reinvented* a filter that took computer-vision researchers years of squinting at image gradients to design.

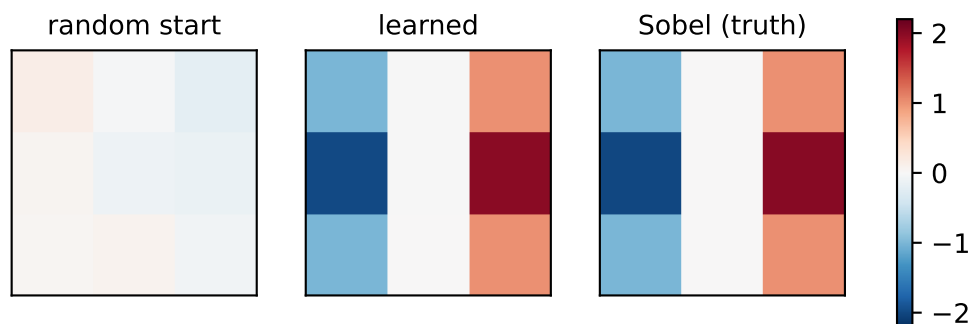


Figure 8.1.: The mystery-detector game. Left: the random starting kernel. Middle: the kernel gradient descent found from input–output pairs alone. Right: the hand-crafted vertical Sobel detector from Chapter 7’s zoo. Nobody told the middle panel what an edge is.

💡 The pivot of Part II

Read that cell again, because the entire deep-learning revolution in vision is that cell writ large. Chapter 7 ended with the bottleneck: hand-crafted kernels are limited to patterns a human can articulate in nine numbers — nobody can write down “cat ear.” The fix is not better articulation. It is to stop articulating: put the numbers on the gradient tape and let the task pull the detector it needs out of the data. *What if the template were learnable?* It is, and the machinery has been in

our hands since Chapter 4.

Two honest disclaimers about the game, both of which make the real thing *more* remarkable, not less. First, the game was rigged: the target came from a known kernel, so a perfect answer existed and the loss surface was a clean convex bowl (the output is linear in V , and squared error in a linear map is Chapter 1's setting all over again). Second, and more important: **in a real CNN, nobody supplies the target feature maps.** The only supervision is the classification loss at the far end of the network. Blame flows backward through every layer — exactly the mechanics of Chapter 5 — and the kernels that emerge are whatever detectors the task itself demands. We chose Sobel here; a trained network might discover it needs something no one has named.

One backpropagation detail deserves a sentence, because it is the same fan-out rule we met in Chapter 5. The *same* nine numbers are used at every position of the slide $\rightarrow$ every position contributes blame to the shared kernel. Those contributions accumulate, then the loss reduction supplies its scale; `F.mse_loss` with the default `reduction="mean"` averages over batch, channel, and spatial elements. Weight sharing is not just a parameter economy. At training time every patch of every image teaches the one shared template.

8.2. From one detector to a layer of detectives

One learnable kernel gives one feature map: one specialist, sweeping the image with one magnifying glass. But Chapter 7's zoo already told us one specialist is not enough — vertical edges, horizontal edges, blobs, textures all carry information. So we hire a *team*. Employ six detectives, each with a different expertise, each producing their own annotated map of the crime scene.

That is all the **channel** machinery is:

- **Out-channels: more detectives.** A conv layer with 6 output channels owns 6 independent kernels and produces a stack of 6 feature maps.
- **In-channels: detectives read the whole stack.** The *next* layer's input is that 6-deep stack, so each of its kernels grows a third dimension: a $6 \times 5 \times 5$ volume that looks at all six incoming maps *simultaneously* and sums the evidence into one output map, one bias per output channel at the end.

For an input stack X with C_{in} channels, output map o is

$$[H_o]_{i,j} = b_o + \sum_{c=1}^{C_{\text{in}}} \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [V_{o,c}]_{a,b} [X_c]_{i+a,j+b}, \quad (8.1)$$

which is Equation 7.1 run once per input channel and summed — followed, as always since Chapter 3, by a nonlinearity. A convolutional layer is Chapter 3's recipe with a structured linear map: dot products $\rightarrow$ add bias $\rightarrow$ ReLU. Nothing in the recipe changed; only the wiring of the linear part did.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Run one grayscale image through two conv layers.
-

CODE

```
# [1]
conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2)
conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)

x = X_tr[:1]                                # (1, 1, 28, 28): one garment
h1 = F.relu(conv1(x))
h2 = F.relu(conv2(h1))
# [2]
print(f"input      {tuple(x.shape)}")
print(f"conv1      {tuple(h1.shape)}   kernels: {tuple(conv1.weight.shape)}")
print(f"conv2      {tuple(h2.shape)}   kernels: {tuple(conv2.weight.shape)}")
```

```
input      (1, 1, 28, 28)
conv1      (1, 6, 28, 28)   kernels: (6, 1, 5, 5)
conv2      (1, 16, 24, 24)  kernels: (16, 6, 5, 5)
```

Look at `conv2.weight`: sixteen kernels, each $6 \times 5 \times 5$. This is where features start *composing*. Suppose detective 3 in the first layer is a vertical-edge expert and detective 5 hunts horizontal edges. A second-layer kernel that weights both maps positively in the same neighborhood fires precisely when both experts agree $\rightarrow$ a **corner detector**, built from evidence no single first-layer kernel could see. The cross-talk between channels is how edges become corners, corners become textures, and textures become parts. We promised this compositional ladder in Chapter 3; here is the hardware it runs on.

Chapter 7's zoo can stage the story before any learning enters. Give the first layer two hand-crafted experts — vertical and horizontal edge energy — then hand-build one second-layer kernel that reads both reports at once:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Run a corner detector built from two edge experts.
-

CODE

```
# [1]
square = torch.zeros(28, 28)
square[7:21, 7:21] = 1.0
sobel_v = torch.tensor([[-1., 0., 1.], [-2., 0., 2.], [-1., 0., 1.]])
v = F.conv2d(square[None, None], sobel_v[None, None], padding=1).abs()
h = F.conv2d(square[None, None], sobel_v.T.contiguous()[None, None], padding=1).abs()

# [2]
experts = torch.cat([v, h], dim=1) # (1, 2, 28, 28): two reports
corner_k = torch.full((1, 2, 5, 5), 1 / 50) # one kernel, spanning both
corners = F.conv2d(experts, corner_k, padding=2).squeeze()
```

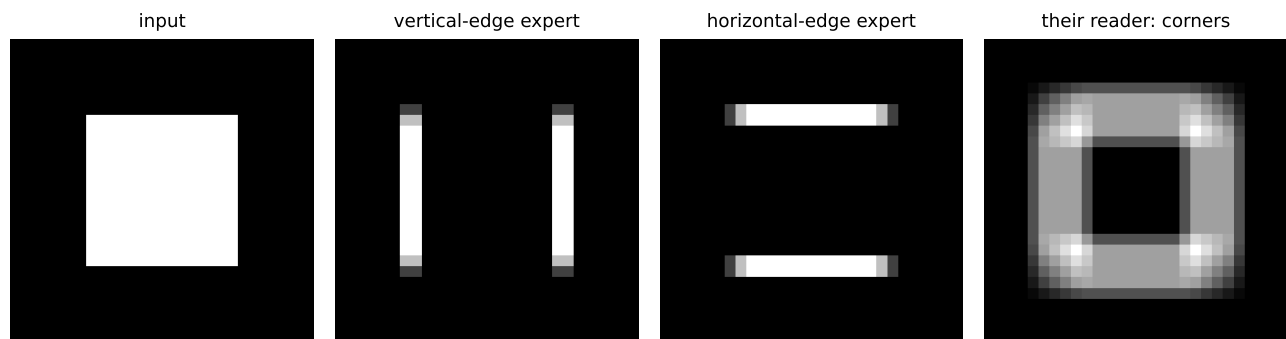


Figure 8.2.: Cross-talk staged by hand. Two first-layer experts report edge energy (absolute Sobel response) on a square. One second-layer kernel spanning both channels (all-positive weights; Equation 8.1 with two input channels) fires hardest exactly where both experts agree — the four corners, a feature neither channel could see alone.

The four brightest cells of the right panel are exactly the square’s four corners: a corner detector, assembled from experts that individually know nothing about corners. In a trained CNN, gradient descent builds combinations like this — and far stranger ones — on its own.

💡 nn versus F, appliances versus recipes

A useful analogy: `nn` modules are *appliances* — they have knobs and memory, and PyTorch carries their state around for you (`nn.Conv2d` owns its kernels and biases as `nn.Parameters`, registered for autograd and visible to the optimizer). `F` functions are *recipes* — stateless, everything passed in by hand. In Chapter 7 the kernel was ours, so `F.conv2d` was the honest choice; now the kernel is the layer’s business, so `nn.Conv2d` is. Use appliances for anything with parameters, recipes for everything else (`F.relu`, `F.max_pool2d`).

8.3. Padding and stride: the bookkeeping with a payoff

Chapter 7 left an administrative debt: a $k \times k$ kernel on an $n \times n$ image yields only $(n - k + 1) \times (n - k + 1)$ outputs, because the window must stay inside the frame. For one layer, a nuisance. For a *deep* stack,

fatal: $28 \rightarrow 24 \rightarrow 20 \rightarrow \dots$ and the feature map shrinks toward nothing, with edge pixels contributing to ever fewer windows along the way.

The fix is blunt: **padding**. Add a border of zeros around the input so the window can center itself on every real pixel. For a 5×5 kernel, a 2-pixel border returns exactly the input size $\rightarrow$ that is why `conv1` above says `padding=2`. Padding decouples “how deep can the network be” from “how fast do the maps shrink,” and that decoupling is what makes deep stacks possible at all.

Sometimes, though, we *want* to shrink — coarser maps mean less computation and a wider view per pixel. Then we shrink on purpose with **stride**: slide the window s pixels at a time instead of one. Both knobs live in one formula, worth memorizing because you will run it in your head for every architecture you ever read:

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1. \quad (8.2)$$

The numerator is the padded length minus the window, i.e. how far the window can travel; dividing by the step and flooring counts the stops; $+1$ counts the starting position. Three worked cases make the arithmetic visible:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Implement the output-size formula, three regimes.
-

CODE

```
# [1]
x8 = torch.randn(1, 1, 8, 8)
# [2]
for p, s in [(0, 1), (1, 1), (1, 2)]:
    out = F.conv2d(x8, torch.randn(1, 1, 3, 3), padding=p, stride=s)
    n_out = (8 + 2 * p - 3) // s + 1
    print(f"n=8, k=3, p={p}, s={s}: formula {n_out} torch {tuple(out.shape[2:])}")
```

```
n=8, k=3, p=0, s=1: formula 6 torch (6, 6)
n=8, k=3, p=1, s=1: formula 8 torch (8, 8)
n=8, k=3, p=1, s=2: formula 4 torch (4, 4)
```

8.4. Pooling: trading exact location for local tolerance

Now for the deepest design decision in the network, and it starts from the asset Chapter 7 banked. Under matched boundary rules, convolution is equivariant on the interior: shift the garment, and every feature map shifts in lockstep. The detective’s map of clues faithfully tracks the scene. But the network’s final job is not mapping clues; it is a verdict. *Is this a trouser?* The answer must not depend on where the trouser stands. The feature-finding stage wants **equivariance** (where things are matters); the decision

stage wants **invariance** (only what was found matters). Equivariance is what we have; invariance is what the classifier head needs → something must reduce sensitivity to location.

That converter is **pooling**. Slide a window over each feature map (no parameters this time) and summarize each neighborhood by one number:

- **Max pooling** asks: *what is the strongest clue here?* Keep the loudest activation, discard the rest.
- **Average pooling** asks: *what is the general vibe of this neighborhood?* Smoother, but a single sharp clue gets diluted by quiet neighbors.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Max and average pooling on a known grid.

CODE

```
# [1]
t = torch.arange(16.).reshape(1, 1, 4, 4)
# [2]
print(f"input:\n{t.squeeze()}")
print(f"max pool 2x2:\n{F.max_pool2d(t, 2).squeeze()}")
print(f"avg pool 2x2:\n{F.avg_pool2d(t, 2).squeeze()}")
```

```
input:
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
max pool 2x2:
tensor([[ 5.,  7.],
        [13., 15.]])
avg pool 2x2:
tensor([[ 2.5000,  4.5000],
        [10.5000, 12.5000]])
```

The common configuration is the one above: 2×2 windows with stride 2, non-overlapping, halving each spatial dimension. Max is the default choice in practice, precisely because feature detection is about the strongest evidence, not the committee average.

How can this buy local tolerance? Start with a deliberately clean construction in which a shift becomes invisible:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Run one constructed shift that pooling cannot see.

CODE

```
# [1]
scene = torch.zeros(1, 1, 4, 4)
scene[0, 0, 1, 0] = 9.0      # a strong clue, upper-left region
scene[0, 0, 2, 2] = 3.0      # a weaker clue, lower-right region

shifted = torch.zeros_like(scene)
shifted[0, 0, 1, 1] = 9.0    # both clues moved one pixel right
shifted[0, 0, 2, 3] = 3.0

# [2]
print(f"pooled original: {F.max_pool2d(scene, 2).squeeze().tolist()}")
print(f"pooled shifted: {F.max_pool2d(shifted, 2).squeeze().tolist()}")
```

```
pooled original: [[9.0, 0.0], [0.0, 3.0]]
pooled shifted:  [[9.0, 0.0], [0.0, 3.0]]
```

Both clues moved, yet these particular pooled maps are *identical*: each isolated maximum stayed inside its 2×2 window. That equality belongs to this construction, not to max pooling in general. A shift can change which values share a window, which maximum wins, or what crosses the boundary. Pooling can therefore buy local tolerance to jitter, but the amount depends on the activations and alignment; the end of this chapter measures it rather than assuming it.

 Tolerance has a price tag

Pooling can buy tolerance by *throwing away position*. After two rounds, the network knows a strong vertical edge exists in a region; it no longer knows exactly where. For classification this is the right trade — but it is a trade, and it will come due twice in this book. Deeper in Part II, tasks that need precise locations will have to be more careful with resolution. And in Part IV the tables turn completely: self-attention ([Chapter 14](#)) is so thoroughly position-agnostic that we will have to *pay to put position back in*. Remember, when we get there, that position was something we once discarded on purpose.

8.5. How far a deep neuron sees

There is one more consequence of stacking, and it quietly resolves the tension Chapter 6 left us with. Every kernel here is tiny — 5×5 , resolutely local. Yet recognizing a garment is a *global* judgment. Where does globality come from?

From depth. A neuron in the first feature map sees a 5×5 patch of the image: its **receptive field**. A neuron one layer up sees a 5×5 patch of *feature maps*, each pixel of which already summarizes a patch below $\rightarrow$ its receptive field on the original image is wider. Pooling accelerates this: after a 2×2 /stride-2 pool, neighboring pixels in the pooled map stand two original pixels apart, so every later window covers twice the ground. For our upcoming network the ledger reads: conv1 sees $5 \times 5 \rightarrow$ pooling nudges it to $6 \times 6 \rightarrow$ conv2's 5×5 window, at stride-2 spacing, expands it to $14 \times 14 \rightarrow$ the final pool reaches 16×16 — over half the frame in every direction, from nothing but 5×5 looks.

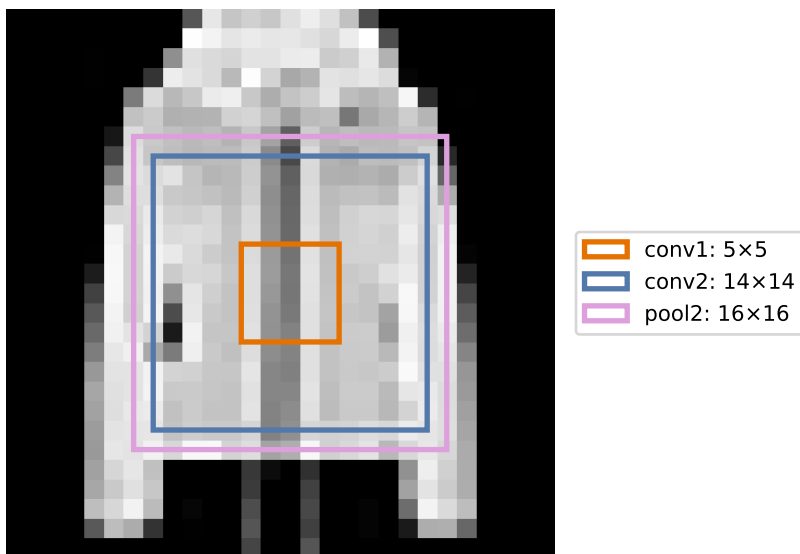


Figure 8.3.: What one neuron sees, by depth. A coat replaces the earlier sandal so the nested fields cross visible sleeves, torso, and boundary. Conv1 sees a 5×5 patch, conv2 sees 14×14 , and the final pooled cell sees 16×16 . Local wiring earns increasingly global sight; the dense head then reads all 25 overlapping final views at once.

This is the compositional hierarchy of **Chapter 3** given a strong architectural invitation. The MLP *could* represent features-of-features but nothing encouraged it to; locality and growing receptive fields encourage later layers to compose earlier responses. Learned channels often progress from edge-like patterns to textures and parts, but the architecture does not force those human labels. The CNN buys global sight gradually, paying for it with depth; hold that thought until Part IV, where attention will buy global sight in a single step and pay for it differently (**Chapter 13**).

8.6. LeNet: the whole machine

Time to assemble. The canonical blueprint is **LeNet**, Yann LeCun's digit-reading network, and its rhythm is the CNN design pattern to this day: *convolve, shrink, deepen; repeat; then decide*. Spatial size falls ($28 \rightarrow 14 \rightarrow 5$) while feature depth grows ($1 \rightarrow 6 \rightarrow 16$): the network trades “where” for “what” stage by stage, until a small dense head reads the final $16 \times 5 \times 5$ summary and votes.

PLAN

1. LeNet.
-

CODE

```
# [1]
class LeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, padding=2) # (N,1,28,28)->(N,6,28,28)
        self.conv2 = nn.Conv2d(6, 16, 5) # (N,6,14,14)->(N,16,10,10)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.max_pool2d(F.relu(self.conv1(x)), 2) # -> (N, 6, 14, 14)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2) # -> (N, 16, 5, 5)
        x = x.flatten(1) # -> (N, 400)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x) # logits, (N, 10)
```

Every line is a tool we already own: Equation 8.1 twice, Equation 8.2 silently fixing `padding=2`, pooling twice, then Chapter 3’s MLP as the decision head — and, per Chapter 2’s advice, the model returns raw logits and lets `F.cross_entropy` handle the probabilities.

i LeNet, then and now

The LeNet family (LeCun and colleagues, 1989–1998) began by reading handwritten ZIP codes for the US Postal Service, and its mature form, LeNet-5, went on to read a substantial share of America’s bank checks — convolutional networks were literally cashing checks decades before they were fashionable. Our variant keeps the classic skeleton but upgrades the internals with Part I’s verdicts: ReLU where the original used sigmoid-family activations (Chapter 3, Chapter 5), max pooling where it used average-flavored subsampling, Adam as the optimizer. A later implementation adds one more modern stabilizer, batch normalization, which we meet properly in Chapter 9.

Before training, the parameter audit — because parameter *economy* was half of Chapter 7’s sales pitch, and Chapter 6’s MLP is the yardstick:

PLAN

1. Define the reusable `make_mlp` helper.
2. Prepare the inputs and fixed settings for the example.
3. Parameter audit, LeNet vs. Chapter 6's MLP.

CODE

```
# [1]
def make_mlp() -> nn.Sequential:
    return nn.Sequential(nn.Flatten(),
                          nn.Linear(784, 256), nn.ReLU(), nn.Linear(256, 10))

# [2]
lenet, mlp = LeNet(), make_mlp()
# [3]
for name, p in lenet.named_parameters():
    print(f"{name:14s} {str(tuple(p.shape)):16s} {p.numel():>7,}")
n_conv = sum(p.numel() for n, p in lenet.named_parameters() if "conv" in n)
n_lenet = sum(p.numel() for p in lenet.parameters())
n_mlp = sum(p.numel() for p in mlp.parameters())
print(f"\nLeNet: {n_lenet:,} parameters ({n_conv:,} of them convolutional)")
print(f"MLP:    {n_mlp:,} parameters")
```

conv1.weight	(6, 1, 5, 5)	150
conv1.bias	(6,)	6
conv2.weight	(16, 6, 5, 5)	2,400
conv2.bias	(16,)	16
fc1.weight	(120, 400)	48,000
fc1.bias	(120,)	120
fc2.weight	(84, 120)	10,080
fc2.bias	(84,)	84
fc3.weight	(10, 84)	840
fc3.bias	(10,)	10

LeNet: 61,706 parameters (2,572 of them convolutional)

MLP: 203,530 parameters

Two readings of this table. Against the MLP: LeNet carries 61,706 parameters to the MLP's 203,530 (less than a third) while performing far richer spatial computation. Weight sharing is doing exactly what Chapter 7's matrix view promised. Within LeNet: look where the parameters live. The two conv layers, the part that actually *sees*, own 2,572 parameters (about 4%) while the dense head hoards the rest. That imbalance is a design smell we will fix in [Chapter 9](#), where modern architectures go deeper in convolution and lighter in the head.

Now train both models under one shared optimization recipe: same fitting split, optimizer, minibatch size, number of epochs, and therefore number of parameter updates. The comparison does **not** match

8. CNNs: *Making the Filters Learnable*

parameter count or floating-point work; a convolution and a dense layer spend different computation per update. Architecture is the intended difference, compute is not controlled:

PLAN

1. Define the reusable helpers: `train_model` and `accuracy`.
2. Run one recipe, two architectures.
3. Report or visualize the measured result.

CODE

```

from dlbook.supervised import fit_supervised    # Listing 4.1

# [1]
def train_model(
    model_fn, X_fit: torch.Tensor = X_tr, y_fit: torch.Tensor = y_tr,
    epochs: int = 150, batch: int = 128, seed: int = 6050
) -> nn.Module:
    # Listing 4.1 verbatim; this chapter's deltas are the budget (150
    # epochs, batch 128) and, below, the models that get trained.
    return fit_supervised(
        model_fn, X_fit, y_fit, epochs=epochs, batch=batch, seed=seed
    )

@torch.no_grad()
def accuracy(model: nn.Module, X: torch.Tensor, y: torch.Tensor) -> float:
    model.eval()
    return (model(X).argmax(1) == y).float().mean().item()

# [2]
mlp = train_model(make_mlp)
lenet = train_model(LeNet)
# [3]
for name, model in [("MLP ", mlp), ("LeNet", lenet)]:
    print(f"{name}  train {accuracy(model, X_tr, y_tr):.1%}    "
          f"validation {accuracy(model, X_val, y_val):.1%}")

```

```

MLP    train 100.0%   validation 75.5%
LeNet  train 99.5%   validation 74.5%

```

Read the clean-validation column honestly: the MLP scores 75.5% and LeNet 74.5% in this single seeded run. A one-point difference is descriptive, not an uncertainty estimate, so it does not establish which architecture has higher expected clean accuracy. Chapter 6 already told us why a landslide is unlikely here: centered garments do not expose the MLP's main weakness. Both models nearly interpolate the fitting split. Our small-data regime makes a narrower point: LeNet reaches comparable clean accuracy with less than a third of the parameters. The shift experiment now asks whether it learned a different kind of representation.

8.7. The rematch

Chapter 6 convicted our MLP with one experiment: shift every validation garment two pixels right, and accuracy fell off a cliff — the full-frame templates kept scoring sleeves against background. We then promised that an architecture built on locality and weight sharing would face the same trial. Write your predictions down before running — two numbers: LeNet’s clean accuracy, and LeNet at a two-pixel shift. The MLP managed 76% and 43%.

PLAN

1. Define the reusable `shift_right` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Plot accuracy vs. shift, both models.
-

CODE

```
# [1]
def shift_right(X: torch.Tensor, px: int) -> torch.Tensor:
    out = torch.zeros_like(X)
    if px == 0:
        return X.clone()
    out[..., px:] = X[..., :-px]
    return out

# [2]
shifts = list(range(5))
acc_mlp = [accuracy(mlp, shift_right(X_val, s), y_val) for s in shifts]
acc_cnn = [accuracy(lenet, shift_right(X_val, s), y_val) for s in shifts]
# [3]
for s in shifts:
    print(f"shift {s}px:    MLP {acc_mlp[s]:.1%}    LeNet {acc_cnn[s]:.1%}")

plt.figure(figsize=(5.5, 3.2))
plt.plot(shifts, acc_mlp, "o-", color="#E57200", lw=2, ms=7, label="MLP (Ch. 6)")
plt.plot(shifts, acc_cnn, "s-", color="#232D4B", lw=2, ms=7, label="LeNet")
plt.axhline(0.1, ls=":", color="#B8B8A8")
plt.xlabel("shift (pixels right)"); plt.ylabel("validation accuracy")
plt.ylim(0, 0.9); plt.xticks(shifts); plt.legend()
plt.tight_layout(); plt.show()
```

shift 0px:	MLP 75.5%	LeNet 74.5%
shift 1px:	MLP 64.0%	LeNet 66.0%
shift 2px:	MLP 42.0%	LeNet 57.5%
shift 3px:	MLP 27.0%	LeNet 42.0%
shift 4px:	MLP 18.5%	LeNet 23.0%

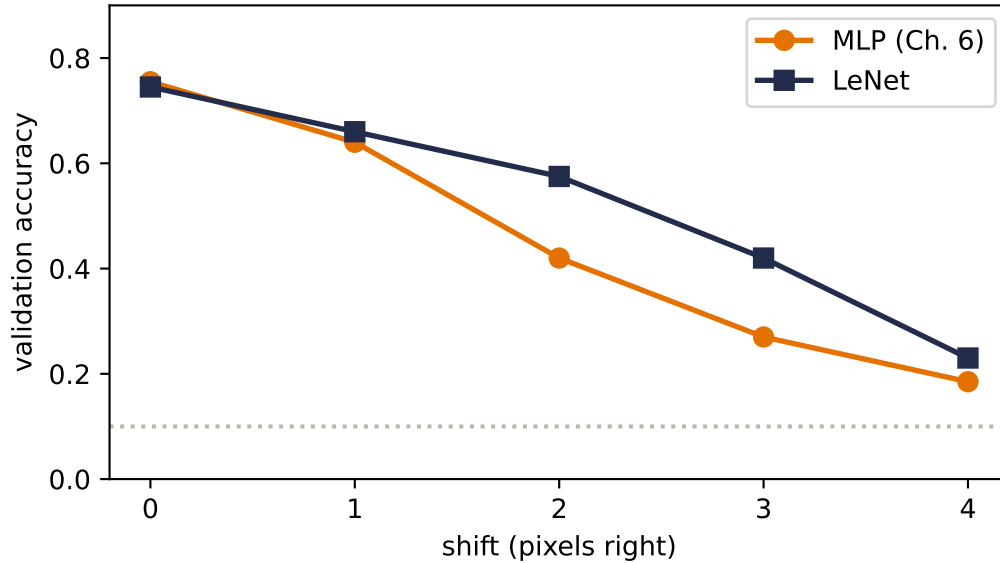


Figure 8.4.: The validation rematch. Both architectures use the same optimizer schedule, though not matched compute. At two pixels the MLP falls from 75.5% to 42.0%, while LeNet falls from 74.5% to 57.5%. The dotted line is chance.

There is the payoff, and it is worth stating precisely. At two pixels the MLP falls from 75.5% to 42.0%, while LeNet falls from 74.5% to 57.5%. That is one deterministic validation split, not a sampling distribution, but the within-run contrast is large. The mechanism is everything this chapter built: the kernels travel with the garment (equivariance), so the *features are still found*; pooling forgives some sub-window jitter (local insensitivity); only the coarse layout entering the head changes at all.

And yet the navy curve falls too. Also worth stating precisely, because it teaches the architecture’s fine print. Two rounds of $2\times$ pooling buy tolerance measured in a few pixels, not unlimited; and after `flatten(1)`, the dense head reads the final 5×5 grid *positionally* — a four-pixel input shift moves features roughly one full cell in that grid, and the head is as brittle to cell-level shifts as Chapter 6’s MLP was to pixel-level ones. The inductive bias did not abolish the cliff; it moved it outward and flattened it into a slope. Reducing the remaining sensitivity is the next chapter’s business, where we revisit depth and the position-sensitive head.

One last look inside, because Chapter 6 also showed us what the MLP’s first layer *looked like* — global, smeared, garment-shaped templates ([Chapter 6](#)) — and promised that structure would change:

PLAN

1. Extract first-layer kernels and their feature maps.
-

CODE

```
# [1]
example_index = 3                                # the Figure 8.3 coat
img = X_val[example_index:example_index + 1]
with torch.no_grad():
    fmaps = F.relu(lenet.conv1(img)).squeeze(0)    # (6, 28, 28)
    kernels = lenet.conv1.weight.detach().squeeze(1) # (6, 5, 5)
```

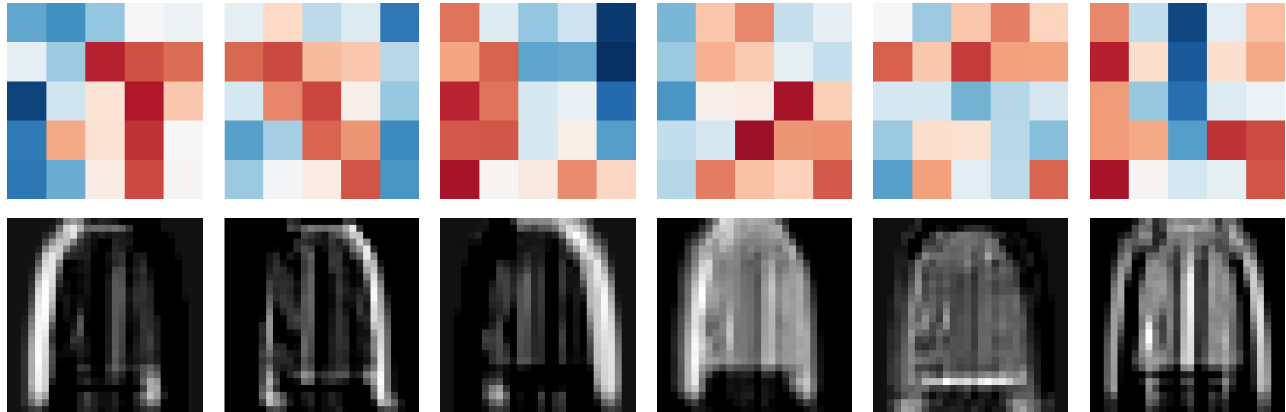


Figure 8.5.: What LeNet learned to look for. Top: six first-layer 5×5 kernels (red positive, blue negative). Bottom: their feature maps on the same validation coat used in Figure 8.3, whose sleeves and torso make spatial responses easier to compare than the earlier sandal. The maps respond differently, but several kernels remain diffuse and should not be over-interpreted.

Six small, local kernels produce distinct response maps on the same garment. Some show directional structure; others remain noisy enough that naming a detector would tell more story than this one run supports. Nobody designed them. The classification loss, pulling blame backward through Equation 8.1 for roughly 1,200 optimizer steps, chose these local measurements.

8.8. One sealed test check

We have now fixed both architectures, the optimizer schedule, the shift size, and the two reported metrics. We refit each model on all 1,200 development images, then open the 600-image test set once:

Here *sealed* is chapter-local. The experiment interlude has already disclosed that this 600-image file is a reused book endpoint, not a globally untouched test set; it remained decision-inert while this chapter's choices were made.

PLAN

1. Run one final test-set check.
2. Report or visualize the measured result.

CODE

```
# [1]
mlp_final = train_model(make_mlp, X_dev, y_dev)
lenet_final = train_model(LeNet, X_dev, y_dev)
# [2]
for name, model in [("MLP ", mlp_final), ("LeNet", lenet_final)]:
    print(f"{name}  clean {accuracy(model, X_test, y_test):.1%}    "
          f"shift-2 {accuracy(model, shift_right(X_test, 2), y_test):.1%}")
```

```
MLP      clean 82.2%    shift-2 41.8%
LeNet    clean 82.5%    shift-2 62.3%
```

The sealed check confirms the validation diagnosis. Clean accuracy is 82.2% for the MLP and 82.5% for LeNet, a descriptive 0.3-point gap from one seed. Under the predeclared two-pixel shift, the gap becomes 41.8% versus 62.3%.

i Check yourself

Close the book for one minute and rebuild the CNN.

- Which kernel entries become learnable, and which convolutional structure remains built in?
- What do channels, pooling, and receptive-field growth each contribute?
- Why does the shift rematch support a structural claim rather than a universal architecture ranking?

8.9. Okay, so — the kernel became learnable

1. **One change made the revolution:** the kernel's numbers became parameters. Gradient descent can recover a hand-crafted detector from data (our rigged game) and, more importantly, discover detectors nobody designed, supervised only by the task loss at the far end.
2. **Channels turn one detector into a team:** out-channels stack specialists' feature maps; in-channels let the next layer read all maps at once (Equation 8.1), which is how edges compose into corners and parts.
3. **Padding and stride are the shape budget:** $n_{\text{out}} = \lfloor (n + 2p - k)/s \rfloor + 1$ (Equation 8.2). Padding lets networks go deep without shrinking to nothing; stride shrinks on purpose.
4. **Pooling can turn equivariance into local shift tolerance** — the strongest clue per window, some position discarded. It is a purchase, and position is the currency.
5. **Receptive fields grow with depth:** $5 \rightarrow 6 \rightarrow 14 \rightarrow 16$ in our LeNet. Local wiring earns global sight gradually; the hierarchy Chapter 3 could only hope for is now encouraged by architecture.

6. **The verdict:** on the sealed test set, one seed puts clean accuracy at 82.2% for the MLP and 82.5% for LeNet; this does not establish an expected clean-accuracy difference. Under the predeclared two-pixel shift, the same models score 41.8% and 62.3%. The bias did not add capacity; it added the *right constraints*, exactly Chapter 6’s prescription.

Sources and further reading

- LeCun et al., “[Gradient-Based Learning Applied to Document Recognition](#)” (1998): the primary LeNet account, including the architecture’s document-recognition setting.
- Fukushima, “[Neocognitron: A Self-organizing Neural Network Model](#)” (1980): historical context for alternating feature extraction and spatial tolerance.
- Zeiler and Fergus, “[Visualizing and Understanding Convolutional Networks](#)” (2014): methods and evidence for inspecting learned convolutional features without over-reading a single filter image.
- Dumoulin and Visin, “[A Guide to Convolution Arithmetic for Deep Learning](#)” (2016): a compact companion for checking convolution, pooling, and receptive-field shapes.

Exercises

1. **(Pencil.)** Run Equation 8.2 through LeNet layer by layer, starting from 28×28 : verify every shape comment in the LeNet class, then verify the parameter counts of `conv1` and `conv2` from the table (weights *and* biases).
2. **(Pencil.)** Verify the receptive-field ledger $5 \rightarrow 6 \rightarrow 14 \rightarrow 16$. (Track two numbers per stage: the field size, and the *jump* — the spacing in input pixels between neighboring units — which each stride-2 pool doubles.) Then show that no convolutional or pooling unit in LeNet ever sees the full 28×28 frame. Where does globality finally enter the network?
3. **(Code.)** Play the mystery-detector game with the 3×3 blur kernel as the target. Does gradient descent recover it as cleanly as Sobel? Now rerun with only 8 training images instead of 256. What happens to the recovered kernel, and what does that say about data volume versus parameter identifiability?
4. **(Code.)** Swap LeNet’s max pooling for average pooling and retrain. Compare clean accuracy and the shift curve. Which changes more, and does the result match the “strongest clue vs. general vibe” story?
5. **(Pencil.)** The final dense head receives 25 spatial positions per feature map. Explain why this can reintroduce shift sensitivity after equivariant convolutions. Describe what information a position-insensitive head would keep and discard; we will build one in [Chapter 9](#).

9. Modern CNNs and Transfer Learning

Chapter 8 ended with a working machine and a report card. LeNet reads garments at 82.5% with 61,706 parameters; graceful under shift where the MLP cliff-dived. But the card has two demerits we wrote down explicitly: 96% of those parameters sit in the dense head rather than in the convolutions that do the seeing, and the shift cliff was softened, not abolished. And one IOU: batch normalization, promised for this chapter.

What happened after LeNet is a decade of architecture research, and it can drown you in named networks. Four design questions will keep the mechanism visible; we let them drive:

1. **Why 5×5 ?** Could smaller filters do the same job with fewer parameters?
2. **How do we go deeper** without gradients dying on the way down?
3. **Where do the parameters actually live**, and how do we evict them?
4. **Can we design reusable blocks:** optimized layer-combinations we stack, keeping a bird's-eye view instead of re-designing what already works?

Each question has a famous answer (VGG, ResNet, NiN's 1×1 + global pooling, and the block habit itself), and each answer is a constraint-shaped idea we can test on our own data. At the end, the practical superpower this sequence builds to, reusing a pretrained backbone, gets the most honest experiment in this book.

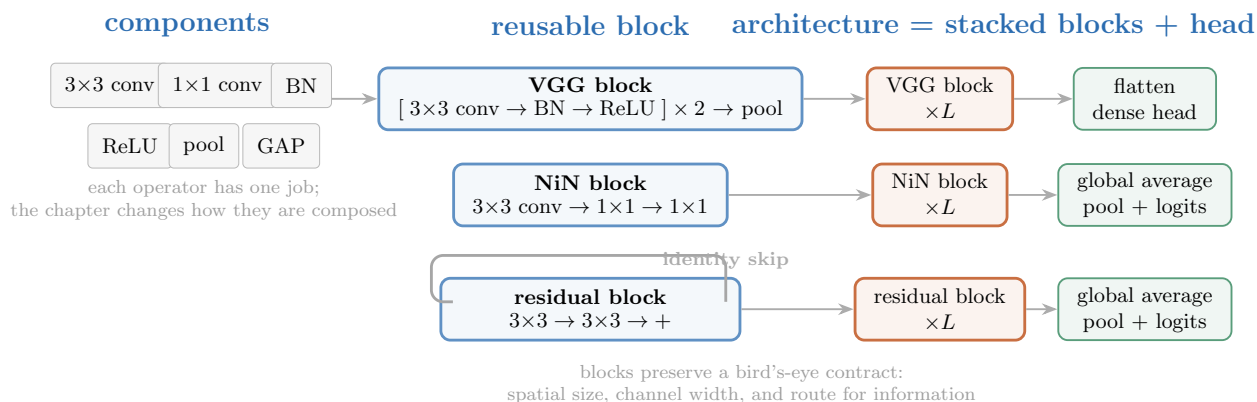


Figure 9.1.: The chapter's construction grammar. Individual operators become reusable blocks; architectures repeat those blocks and attach a head. VGG repeats small-kernel atoms, NiN adds per-pixel channel mixing and global averaging, and ResNet adds an identity route around each learned correction.

Read the figure from left to right whenever a named architecture becomes noisy. First identify the operators, then the block contract, then the repetition schedule and head. The rest of the chapter changes one of those three levels at a time.

9. Modern CNNs and Transfer Learning

Both trainers below are Listing 4.1 with one factoring: the inner loop (`train_existing`) accepts a model and optimizer that already exist, because this chapter's whole subject is training models it did not just construct — fresh heads on frozen trunks, and selectively thawed layers at their own learning rates.

PLAN

1. Load imports, data, and the shared training recipe.
2. Define the reusable helpers: `train_existing` and `train_model`.
3. Define the reusable helpers: `accuracy` and `count`.

CODE

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from collections.abc import Callable

# [1]
torch.manual_seed(6050)
train = torch.load("../data/fashion-train.pt")
test = torch.load("../data/fashion-test.pt")
X_tr = train["X"].float().unsqueeze(1) / 255.0    # (1200, 1, 28, 28)
X_te = test["X"].float().unsqueeze(1) / 255.0    # (600, 1, 28, 28)
y_tr, y_te, classes = train["y"], test["y"], train["classes"]

# [2]
def train_existing(
    model: nn.Module, opt: torch.optim.Optimizer, epochs: int,
    batch: int = 128,
    data: tuple[torch.Tensor, torch.Tensor] | None = None,
) -> nn.Module:
    X, y = data if data is not None else (X_tr, y_tr)
    n = len(X)
    for _ in range(epochs):
        perm = torch.randperm(n)
        for i in range(0, n, batch):
            idx = perm[i:i + batch]
            model.train()
            loss = F.cross_entropy(model(X[idx]), y[idx])
            opt.zero_grad(); loss.backward(); opt.step()
    model.eval()
    return model

def train_model(model_fn: Callable[[], nn.Module], epochs: int,
                batch: int = 128, seed: int = 6050, lr: float = 1e-3,
                data: tuple[torch.Tensor, torch.Tensor] | None = None) -> nn.Module:
    torch.manual_seed(seed)
    model = model_fn()
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    return train_existing(model, opt, epochs, batch, data)
```

```
# [3]
@torch.no_grad()
def accuracy(model: nn.Module, X: torch.Tensor, y: torch.Tensor) -> float:
    model.eval()
    return (model(X).argmax(1) == y).float().mean().item()

def count(model: nn.Module) -> int:
    return sum(p.numel() for p in model.parameters())
```

One evaluation label needs an honesty note. Chapter 8 already opened the 600-image holdout, and this chapter queries it repeatedly while comparing architectures. We keep the familiar `X_te` name, but treat it as the book’s fixed **benchmark**. These comparisons are descriptive, not an unbiased estimate after model selection; a final claim would require a fresh, untouched test set.

Two small novelties in the recipe, both because this chapter’s networks contain batch normalization: `model.train()` before each step and `model.eval()` before each evaluation. Why that matters is the subject of the second section.

9.1. Question 1: why 5×5 ? Stack small kernels instead

VGG’s answer (Simonyan & Zisserman, 2014) is the cleanest idea in this chapter. Recall the receptive-field ledger of [Chapter 8](#): stacking grows the field. Two stacked 3×3 convolutions let the second layer see 5×5 of the input, the *same* receptive field as one 5×5 kernel. But compare the bills, for a layer with C channels in and C out:

$$\underbrace{25 C^2}_{\text{one } 5 \times 5} \quad \text{versus} \quad \underbrace{2 \times 9 C^2 = 18 C^2}_{\text{two } 3 \times 3\text{s}},$$

a 28% saving. The stacked pair also fires a ReLU *twice* where the big kernel fires once. Same sight, fewer parameters, more nonlinearity. Three 3×3 s reach a 7×7 field for $27C^2$ against $49C^2$: the deeper you take the idea, the better the deal gets.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Implement the parameter bill, $C = 32$.
-

CODE

```
# [1]
C = 32
one_5x5 = nn.Conv2d(C, C, 5, padding=2)
two_3x3 = nn.Sequential(nn.Conv2d(C, C, 3, padding=1), nn.ReLU(),
                        nn.Conv2d(C, C, 3, padding=1))

# [2]
print(f"one 5x5: {count(one_5x5):,} parameters, 1 ReLU after")
print(f"two 3x3s: {count(two_3x3):,} parameters, 2 ReLUs")
```

```
one 5x5: 25,632 parameters, 1 ReLU after
two 3x3s: 18,496 parameters, 2 ReLUs
```

One honest caveat before you re-derive the field equations of vision from this: the $18C^2 < 25C^2$ arithmetic assumes the channel width stays C through the stack. When a layer *grows* channels (as LeNet's $6 \rightarrow 16$ did), splitting it into two growing 3×3 s can cost more, not less. VGG's design sidesteps this by keeping width constant inside each block and changing it only between blocks, which is exactly the shape our code will take. (Exercise 1 makes you find the break-even point.)

9.2. The stabilizer we owe you: batch normalization

Before we stack deeper than ever, we need the tool Chapter 8 used and we deferred. Here is the problem it solves. Watch what happens to the *scale* of activations as a signal crosses twelve freshly initialized 3×3 conv layers:

 PLAN

1. Define the reusable `stack12` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Activation scale through 12 conv layers, with and without BN.
 4. Report or visualize the measured result.
-

CODE

```
# [1]
def stack12(with_bn: bool, ch: int = 16) -> nn.Sequential:
    layers = [nn.Conv2d(1, ch, 3, padding=1)]
    for _ in range(11):
        if with_bn:
            layers.append(nn.BatchNorm2d(ch))
        layers += [nn.ReLU(), nn.Conv2d(ch, ch, 3, padding=1)]
    return nn.Sequential(*layers)

# [2]
torch.manual_seed(0)
```

```

x = X_tr[:256]
plain_stack = stack12(False)
bn_stack = stack12(True)
plain_convs = [m for m in plain_stack if isinstance(m, nn.Conv2d)]
bn_convs = [m for m in bn_stack if isinstance(m, nn.Conv2d)]
# [3]
for source, target in zip(plain_convs, bn_convs):
    target.load_state_dict(source.state_dict())           # identical convolution weights

# [4]
for tag, network in [("without BN", plain_stack), ("with BN", bn_stack)]:
    h, stds = x, []
    with torch.no_grad():
        for layer in network:
            h = layer(h)
            if isinstance(layer, nn.Conv2d):
                stds.append(h.std().item())
    print(f"{tag} layer stds: " + " ".join(f"{s:.3f}" for s in stds[:2]))

```

```

without BN layer stds: 0.344  0.059  0.043  0.046  0.049  0.061
with BN      layer stds: 0.344  0.404  0.395  0.385  0.432  0.384

```

Without normalization the signal's spread collapses by an order of magnitude within a few layers and keeps sagging. Forward activation scales and backward gradients are not the same quantity, but both are shaped by the stack's weights and nonlinearities; poorly scaled activations are therefore a warning to inspect gradient flow directly, as we do below.

Batch normalization (Ioffe & Szegedy, 2015) re-standardizes at every layer. For each channel of an NCHW tensor, `BatchNorm2d` computes the mean and variance across the current minibatch **and both spatial axes** (N , H , and W), normalizes the activations, then lets the layer undo that transformation through two learnable knobs:

$$\hat{x} = \frac{x - \mu_{\text{batch}}}{\sqrt{\sigma_{\text{batch}}^2 + \epsilon}}, \quad y = \gamma \hat{x} + \beta. \quad (9.1)$$

The γ, β pair matters: normalization is a *reset to a healthy scale*, not a straitjacket. The network can learn to restore any mean and spread that helps, but it starts every layer from sane numbers. In the printout above, the BN column holds near a constant spread through all twelve layers: every layer trains from day one. The standard placement we adopt is $\text{conv} \rightarrow \text{BN} \rightarrow \text{ReLU}$, with `bias=False` on the convolution, since β already provides the shift.

 **BN is two different machines:** tell PyTorch which one you're running

At training time BN normalizes by the *current batch's* statistics. At evaluation time there may be no batch (one image!), so it uses running averages collected during training. `model.train()` and `model.eval()` switch between the two. Forgetting the switch is the classic BN bug: evaluate in train mode and your predictions depend on whatever else happens to be in the batch; train in eval mode and BN never learns its statistics. Our `train_model` recipe flips the switch in both directions; look for it. Corollary: BN needs real batches to estimate statistics, so it gets unreliable

at tiny batch sizes.

i Normalize across *what*? A seed for Part IV

Batch statistics are one choice among several. When we build transformers (Chapter 14), batches of variable-length sequences will make per-batch statistics awkward, and the same standardize-then-rescale idea will reappear computed per *token* instead: **layer normalization**. Same equation, different axis. Remember Equation 9.1 when you meet it.

9.3. Building with blocks: a small VGG

Now assemble Question 1’s answer with Question 4’s habit. Define the *atom* — conv, BN, ReLU — once; a block stacks atoms and pools; a network stacks blocks. This is the point about VGG versus its hand-tuned predecessors: you stop re-designing and start repeating.

PLAN

1. Define the reusable helpers: `atom` and `VGGSmall`.
2. `VGGSmall` — two blocks of stacked 3×3 atoms.
3. Report or visualize the measured result.

CODE

```
# [1]
def atom(c_in: int, c_out: int) -> list[nn.Module]:
    return [nn.Conv2d(c_in, c_out, 3, padding=1, bias=False),
            nn.BatchNorm2d(c_out), nn.ReLU()]

class VGGSmall(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            *atom(1, 16), *atom(16, 16), nn.MaxPool2d(2),    # -> (16, 14, 14)
            *atom(16, 32), *atom(32, 32), nn.MaxPool2d(2), # -> (32, 7, 7)
        )
        self.head = nn.Sequential(nn.Flatten(),
                                   nn.Linear(32 * 7 * 7, 128), nn.ReLU(),
                                   nn.Linear(128, 10))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.features(x))

# [2]
vgg = train_model(VGGSmall, epochs=60)
n_head = count(vgg.head)
```

```
# [3]
print(f"VGGSsmall: {count(vgg):,} parameters ({n_head:,} in the head, "
      f"{n_head / count(vgg):.0%})")
print(f"test accuracy {accuracy(vgg, X_te, y_te):.1%}")
```

```
VGGSsmall: 218,586 parameters (202,122 in the head, 92%)
test accuracy 86.7%
```

Two readings again. The good: this VGG-style recipe reaches 86.7%, four points above LeNet in this run. Because depth, width, normalization, kernel sizes, parameter count, and training duration all changed together, this comparison does not isolate which ingredient earned the gain. The bad: 218,586 parameters, *three and a half times* LeNet's total, and the head's share got worse — 92% of the network is a dense layer reading a flattened grid. The convolutional diet succeeded and the total got fatter anyway. Which forces the third question: where do the parameters live, and where can we do the most damage to the count?

9.4. Question 3: 1×1 convolutions, and firing the flatten head

The bloat has one address: `nn.Linear(1568, 128)`. To evict it we need one more tool, odd-looking at first sight: a convolution whose kernel is 1×1 .

A 1×1 kernel does no spatial mixing at all — it looks at a single pixel. But across *channels* it does exactly what Equation 8.1 says: at each position, take the C_{in} channel values and form C_{out} weighted combinations plus bias. Pause on what that is: a **linear model across the channels, run separately at every pixel** — Chapter 1's machine, miniaturized and stamped across the image. Add a ReLU and each pixel is running a tiny MLP over its own feature vector. We *summarize* channels, we do not discard them: compressing 64 feature maps into 32 loses far less than pooling them away, and each compression injects another nonlinearity almost for free.

That tool enables the **Network-in-Network** design (Lin et al., 2013), and with it the clean head. A NiN block is one 3×3 (some spatial mixing) followed by two 1×1 s (per-pixel channel mixing). And the classifier becomes:

1. Let the last block produce a stack of feature maps.
2. **Global average pooling (GAP)**: average each map over all positions — 64 maps in, 64 numbers out. No parameters at all.
3. One small linear layer to the logits.

PLAN

1. Define the reusable helpers: `nin_block` and `NINSmall`.
 2. Prepare the inputs and fixed settings for the example.
 3. `NINSmall` — architecture and first 75 epochs.
-

CODE

```
# [1]
def nin_block(c_in: int, c_out: int) -> list[nn.Module]:
    return [nn.Conv2d(c_in, c_out, 3, padding=1, bias=False),
            nn.BatchNorm2d(c_out), nn.ReLU(),
            nn.Conv2d(c_out, c_out, 1), nn.ReLU(),
            nn.Conv2d(c_out, c_out, 1), nn.ReLU()]

class NINSmall(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            *nin_block(1, 16), nn.MaxPool2d(2),
            *nin_block(16, 32), nn.MaxPool2d(2),
            *nin_block(32, 64),
        )
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                                   nn.Linear(64, 10))
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.features(x))

# [2]
torch.manual_seed(6050)
nin = NINSmall()
nin_opt = torch.optim.Adam(nin.parameters(), lr=1e-3)
# [3]
nin = train_existing(nin, nin_opt, epochs=75)
```

PLAN

1. NINSmall — continue the same optimizer through epoch 150.
 2. Report or visualize the measured result.
-

CODE

```
# [1]
nin = train_existing(nin, nin_opt, epochs=75)
# [2]
print(f"NINSmall: {count(nin):,} parameters "
      f"(head: {count(nin.head):,})")
print(f"test accuracy {accuracy(nin, X_te, y_te):.1%}")
```

```
NINSmall: 35,034 parameters (head: 650)
test accuracy 76.2%
```

The parameter story is a rout: 35,034 total, a 650-parameter head, six times smaller than VGGSmall. The accuracy is 76.2%, and that dip is worth more attention than the win, so let us not rush past it. First, though, collect the promised payoff. Chapter 8 said the remaining shift-cliff was the flatten head's fault: it reads the final grid *positionally*. GAP averages over positions, so the head no longer assigns a different weight to every location. The rematch of the rematch:

PLAN

1. Define the reusable helpers: LeNet and `shift_right`.
 2. Shift curves, flatten head vs GAP head.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
class LeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, padding=2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.flatten(1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

# [2]
lenet = train_model(LeNet, epochs=150)  # Chapter 8's model, same recipe

def shift_right(X: torch.Tensor, px: int) -> torch.Tensor:
    if px == 0:
        return X.clone()
    out = torch.zeros_like(X)
    out[..., px:] = X[..., :-px]
    return out

shifts = list(range(5))
acc_lenet = [accuracy(lenet, shift_right(X_te, s), y_te) for s in shifts]
acc_nin = [accuracy(nin, shift_right(X_te, s), y_te) for s in shifts]

# [3]
for s in shifts:
    print(f"shift {s}px:  LeNet {acc_lenet[s]:.1%}  NIN+GAP {acc_nin[s]:.1%}")
```

```
plt.figure(figsize=(5.5, 3.2))
plt.plot(
    shifts, acc_lenet, "s-", color="#232D4B", lw=2, ms=7,
    label="LeNet (flatten head)",
)
plt.plot(
    shifts, acc_nin, "^-", color="#E57200", lw=2, ms=7,
    label="NINSmall (GAP head)",
)
plt.axhline(0.1, ls=":", color="#B8B8A8")
plt.xlabel("shift (pixels right)"); plt.ylabel("test accuracy")
plt.ylim(0, 0.9); plt.xticks(shifts); plt.legend()
plt.tight_layout(); plt.show()
```

shift 0px:	LeNet 82.5%	NIN+GAP 76.2%
shift 1px:	LeNet 74.8%	NIN+GAP 70.3%
shift 2px:	LeNet 62.3%	NIN+GAP 69.3%
shift 3px:	LeNet 45.3%	NIN+GAP 67.5%
shift 4px:	LeNet 26.8%	NIN+GAP 66.5%

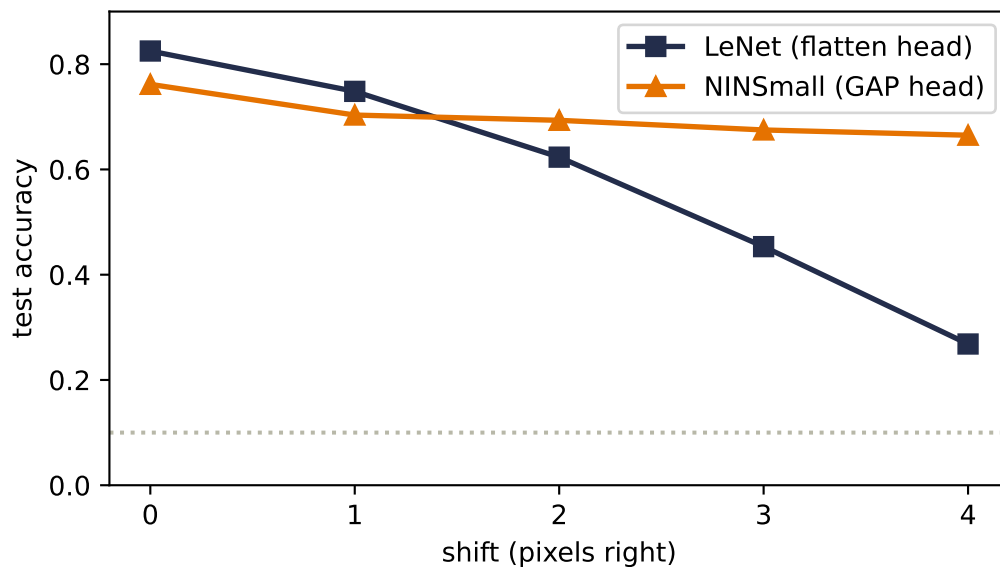


Figure 9.2.: Chapter 8’s experiment, third round. LeNet’s flatten head still slides below 30% by four pixels of shift. NINSmall with its global-average-pool head barely tilts: per-channel averages change much less when features move. GAP removes position-specific weights from the head, but padding, stride, pooling, and content clipped at the image edge keep the complete network from being exactly shift-invariant.

The curve that fell from 82% to 27% across this book’s Part II is now a gentle slope from 76% to 67%. Position-specific weights are gone from NINSmall’s head, and the result is consistent with the promised robustness benefit. It is not a matched-head ablation: the trunks and training recipes differ too. The

complete CNN is only approximately shift-tolerant; boundaries, strides, and pooling can still change its features.

Now the dip. On clean, centered test garments NINSmall trails LeNet by six points. That result is *consistent with* Chapter 8’s warning: on centered data, position can carry information, and a GAP head cannot assign separate weights to separate locations. But GAP is not the only changed ingredient, so this run does not prove it caused the gap. On the full 60,000-image dataset, the book’s pinned Rivanna runs put NiN at $92.78\% \pm 0.08\%$ across seeds 6050–6052. At our scale the comparison exposes a hypothesis worth testing with a matched-head ablation: shift tolerance may be bought with positional information.

i Question 4’s other famous block: Inception, in one breath

GoogLeNet’s Inception block (2014) answers “which kernel size?” with “all of them”: parallel 1×1 , 3×3 , 5×5 , and pooling branches, concatenated. Its enabling trick is the 1×1 *bottleneck*: compress channels before the expensive spatial kernels. For one 5×5 branch at 256 channels: direct, $5^2 \times 256 \times 128 \approx 819\text{k}$ parameters; with a 1×1 squeeze to 32 first, $256 \times 32 + 5^2 \times 32 \times 128 \approx 111\text{k}$ — an 86% cut for the same nominal operation. We won’t build Inception here (the principle, channel compression before spatial expense, is the transferable part), but Exercise 2 walks the arithmetic and the course assignment has you build the block itself.

i Architecture bridge (non-examinable): DenseNet concatenates the history

The residual block below preserves an old representation by **addition**: $x_{\ell+1} = x_{\ell} + F_{\ell}(x_{\ell})$. A DenseNet block makes a different connectivity choice. If x_j denotes layer j ’s output feature map, layer ℓ receives the channel-wise concatenation $[x_0, x_1, \dots, x_{\ell-1}]$ and contributes a small new group of feature maps for later layers. Earlier features therefore remain directly available instead of being recreated, while the channel axis grows; transition blocks compress channels and downsample between dense blocks. In Appendix B’s language, the defining operation is concatenation along the feature-channel axis, not another kind of residual addition (Appendix B).

9.5. Question 2: going deeper, and the wall you hit

VGG says depth is the direction. So push it: stack twenty of our two-conv blocks (with BN, per the recipe, on a 14×14 grid so the experiment fits a laptop) and compare against the *identical* stack with one change we’ll reveal after the numbers.

Predict before running: two networks, identical parameter counts, twenty blocks each — one plain, one residual. Will the plain net merely trail, or fail to fit the *training* set at all?

PLAN

1. Define the reusable helpers: `Block` and `DeepNet`.
 2. Train the plain 20-block network.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
class Block(nn.Module):
    def __init__(self, ch: int, residual: bool):
        super().__init__()
        self.c1 = nn.Conv2d(ch, ch, 3, padding=1, bias=False)
        self.b1 = nn.BatchNorm2d(ch)
        self.c2 = nn.Conv2d(ch, ch, 3, padding=1, bias=False)
        self.b2 = nn.BatchNorm2d(ch)
        self.residual = residual
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        y = self.b2(self.c2(F.relu(self.b1(self.c1(x)))))
        return F.relu(y + x) if self.residual else F.relu(y)

class DeepNet(nn.Module):
    def __init__(self, nblocks: int, residual: bool, ch: int = 12):
        super().__init__()
        self.stem = nn.Conv2d(1, ch, 3, padding=1)
        self.pool = nn.MaxPool2d(2) # 28 -> 14 early
        self.blocks = nn.Sequential(*[Block(ch, residual)
                                       for _ in range(nblocks)])
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                                   nn.Linear(ch, 10))
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.blocks(self.pool(F.relu(self.stem(x)))))

# [2]
plain = train_model(lambda: DeepNet(20, residual=False), epochs=30)
# [3]
print(f"plain-20      train {accuracy(plain, X_tr, y_tr):.1%}    "
      f"test {accuracy(plain, X_te, y_te):.1%}")
```

```
plain-20      train 60.8%    test 51.0%
```

 PLAN

1. Train the matched residual 20-block network.
 2. Report or visualize the measured result.
-

CODE

```
# [1]
resid = train_model(lambda: DeepNet(20, residual=True), epochs=30)
# [2]
print(f"residual-20  train {accuracy(resid, X_tr, y_tr):.1%}    "
      f"test {accuracy(resid, X_te, y_te):.1%}")
```

residual-20 train 100.0% test 77.8%

Look at the *training* column first, because it carries the whole lesson. The plain 40-conv-layer network cannot even fit the 1,200 images it sees every epoch: 61% train accuracy, while its residual twin memorizes them and generalizes twenty-seven points better. This is an **optimization failure** in the same family as the degradation problem He et al. reported in 2015, where their 56-layer plain net trained worse than their 20-layer one. Our experiment isolates the residual rescue at one depth; a strict demonstration that adding depth degrades a plain network would also require a shallower plain control.

The one change: each block computes $F(x)$ and outputs $F(x) + x$. A **residual connection** lets the input skip over the block and adds it back.

$$H(x) = F(x) + x \quad \Rightarrow \quad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial H} \left(\frac{\partial F}{\partial x} + I \right). \quad (9.2)$$

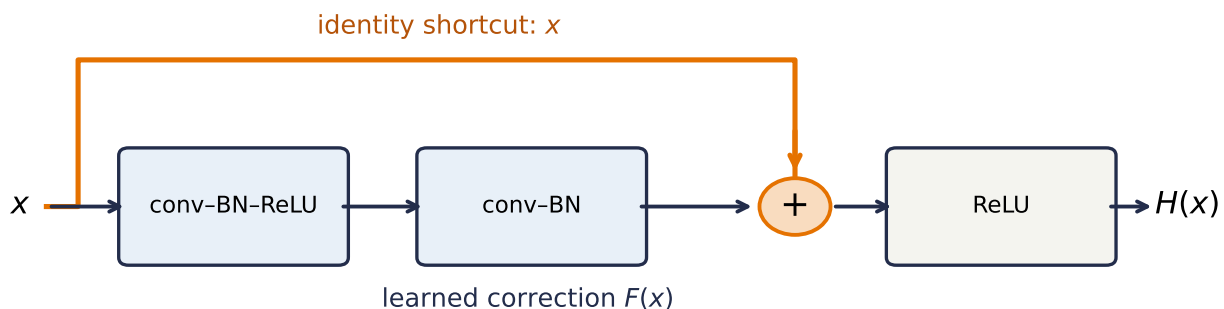


Figure 9.3.: A residual block has two routes. The learned branch computes a correction $F(x)$; the identity branch carries x unchanged to the addition. Even when the learned branch has a difficult Jacobian, the shortcut leaves a direct route for activations and gradients.

Read the right-hand side with Chapter 5 eyes. Blame arriving at a block’s output reaches its input through two additive terms: the learned branch $\partial F/\partial x$ and the identity I . The identity contribution gives gradient flow a direct route that does not itself multiply by the learned weights. It is powerful, not magical: the learned Jacobian can still reinforce, distort, or even partly cancel that term. Chapter 5 called ReLU’s open gate a *gradient superhighway* through a single unit; a skip connection builds a direct lane into every block. The block only needs to learn the *residual*, the correction on top of “pass it through,” and doing nothing is easy to represent: $F = 0$.

Here is the highway visible at initialization, extending Chapter 5’s depth experiment from dense layers to conv stacks:

PLAN

1. Define the reusable helpers: `PlainNoBN` and `stem_grad`.
2. Prepare the inputs and fixed settings for the example.
3. Gradient at the stem vs. depth, three designs.

CODE

```
# [1]
class PlainNoBN(nn.Module):
    def __init__(self, nconvs: int, ch: int = 12):
        super().__init__()
        self.stem = nn.Conv2d(1, ch, 3, padding=1)
        self.pool = nn.MaxPool2d(2)
        self.layers = nn.Sequential(*[m for _ in range(nconvs)
                                       for m in (nn.Conv2d(ch, ch, 3, padding=1),
                                                nn.ReLU())])
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                                   nn.Linear(ch, 10))
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.layers(self.pool(F.relu(self.stem(x)))))

def stem_grad(
    make_net: Callable[[], nn.Module],
) -> tuple[float, float, int, int]:
    torch.manual_seed(0)
    net = make_net()
    F.cross_entropy(net(X_tr[:128]), y_tr[:128]).backward()
    grad = net.stem.weight.grad
    return (grad.double().norm().item(), grad.abs().max().item(),
            torch.count_nonzero(grad).item(), grad.numel())

# [2]
depths = [4, 12, 24] # blocks (2 convs each)
diagnostics = {
    "plain, no BN": [stem_grad(lambda d=d: PlainNoBN(2 * d)) for d in depths],
    "plain + BN": [stem_grad(lambda d=d: DeepNet(d, False)) for d in depths],
    "residual + BN": [stem_grad(lambda d=d: DeepNet(d, True)) for d in depths],
}
curves = {name: [row[0] for row in rows]
           for name, rows in diagnostics.items()}

# [3]
for name, ys in curves.items():
    print(f"{name:14s}: " + " ".join(f"{v:.1e}" for v in ys))
norm64, maxabs, nonzero, total = diagnostics["plain, no BN"][-1]
print(
    f"48-layer no-BN check: max |component| {maxabs:.1e}; "
```

```

f"nonzero {nonzero}/{total}; float64 norm {norm64:.1e}"
)

plt.figure(figsize=(6.2, 3.4))
for (name, ys), color in zip(curves.items(),
                             ["#B8B8A8", "#E57200", "#232D4B"]):
    xs = [2 * d for d in depths]
    plt.semilogy(xs, ys, "o-", ms=5, color=color, label=name)
plt.xlabel("conv layers")
plt.ylabel(r"$\|\partial L / \partial W^{(\text{stem})}\|$")
plt.legend(); plt.tight_layout(); plt.show()

```

```

plain, no BN : 9.0e-06  3.3e-13  2.2e-23
plain + BN   : 7.6e-02  5.0e-01  8.3e+01
residual + BN: 1.6e-01  4.9e-01  5.0e-01
48-layer no-BN check: max |component| 5.2e-24; nonzero 108/108; float64 norm 2.2e-23

```

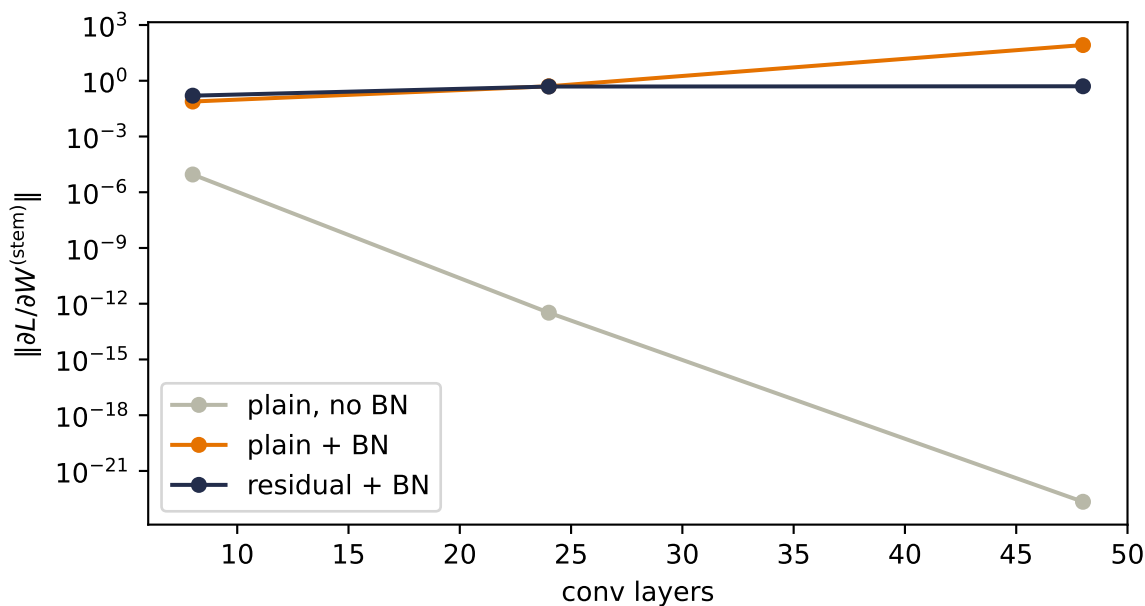


Figure 9.4.: Gradient magnitude reaching the stem (first layer) at initialization, versus depth, for three designs. In the 48-layer no-BN stack, the true norm is 2.2e-23; a float32 norm calculation reports zero because squaring these nonzero components underflows. BN fixes the vanishing but overshoots: in plain stacks the gradient grows with depth. Residual blocks keep the gradient within one order of magnitude across these depths.

The final no-BN row also exposes a numerical trap: its components remain nonzero, but squaring them in float32 can underflow before the norm is summed. The range, resolution, and accumulator roles behind that distinction are gathered in Appendix C.

💡 The most important seed this chapter plants

Residual connections are not only a CNN trick. They became a central design device in many deep architectures; when we assemble a transformer in [Chapter 14](#), every attention layer and every feedforward layer will be wrapped as $x + F(x)$, and the freshly-met layer normalization will sit beside each skip. The picture to carry forward: a *residual stream* with direct additive routes through the network, each block reading from it and writing a correction back. Those routes improve conditioning without making the stream immune to learned-branch interactions. Attention will be one kind of correction. You now own both parts of that skeleton.

9.6. The scorecard: what a decade of design bought

The chapter's small-data studies isolated mechanisms. The architecture comparison deserves the full task: all 60,000 Fashion-MNIST training images, split once into 50,000 for fitting and 10,000 for validation, with the official 10,000-image test set opened only after validation selected the checkpoint. We ran LeNet, NiN, VGG, and a nine-block residual network for three end-to-end seeds on Rivanna:

PLAN

1. Load the 12 pinned full-data Rivanna records.
 2. Summarize test accuracy and parameter count by architecture.
-

CODE

```
# [1]
import json
from pathlib import Path

scorecard_root = Path("../.. /experiments/rivanna/results/scorecard")
scorecard_records = [
    json.loads(path.read_text()) for path in sorted(scorecard_root.glob("*.json"))
]
scorecard_order = ["lenet", "nin", "vgg", "resnet"]
scorecard_colors = ["#B8B8A8", "#232D4B", "#E57200", "#5379AA"]

# [2]
scorecard = {}
for model_name in scorecard_order:
    rows = [row for row in scorecard_records if row["model"] == model_name]
    accuracies = torch.tensor([row["test_accuracy"] for row in rows])
    scorecard[model_name] = {
        "parameters": rows[0]["parameter_count"],
        "mean": accuracies.mean().item(),
        "sd": accuracies.std(unbiased=True).item(),
    }
```

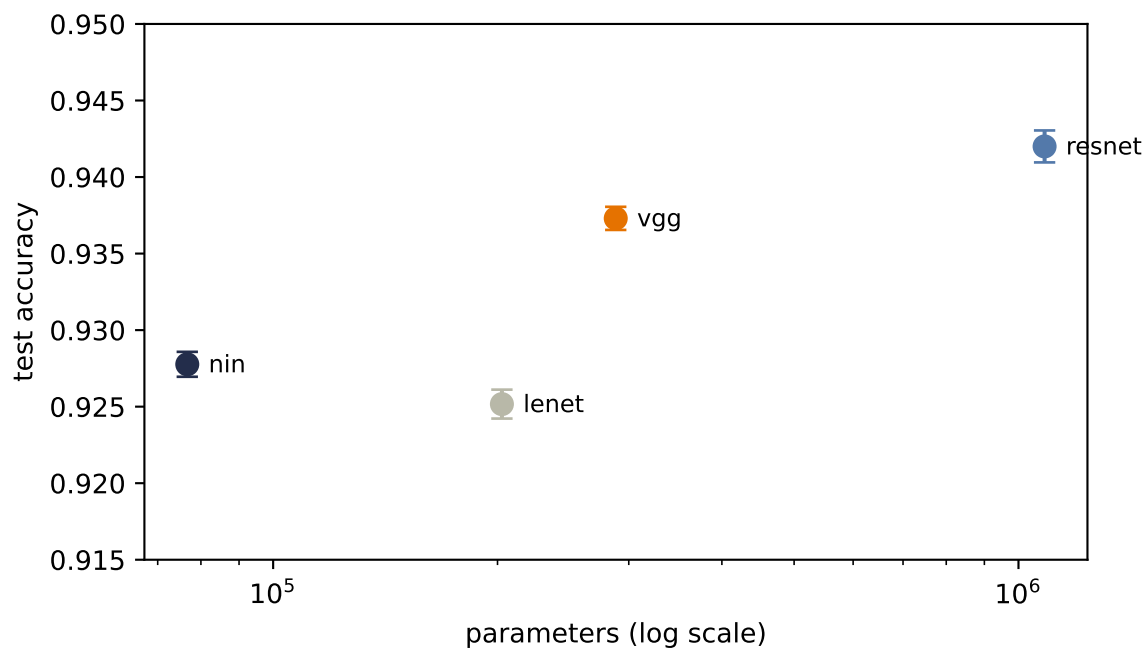


Figure 9.5.: The full-data Part II scorecard on Rivanna. Points show mean official-test accuracy and bars one sample standard deviation across seeds 6050–6052. LeNet reaches 92.52%, NiN 92.78%, VGG 93.73%, and the residual network 94.20%. NiN is the most parameter-efficient point; VGG buys more accuracy with depth; residual blocks reach the strongest endpoint in this declared training regime. Parameter count is not training compute, and these four points are not a universal architecture ranking.

The scale changes the verdict from the 1,200-image mechanism studies. NiN’s global head no longer pays a large clean-accuracy penalty, and all four architectures clear 92%. The residual network leads this declared recipe, but the plot still has one horizontal axis too few: parameters measure storage, not optimizer steps, activation memory, or total training work.

9.7. Transfer learning: the mechanics, and an honest experiment

Everything so far trains from scratch. The final move is the one that defines practice today: most image features are *shared* — edges, textures, parts transfer across tasks — so train one strong backbone once, on enormous data, and reuse it everywhere. The mechanics come in two strengths:

- **Linear probe:** freeze the pretrained trunk entirely (`requires_grad = False`), extract its features, train only a new linear head on your labels.
- **Fine-tuning:** additionally unfreeze the last block or two, training them at a *much* smaller learning rate than the fresh head, so the pretrained features adapt gently instead of being trampled.

We will test the idea properly. The task: classify the three shoe classes (sandal, sneaker, ankle boot) from **ten labeled examples each**. The 30-image regime is exactly where transfer should shine — too few labels to learn features, goes the story, so imported features should dominate. Three contenders:

1. **From scratch:** our VGG-style trunk trained on the 30 images alone.
2. **Our own pretrained trunk:** the same trunk pretrained on the seven *non-shoe* classes (863 images), frozen, linear probe.
3. **A real pretrained backbone:** SqueezeNet 1.1 trained on ImageNet — 1.2 million photographs, 1,000 classes — loaded from weights committed with this book, frozen, linear probe on its 512-dimensional features. (Fashion images get upsampled to 96×96 and repeated to three channels to fit its expectations. That awkwardness is part of the experiment, and part of the lesson.)

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `Trunk` and `squeeze_features`.
 3. Implement the shoe task, our own trunk, and the frozen probes.
 4. Report or visualize the measured result.
 5. Define the reusable helpers: `own_features` and `linear_probe`.
-

CODE

```
from torchvision.models import squeezenet1_1

# [1]
shoe = torch.tensor([5, 7, 9])
new_label = {5: 0, 7: 1, 9: 2}
is_shoe_tr = (y_tr[:, None] == shoe).any(1)
is_shoe_te = (y_te[:, None] == shoe).any(1)
X_shoe_te = X_te[is_shoe_te]
```

```

y_shoe_te = torch.tensor([new_label[int(c)] for c in y_te[is_shoe_te]])

# our own trunk: pretrain on the 7 non-shoe classes
src = [0, 1, 2, 3, 4, 6, 8]
remap = {c: i for i, c in enumerate(src)}
X_src = X_tr[~is_shoe_tr]
y_src = torch.tensor([remap[int(c)] for c in y_tr[~is_shoe_tr]])

# [2]
class Trunk(nn.Module):
    def __init__(self, n_classes: int):
        super().__init__()
        self.features = nn.Sequential(
            *atom(1, 16), *atom(16, 16), nn.MaxPool2d(2),
            *atom(16, 32), *atom(32, 32), nn.MaxPool2d(2), *atom(32, 64))
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                                   nn.Linear(64, n_classes))
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.features(x))

# [3]
own_trunk = train_model(lambda: Trunk(7), epochs=100, data=(X_src, y_src))
X_src_te = X_te[~is_shoe_te]
y_src_te = torch.tensor([remap[int(c)] for c in y_te[~is_shoe_te]])
# [4]
print(f"own trunk, source task (7 classes): "
      f"{accuracy(own_trunk, X_src_te, y_src_te):.1%}")

# the real thing: ImageNet SqueezeNet, from committed weights (no download)
sq = squeezenet1_1(weights=None)
sq.load_state_dict(torch.load("../data/squeezenet1_1-imagenet.pt"))
sq.eval()
for p in sq.parameters():
    p.requires_grad = False

IMNET_MEAN = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1)
IMNET_STD = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1)

@torch.no_grad()
def squeeze_features(X: torch.Tensor) -> torch.Tensor: # (N,1,28,28) -> (N,512)
    x = X.repeat(1, 3, 1, 1)
    x = F.interpolate(x, size=96, mode="bilinear", align_corners=False)
    f = sq.features((x - IMNET_MEAN) / IMNET_STD)
    return F.adaptive_avg_pool2d(f, 1).flatten(1)

# [5]
@torch.no_grad()
def own_features(X: torch.Tensor) -> torch.Tensor: # (N,1,28,28) -> (N,64)
    return F.adaptive_avg_pool2d(own_trunk.features(X), 1).flatten(1)

```



```
def linear_probe(Z_tr: torch.Tensor, y_f: torch.Tensor,
                 Z_te: torch.Tensor, seed: int) -> float:
    torch.manual_seed(seed)
    head = nn.Linear(Z_tr.shape[1], 3)
    opt = torch.optim.Adam(head.parameters(), lr=1e-2,
                            weight_decay=1e-3) # built-in L2 penalty (ch. 1's ridge)
    for _ in range(400):
        loss = F.cross_entropy(head(Z_tr), y_f)
        opt.zero_grad(); loss.backward(); opt.step()
    return (head(Z_te).argmax(1) == y_shoe_te).float().mean().item()
```

own trunk, source task (7 classes): 79.7%

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Scratch vs. own trunk vs. ImageNet trunk, three seeds.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
K = 10
rows = []
# [2]
for seed in [0, 1, 6050]:
    torch.manual_seed(seed)
    idx = torch.cat([(y_tr == c).nonzero().squeeze(1)
                     [torch.randperm(int((y_tr == c).sum()))[:K]] for c in shoe])
    X_few = X_tr[idx]
    y_few = torch.tensor([new_label[int(c)] for c in y_tr[idx]])

    scratch = train_model(lambda: Trunk(3), epochs=100, seed=seed,
                          data=(X_few, y_few))
    rows.append((accuracy(scratch, X_shoe_te, y_shoe_te),
                 linear_probe(own_features(X_few), y_few,
                                own_features(X_shoe_te), seed),
                 linear_probe(squeeze_features(X_few), y_few,
                                squeeze_features(X_shoe_te), seed)))

# [3]
print("seed      scratch   own-trunk probe   ImageNet probe")
for seed, r in zip([0, 1, 6050], rows):
    print(f"{seed:4d}      {r[0]:.1%}      {r[1]:.1%}      {r[2]:.1%}")
mean = [sum(c) / 3 for c in zip(*rows)]
print(f"mean      {mean[0]:.1%}      {mean[1]:.1%}      {mean[2]:.1%}")
```

seed	scratch	own-trunk probe	ImageNet probe
0	86.4%	68.9%	88.1%
1	85.3%	65.5%	85.9%
6050	88.1%	77.4%	87.6%
mean	86.6%	70.6%	87.2%

Read that table slowly, because it does *not* say what the textbook story predicts, and the discrepancy is the best lesson in this chapter. Training from scratch on thirty images fights the mighty ImageNet backbone to a dead heat — the means differ by less than a point, well inside seed noise. And our own pretrained trunk, perfectly competent on its source task per the printout above, transfers *worse than nothing*. Three reasons, each a general principle:

1. **A trunk can only donate what its data taught it.** Our own trunk saw 863 shirts, bags, and trousers. Nothing in that curriculum required foot-shaped detectors, so its 64 features flatten sandals, sneakers, and boots into nearly the same point. Pretraining is not magic; it is *curriculum*.
2. **Domain and resolution gaps tax the donation.** SqueezeNet’s filters expect 224-pixel color photographs; we feed it 28-pixel grayscale icons inflated to 96. Its early layers hunt for detail that simply is not there. (The matching recipe, *resize and normalize to match the pretraining pipeline* — is doing real work; we complied as far as the data allows.)
3. **Transfer wins when the target is big relative to your labels — not small.** This is the quiet one. Three shoe silhouettes at 28×28 is a *small* problem: thirty images genuinely suffice, so the scratch baseline is strong and there is little room for imported knowledge to pay rent. The full ten-class task below is different: 50,000 fitting labels, 224-pixel inputs, and a ResNet-18-sized feature extractor.

9.7.1. The full-data rematch

The full task lets us separate three questions cleanly: what ImageNet features can do without changing, what the last residual stage can learn when allowed to adapt, and what the architecture can learn from scratch with all target labels available.

PLAN

1. Load the nine pinned ResNet-18 transfer records.
 2. Summarize official-test accuracy by training regime.
-

CODE

```
# [1]
transfer_root = Path("../.. /experiments/rivanna/results/transfer")
transfer_records = [
    json.loads(path.read_text()) for path in sorted(transfer_root.glob("*.json"))
]
transfer_order = ["probe", "finetune", "scratch"]

# [2]
transfer_summary = {}
```

```

for regime in transfer_order:
    values = torch.tensor([
        row["test_accuracy"] for row in transfer_records if row["regime"] == regime
    ])
    transfer_summary[regime] = {
        "values": values,
        "mean": values.mean().item(),
        "sd": values.std(unbiased=True).item(),
    }

```

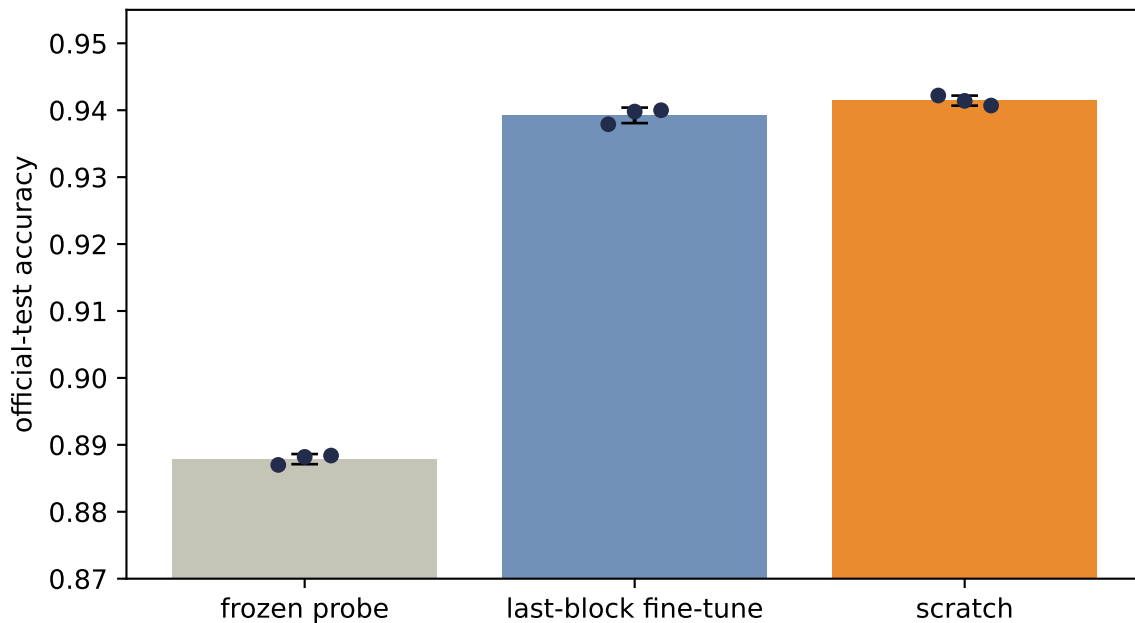


Figure 9.6.: ImageNet transfer on full Fashion-MNIST at 224 pixels. Across seeds 6050–6052, the frozen linear probe reaches $88.79\% \pm 0.08\%$ official-test accuracy, last-block fine-tuning $93.92\% \pm 0.12\%$, and scratch $94.14\% \pm 0.08\%$. Fine-tuning repairs most of the domain mismatch, but 50,000 target labels are enough for scratch to match it in this regime. Bars are means, dots are seeds, and error bars are one sample standard deviation.

The frozen probe's 88.79% is the distribution gap made visible: features learned from natural photographs do not linearly separate all ten grayscale garment classes at this resolution. Updating only the last residual stage recovers more than five points. Scratch ends 0.22 points above fine-tuning, smaller than the run-to-run variation one would need to resolve as a family-level claim. With 50,000 target labels, scratch is no longer feature-starved. Transfer changed the starting point and permitted updates; it did not guarantee the best final endpoint.

💡 When to reach for a pretrained backbone

The honest decision rule supported by the experiments is: transfer pays when (your labels are scarce) *and* (the target task is feature-hungry) *and* (the pretraining data plausibly covers the target's features at a matched scale). At 224 pixels and 18 landmark classes — the course assignment — all three hold, and transfer is the winning move by a wide margin. At 28 pixels and 3 silhouettes,

the conditions fail and scratch fights the superpower to a draw. Knowing *which regime you are in* is the skill; the mechanics (`requires_grad`, learning-rate splits) are the easy part.

The mechanics, for completeness, since you will use them constantly from Part V onward: fine-tuning SqueezeNet's last block with a two-learning-rate recipe (fresh head fast, pretrained layers slow):

PLAN

1. Define the reusable `prep` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Fine-tuning — unfreeze the last block, two learning rates.
 4. Report or visualize the measured result.
-

CODE

```
# [1]
def prep(X: torch.Tensor) -> torch.Tensor:
    x = X.repeat(1, 3, 1, 1)
    x = F.interpolate(x, size=96, mode="bilinear", align_corners=False)
    return (x - IMNET_MEAN) / IMNET_STD

# [2]
torch.manual_seed(6050)
idx = torch.cat([(y_tr == c).nonzero().squeeze(1)
                  [torch.randperm(int((y_tr == c).sum()))[:K]] for c in shoe])
X_few, y_few = prep(X_tr[idx]), torch.tensor([new_label[int(c)]
                                              for c in y_tr[idx]])

ft = squeezenet1_1(weights=None)
ft.load_state_dict(torch.load("../data/squeezenet1_1-imagenet.pt"))
head = nn.Linear(512, 3)
# [3]
for p in ft.parameters():
    p.requires_grad = False
for p in ft.features[-1].parameters():          # last Fire block only
    p.requires_grad = True

opt = torch.optim.Adam([
    {"params": head.parameters(), "lr": 1e-3},      # fresh head: normal
    {"params": ft.features[-1].parameters(), "lr": 1e-4}, # trunk: gentle
], weight_decay=1e-3)

ft.train()
for _ in range(60):
    z = F.adaptive_avg_pool2d(ft.features(X_few), 1).flatten(1)
    loss = F.cross_entropy(head(z), y_few)
    opt.zero_grad(); loss.backward(); opt.step()
```

```
ft.eval()
# [4]
with torch.no_grad():
    z = F.adaptive_avg_pool2d(ft.features(prepare(X_shoe_te)), 1).flatten(1)
    print(f"fine-tuned (last block + head): "
          f"{(head(z).argmax(1) == y_shoe_te).float().mean():.1%}")
```

fine-tuned (last block + head): 88.7%

Fine-tuning edges the frozen probe on this seed and lands in the same tie with scratch. This is consistent with the regime analysis above: adaptation helps at the margin, but no amount of it makes a small target task need imported features.

Keep the pattern, though: *this* loop, scaled to real backbones and feature-hungry tasks, is how most of applied deep learning ships today. It is also this book's destination: Part V is what happened when the field noticed that “pretrain big, adapt cheaply” was not a vision trick but *the* recipe for language too. The BERT moment ([Chapter 15](#)) is this section's idea, taken seriously at scale.

i Check yourself

Close the book for one minute and reconstruct the architecture and transfer choices.

- Why can several small kernels replace one large kernel?
- What changes between a frozen probe, partial fine-tuning, and training from scratch?
- Which evidence here is a mechanism study, and which comes from the pinned Rivanna runs?

9.8. Okay, so — modern CNNs are an optimization story

1. **Small kernels, stacked:** two 3×3 s see what a 5×5 sees, for $18C^2$ against $25C^2$, with twice the nonlinearity — provided width holds constant through the stack. Design in blocks, repeat the block.
2. **BatchNorm** re-standardizes every channel over the batch and spatial axes, then hands back two knobs (γ, β) ; conv→BN→ReLU with `bias=False`; *train and eval are different machines*, so flip the switch. Its per-token cousin, LayerNorm, awaits in [Chapter 14](#). For the graduate treatment of normalization as a choice of invariance, finite-sample state, and derivative path, continue in [Normalization Chooses an Invariance](#).
3. 1×1 **convolutions** run Chapter 1's linear model across channels at every pixel: summarize channels, don't discard them. With **global average pooling** they fire the flatten head: parameters collapse ($218k \rightarrow 35k$) and position-specific weights leave the head, at a clean-accuracy price our small dataset makes visible. Boundaries and downsampling still prevent exact invariance.
4. **Depth can hit an optimization wall:** our plain 40-layer net couldn't fit its own training set. The residual connection $H(x) = F(x) + x$ adds an identity lane to the Jacobian, Chapter 5's gradient superhighway as infrastructure, and the same-depth twin trains to 100%. Transformer stacks will reuse this residual pattern around each attention and feedforward sublayer.
5. **Transfer learning** = freeze + probe, or gently fine-tune. Our honest experiment found scratch and the ImageNet probe in a near tie across three seeds at 28-pixel scale. Pretraining is curriculum, gaps are taxed, and small targets may leave little room for imports. The pinned full-scale Rivanna runs report $88.79\% \pm 0.08\%$ for the frozen probe, $93.92\% \pm 0.12\%$ for fine-tuning, and $94.14\% \pm 0.08\%$

for scratch with ResNet-18. They illustrate a richer regime where adaptation repairs most of the feature mismatch. Diagnosing the regime is the skill.

Sources and further reading

- Xiao, Rasul, and Vollgraf, *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*: the target benchmark for the architecture and transfer studies.
- Deng et al., *ImageNet: A Large-Scale Hierarchical Image Database*: the source benchmark behind the pretrained visual features used here.
- Iandola et al., *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*: the compact pretrained architecture used in the chapter’s first transfer experiment.
- Simonyan & Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*: the small-kernel, repeated-block argument behind VGG.
- Lin, Chen, & Yan, *Network In Network*: introduces the 1×1 channel mixer and global-average-pooling classifier.
- Ioffe & Szegedy, *Batch Normalization*: the original normalization method and its train/evaluation distinction.
- He et al., *Deep Residual Learning for Image Recognition*: the degradation experiment and residual-block solution.
- Huang et al., *Densely Connected Convolutional Networks*: dense connectivity through feature-map concatenation and transition layers.
- Yosinski et al., *How Transferable Are Features in Deep Neural Networks?*: a careful empirical account of when transferred features remain useful.
- Hardt and Recht, *Patterns, Predictions, and Actions: Datasets*: a framework for treating a benchmark as a measured artifact rather than an unquestionable oracle.
- Northcutt, Athalye, and Mueller, *Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks*: a large-scale audit of suspected benchmark-label errors and their effect on model comparison.

Exercises

1. **(Pencil.)** Generalize the kernel-stacking arithmetic: n stacked 3×3 layers versus one $(2n+1) \times (2n+1)$ kernel, at constant width C . Then redo it for a layer that grows channels $C \rightarrow 2C$: split into two 3×3 s ($C \rightarrow C \rightarrow 2C$ and $C \rightarrow 2C \rightarrow 2C$) and find when the split stops saving parameters.
2. **(Pencil.)** Verify the Inception bottleneck arithmetic from the callout (819,200 versus 110,592), then design the cheapest 1×1 -plus- 5×5 pipeline from 128 channels to 64 output channels that keeps the squeeze width at least a quarter of the input width.
3. **(Code.)** Make VGGSmall’s blocks *depthwise separable*: replace each 3×3 conv with a per-channel 3×3 (`groups=c_in`) followed by a 1×1 mixer. Count parameters, retrain, and report the accuracy-per-parameter ratio against the original. (This is MobileNet’s depthwise-separable construction.)
4. **(Code.)** Ablate BatchNorm: retrain VGGSmall with the BN layers removed, same budget. Compare final accuracy *and* the first ten epochs’ training accuracy. Then repeat the `bn-drift` cell’s measurement on both trained networks. Does training itself fix the activation scales?
5. **(Code.)** Data augmentation as a third road: on the 30-image shoe task, train from scratch with random horizontal flips and ± 3 -pixel shifts (Chapter 6’s exercise, now as a tool). Does

augmentation close the gap to the pinned full-data results further than transfer did? Why might augmentation and transfer help in *different* regimes?

6. **(Audit.)** Retrain `VGGSmall` and `NINSmall` under seed 6050, then collect the 20 Fashion-MNIST test items on which the models disagree with each other or with the supplied label while at least one model assigns its prediction probability above 0.9. Preserve the images, supplied labels, predictions, and confidences before adjudication. Have a second reader independently label the 20 items, report agreement and a binomial interval for the suspected-label-error share, and list ambiguous cases separately. Explain why confident disagreement is a biased sample of the full test set, in which direction that selection can distort prevalence, and what the audit can still reveal.

Interlude: Autoencoders—Making PCA Learnable

Part II gave neural networks a spatial prior: local kernels share one rule across an image. The next part must handle objects whose length is not known in advance. Before we add a loop, the course takes a revealing detour—one that begins with a fixed-size object. Could an encoder compress an object into a fixed-size code and a decoder recover what mattered?

That compression-and-reconstruction machine is an **autoencoder**. We will first learn its contract, then discover that principal component analysis (PCA) was already its closed-form linear case. Gradient descent makes the same map learnable; nonlinearity lets it bend around curved data. Its success will expose why a one-shot encoder is not yet a variable-length process.

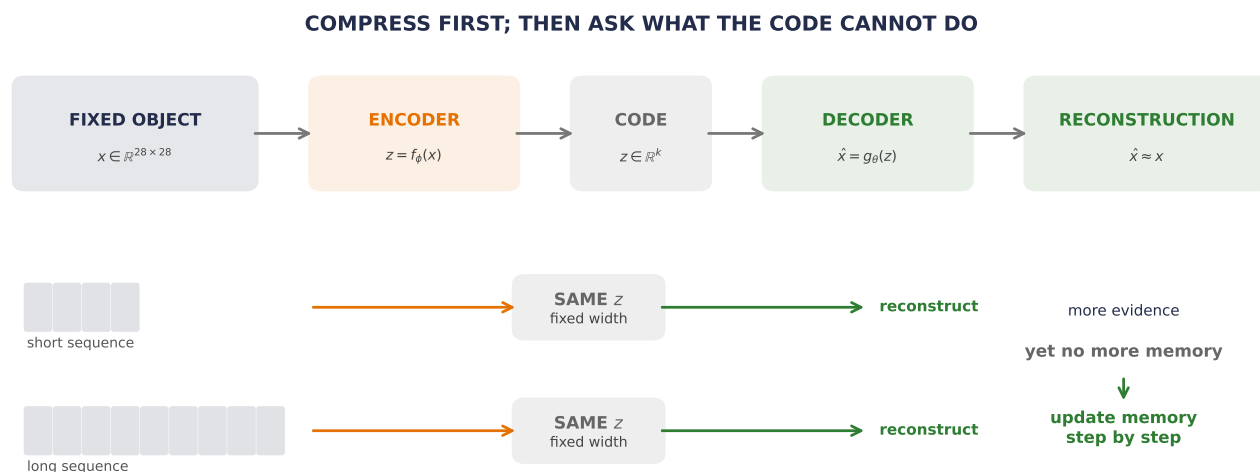


Figure AE.1: The encoder–decoder detour solves one problem and reveals the next. A fixed-size object can be compressed and reconstructed. Variable-length sequences can also be forced through one code, but its width does not grow with the evidence. The unmet need is memory that updates as each new piece arrives.

Compress, then reconstruct

Let us postpone random sampling for a moment and ask a simpler question. If an input has d coordinates, can a network preserve its important structure using only $k < d$ numbers?

An **autoencoder** contains two maps. For one input $\mathbf{x} \in \mathbb{R}^d$, the encoder produces a code $\mathbf{z} \in \mathbb{R}^k$ and the decoder reconstructs the input:

$$\mathbf{z} = f_\phi(\mathbf{x}), \quad \hat{\mathbf{x}} = g_\theta(\mathbf{z}).$$

The narrow code is the **bottleneck**. For n examples stored as rows of $\mathbf{X} \in \mathbb{R}^{n \times d}$, a squared reconstruction objective is

$$\mathcal{L}_{\text{AE}}(\phi, \theta) = \frac{1}{nd} \sum_{i=1}^n \|\mathbf{x}_i - g_{\theta}(f_{\phi}(\mathbf{x}_i))\|_2^2.$$

The target is available for free—the input supplies it—but the objective is still specific: preserve enough information to reconstruct the input. This is commonly called **self-supervised** because each input supplies its own target. This chapter also groups it under unsupervised representation learning. Mean squared error is one choice—suitable when independent, equal-variance Gaussian error is a reasonable model. It is not the universal autoencoder loss.

The hourglass shape creates pressure. If k is small and the maps have controlled capacity, useful regularities may be cheaper to preserve than individual accidents. But pressure is not proof.

 A bottleneck is pressure, not a certificate

A real-valued low-dimensional code can still carry surprising information—and a high-capacity network can memorize a finite training set. Check held-out reconstruction, latent behavior, and the decoder between observed codes. “The bottleneck learned meaning” is an empirical claim—not a consequence of $k < d$.

Make PCA learnable

Appendix A computes PCA from a centered singular value decomposition. Module 6 asks the more neural-network-shaped question: what if we replace that closed-form solve with weights and a reconstruction loss? The answer begins with the precise statement **PCA is a linear autoencoder**. Let the training rows be centered, so their mean is zero. Let $\mathbf{V}_k \in \mathbb{R}^{d \times k}$ contain k orthonormal principal directions:

$$\mathbf{V}_k^{\top} \mathbf{V}_k = \mathbf{I}_k.$$

PCA encodes and decodes by

$$\mathbf{z} = \mathbf{V}_k^{\top} \mathbf{x}, \quad \hat{\mathbf{x}} = \mathbf{V}_k \mathbf{z} = \mathbf{V}_k \mathbf{V}_k^{\top} \mathbf{x}.$$

The product $\mathbf{P}_k = \mathbf{V}_k \mathbf{V}_k^{\top} \in \mathbb{R}^{d \times d}$ is the orthogonal projector onto the principal k -dimensional subspace. Among all linear reconstruction maps of rank at most k , this projector minimizes squared reconstruction error. Now replace the SVD with gradient descent. A tied linear autoencoder uses one trainable matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$ in both directions:

$$\mathbf{z} = \mathbf{W}^{\top} \mathbf{x}, \quad \hat{\mathbf{x}} = \mathbf{W} \mathbf{W}^{\top} \mathbf{x}.$$

At a global optimum under the centered, squared-loss, undercomplete conditions, it can recover the same principal reconstruction subspace. That is the accurate content of “PCA is a linear autoencoder”—not a claim that every optimizer run succeeds or that every learned weight equals a particular PCA basis.

i Same subspace, not the same coordinates

For any orthogonal $\mathbf{Q} \in \mathbb{R}^{k \times k}$,

$$(\mathbf{V}_k \mathbf{Q})(\mathbf{V}_k \mathbf{Q})^\top = \mathbf{V}_k \mathbf{V}_k^\top.$$

The latent axes may rotate or reflect—the reconstruction projector stays fixed. For an untied encoder and decoder, any invertible latent change of basis can cancel between the two maps. Compare reconstruction products, projectors, or principal angles, not raw latent coordinates.

i Framework previews: SVD and tanh

The experiment uses `torch.linalg.svd` only to compute PCA’s closed-form reference after the projection has been derived above; Appendix A opens that decomposition and its shape/centering contract. It also uses `tanh`, the smooth pointwise map $\tanh(u) \in (-1, 1)$, as the nonlinear activation inside the curved autoencoder. Chapter 3 already supplied the activation-layer idea. Here `tanh` is simply a nonlinear map; its bounded range and derivative will matter only when repeated state updates make them a design constraint.

What if the map could bend?

PCA gives the best **flat** rank- k reconstruction. What if the encoder and decoder were learnable nonlinear maps that could bend with the data?

Let us use a controlled curve with a known one-dimensional coordinate. For $t \in [-1, 1]$, define

$$\mathbf{x}(t) = \left(t, 1.5 \left(t^2 - \frac{1}{3} \right) \right)^\top.$$

The alternating grid points form train and held-out sets. PCA and a tied linear autoencoder receive a one-dimensional bottleneck, as does a nonlinear $2 \rightarrow 24 \rightarrow 1 \rightarrow 24 \rightarrow 2$ autoencoder. The comparison is intentionally about representational shape, not parameter efficiency—the nonlinear model has far more adjustable knobs.

PLAN

1. Define the reusable helpers: `planted_curve` and `principal_projection`.
 2. Define the reusable helpers: `CurveAutoencoder` and `fit_curve`.
 3. Prepare the inputs and fixed settings for the example.
 4. Rematch PCA and linear and nonlinear autoencoders on a planted curve.
 5. Report or visualize the measured result.
-

CODE

```

# [1]
def planted_curve(t: Tensor) -> Tensor:
    return torch.stack((t, 1.5 * (t.square() - 1.0 / 3.0)), dim=1)

def principal_projection(train_x: Tensor) -> tuple[Tensor, Tensor]:
    mean = train_x.mean(dim=0)
    _, _, vh = torch.linalg.svd(train_x - mean, full_matrices=False)
    return mean, vh[:,1].T

# [2]
class CurveAutoencoder(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(2, 24), nn.Tanh(), nn.Linear(24, 1))
        self.decoder = nn.Sequential(nn.Linear(1, 24), nn.Tanh(), nn.Linear(24, 2))

    def forward(self, x: Tensor) -> tuple[Tensor, Tensor]:
        z = self.encoder(x)
        return self.decoder(z), z

def fit_curve(seed: int) -> tuple[float, float, float, float, float,
    ↪ CurveAutoencoder]:
    torch.manual_seed(seed)
    t_all = torch.linspace(-1.0, 1.0, 513)
    train_t, test_t = t_all[:,2], t_all[1::2]
    train_x, test_x = planted_curve(train_t), planted_curve(test_t)
    mean, basis = principal_projection(train_x)

    pca_reconstruction = (test_x - mean) @ basis @ basis.T + mean
    pca_mse = F.mse_loss(pca_reconstruction, test_x).item()

    tied_weight = nn.Parameter(torch.randn(2, 1) * 0.2)
    tied_optimizer = torch.optim.Adam([tied_weight], lr=0.03)
    centered_train = train_x - mean
    for _ in range(1800):
        tied_reconstruction = centered_train @ tied_weight @ tied_weight.T
        tied_loss = F.mse_loss(tied_reconstruction, centered_train)
        tied_optimizer.zero_grad()
        tied_loss.backward()
        tied_optimizer.step()

    tied_test = (test_x - mean) @ tied_weight @ tied_weight.T + mean
    tied_mse = F.mse_loss(tied_test, test_x).item()
    projector_gap = torch.linalg.norm(
        tied_weight @ tied_weight.T - basis @ basis.T

```

```

).item()

model = CurveAutoencoder()
optimizer = torch.optim.Adam(model.parameters(), lr=0.012)
for step in range(4000):
    reconstruction, _ = model(train_x)
    loss = F.mse_loss(reconstruction, train_x)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if step == 2500:
        for group in optimizer.param_groups:
            group["lr"] = 0.003

with torch.no_grad():
    nonlinear_reconstruction, z = model(test_x)
    nonlinear_mse = F.mse_loss(nonlinear_reconstruction, test_x).item()
    centered_z = z[:, 0] - z[:, 0].mean()
    centered_t = test_t - test_t.mean()
    latent_correlation = (
        centered_z @ centered_t
        / torch.sqrt((centered_z @ centered_z) * (centered_t @ centered_t))
    ).abs().item()
return pca_mse, tied_mse, projector_gap, nonlinear_mse, latent_correlation, model

# [3]
curve_rows = []
curve_models = []
# [4]
for curve_seed in range(6050, 6055):
    *curve_metrics, curve_model = fit_curve(curve_seed)
    curve_rows.append(curve_metrics)
    curve_models.append(curve_model)

curve_table = torch.tensor(curve_rows)
t_plot = torch.linspace(-1.0, 1.0, 257)
x_plot = planted_curve(t_plot)
plot_mean, plot_basis = principal_projection(planted_curve(torch.linspace(-1, 1,
    ↪ 257)))
pca_plot = (x_plot - plot_mean) @ plot_basis @ plot_basis.T + plot_mean
with torch.no_grad():
    nonlinear_plot, z_plot = curve_models[0](x_plot)

# [5]
print("seed  PCA MSE    tied MSE    projector gap  nonlinear MSE  |corr(z,t)|")
for seed, row in zip(range(6050, 6055), curve_rows):
    print(seed, *(f"{value:.9f}" for value in row))
print("mean", *(f"{value:.9f}" for value in curve_table.mean(dim=0).tolist()))

```

```
print("SD ", *(f"{value:.9f}" for value in curve_table.std(dim=0).tolist()))
```

seed	PCA MSE	tied MSE	projector gap	nonlinear MSE	corr(z,t)
6050	0.100000030	0.100000030	0.000000000	0.000001906	0.997166909
6051	0.100000030	0.100000030	0.000000000	0.000002822	0.985270616
6052	0.100000030	0.100000030	0.000000000	0.000002622	0.998887989
6053	0.100000030	0.100000030	0.000000000	0.000004865	0.989845965
6054	0.100000030	0.100000030	0.000000000	0.000001859	0.992839221
mean	0.100000030	0.100000030	0.000000000	0.000002815	0.992802140
SD	0.000000000	0.000000000	0.000000000	0.000001223	0.005512553

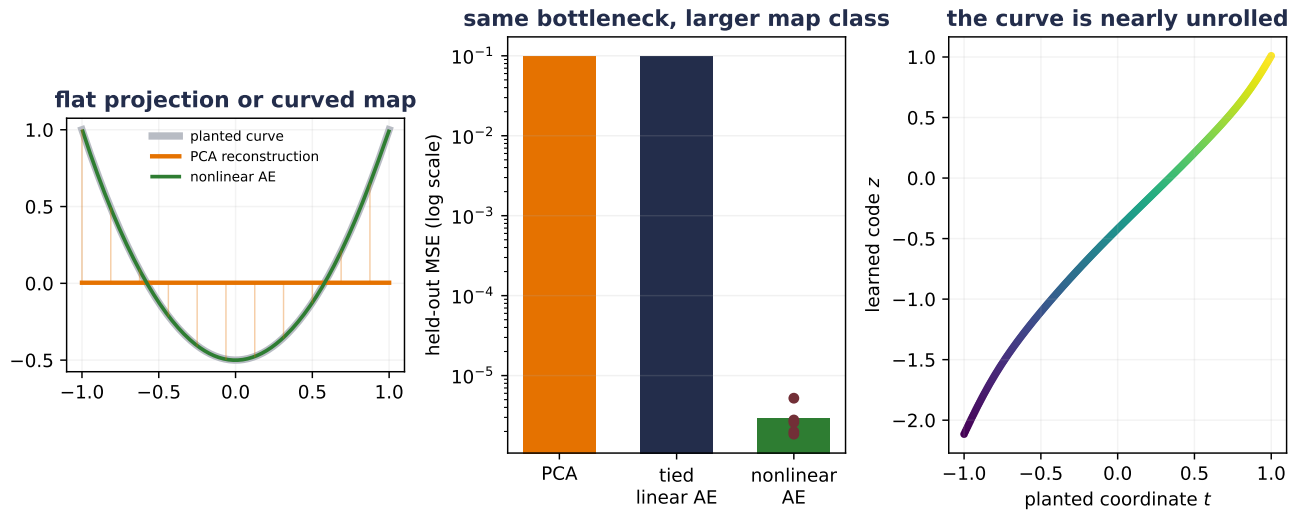


Figure AE.2: A one-dimensional bottleneck behaves differently when its maps may bend. Rank-one PCA and the tied linear autoencoder recover the same projector here; the nonlinear autoencoder follows the planted curve and nearly unrolls its coordinate. This capacity-mismatched study shows a mechanism, not a universal nonlinear-autoencoder advantage.

All five linear runs recover the principal projector at the displayed precision. The nonlinear decoder follows the curve between alternating training points, and its one-dimensional code is almost monotone in the coordinate that generated the data. This is the memorable “**PCA on steroids**” picture, with an important qualification: nonlinearity enlarges the reconstruction class from flat subspaces to curved maps. It does not guarantee that the chosen code is semantic, stable outside the observed region, or unique.

The same point appears in a rolled two-dimensional surface in three dimensions. The smaller curve here makes the estimand visible and leaves us a true held-out interleaving test: did the decoder learn to follow the bend between training points? The geometry changes; the claim does not.

Learning a lower-dimensional coordinate system for data concentrated near a curved set is one form of **manifold learning**. The phrase names the goal—not a guarantee that every bottleneck recovers a true manifold or its preferred coordinates.

A code is not yet a distribution

The nonlinear autoencoder can reconstruct held-out points on this curve. Can we now generate by drawing an arbitrary random z and passing it through the decoder?

Nothing in the reconstruction objective tells us which distribution to draw from. Reconstruction trains the decoder on codes produced by the encoder. Between those codes—and especially outside their range—many decoder functions can fit the same targets. A small algebraic example makes the missing contract visible. Suppose observed codes are $z \in \{-1, 0, 1\}$ and one decoder is $g_1(z) = z^2$. A second decoder can add any multiple of a function that vanishes at all three codes:

$$g_2(z) = z^2 + 0.8z(z^2 - 1).$$

The reconstructions agree on every observed code—yet disagree elsewhere.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Show why reconstruction alone does not identify latent sampling.
-

CODE

```
# [1]
latent_grid = torch.linspace(-1.45, 1.45, 400)
decoder_one = latent_grid.square()
decoder_two = latent_grid.square() + 0.8 * latent_grid * (latent_grid.square() - 1.0)
observed_codes = torch.tensor([-1.0, 0.0, 1.0])
observed_targets = observed_codes.square()
random_code = torch.tensor(0.5)

# [2]
training_gap = (
    observed_codes.square()
    - (observed_codes.square() + 0.8 * observed_codes * (observed_codes.square() -
        ↪ 1))
).abs().max()
print(f"largest decoder gap at observed codes: {training_gap.item():.1f}")
print(f"g1(0.5): {random_code.square().item():.2f}")
g2_random = random_code.square() + 0.8 * random_code * (random_code.square() - 1)
print(
    "g2(0.5):",
    f"{g2_random.item():.2f}",
)
```

```
largest decoder gap at observed codes: 0.0
g1(0.5): 0.25
g2(0.5): -0.05
```

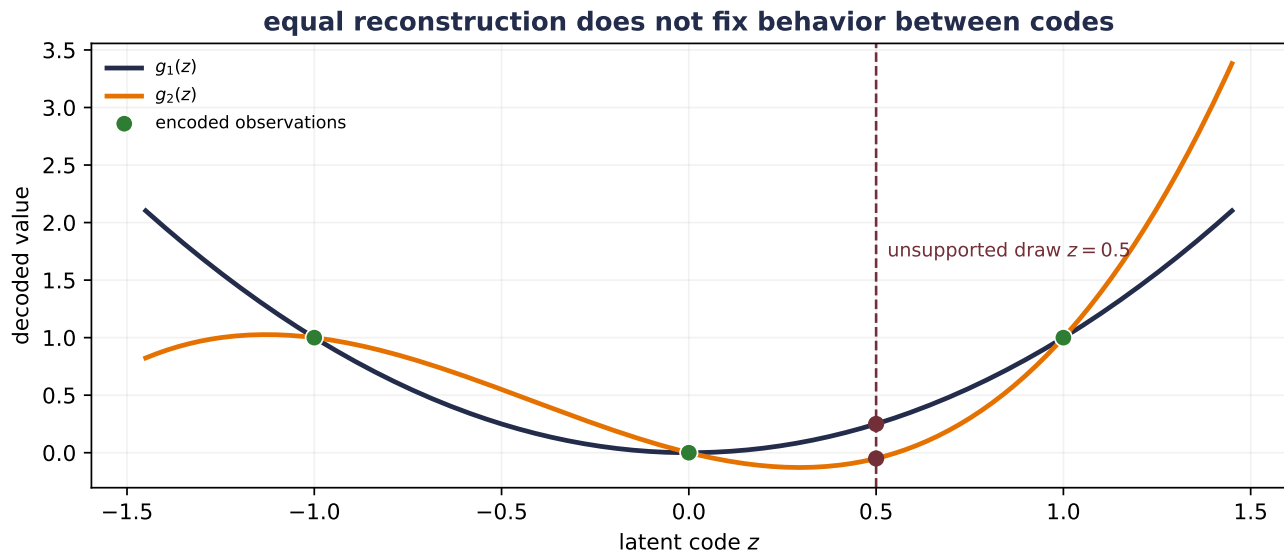


Figure AE.3: Reconstruction does not identify a latent sampler or a unique decoder away from observed codes. These decoders agree exactly at $z = -1, 0, 1$ and disagree between and beyond those codes; at the unsupported draw $z = 0.5$, their outputs differ.

⚠ Reconstruction does not specify a sampler

A deterministic autoencoder may still be used inside a generative system, and one can fit a separate distribution to its codes. The narrower statement is the important one—ordinary reconstruction training supplies no built-in latent prior and no principled rule for arbitrary random draws. A code is not yet a probability distribution.

One useful intermediate move is **same architecture, different training**. A denoising autoencoder corrupts an input to $\tilde{\mathbf{x}}$ and asks the network to recover the clean $\mathbf{x}$:

$$\mathcal{L}_{\text{DAE}} = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}, \tilde{\mathbf{x}} \sim q(\tilde{\mathbf{x}}|\mathbf{x})} \left[\|\mathbf{x} - g_{\theta}(f_{\phi}(\tilde{\mathbf{x}}))\|_2^2 \right].$$

Clean $\rightarrow$ clean learns reconstruction. Under squared loss, noisy $\rightarrow$ clean trains the output toward the conditional mean of clean inputs compatible with the corrupted observation—same architecture, different training. Different-domain input $\rightarrow$ target uses the same contract for translation. Changing the input–target contract changes what the code must preserve.

Let the autoencoder see locally

A fully connected autoencoder flattens an image and forgets the spatial prior that Part II worked so hard to earn. A **convolutional autoencoder** keeps it. Its encoder uses learned local filters and stride to reduce spatial resolution; its decoder expands the code back into an image.

Here is one shape ledger. The example is deliberately small enough to train during the book build:

Stage	Shape for one example	Role
image	$1 \times 28 \times 28$	784 observed pixel values
stride-2 convolution	$8 \times 14 \times 14$	local features, half resolution
stride-2 convolution	$16 \times 7 \times 7$	broader local features
learned code	16	a 49:1 value-count bottleneck
linear expansion	$16 \times 7 \times 7$	seed the spatial decoder
transposed convolutions	$8 \times 14 \times 14 \rightarrow 1 \times 28 \times 28$	reconstruct resolution

The expanding operator is called a **transposed convolution** because it is the transpose, or adjoint, of the linear map implemented by a convolution. If $\mathcal{C}_{\mathbf{W}}$ denotes convolution with fixed weights, then

$$\langle \mathcal{C}_{\mathbf{W}}(\mathbf{x}), \mathbf{y} \rangle = \langle \mathbf{x}, \mathcal{C}_{\mathbf{W}}^{\top}(\mathbf{y}) \rangle.$$

That equation says **adjoint**, not **inverse**. Stride may discard information, and learned filters need not be invertible. A transposed convolution provides the right learnable shape-changing map for the decoder; it does not mechanically undo the encoder.

We can now make “same architecture, different training” measurable. Both models below use exactly the ledger above. One sees clean images and reconstructs clean images. The other receives Gaussian-corrupted images but is scored against the clean targets. The development file is split into 900 fit and 300 validation examples; the committed 600-example shared benchmark is evaluated only after validation has selected each checkpoint. Earlier chapters already use that benchmark, so this is an endpoint reuse, not a newly sealed test. Five seeds repeat training. Labels never enter the experiment.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable helpers: `ConvolutionalAutoencoder`, `reconstruction_mse`, and `fit_convolutional_autoencoder`.
3. Compare clean and denoising input–target contracts with the same convolutional autoencoder.
4. Report or visualize the measured result.

CODE

```
# [1]
torch.set_default_dtype(torch.float32)
fashion_development = torch.load("../data/fashion-train.pt")
fashion_holdout = torch.load("../data/fashion-test.pt")
development_images = fashion_development["X"].float().unsqueeze(1) / 255.0
holdout_images = fashion_holdout["X"].float().unsqueeze(1) / 255.0

split = torch.randperm(
    len(development_images), generator=torch.Generator().manual_seed(6050)
)
fit_indices, validation_indices = split[:900], split[900:]
fit_images = development_images[fit_indices]
```



```
validation_images = development_images[validation_indices]

noise_sd = 0.35
validation_noise = torch.randn(
    validation_images.shape, generator=torch.Generator().manual_seed(70_050)
)
holdout_noise = torch.randn(
    holdout_images.shape, generator=torch.Generator().manual_seed(70_051)
)
noisy_validation = (validation_images + noise_sd * validation_noise).clamp(0.0, 1.0)
noisy_holdout = (holdout_images + noise_sd * holdout_noise).clamp(0.0, 1.0)

# [2]
class ConvolutionalAutoencoder(nn.Module):
    def __init__(self, latent_dim: int = 16) -> None:
        super().__init__()
        self.encoder_map = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
        )
        self.to_code = nn.Linear(16 * 7 * 7, latent_dim)
        self.from_code = nn.Linear(latent_dim, 16 * 7 * 7)
        self.decoder_map = nn.Sequential(
            nn.ConvTranspose2d(16, 8, kernel_size=4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(8, 1, kernel_size=4, stride=2, padding=1),
            nn.Sigmoid(),
        )

    def forward(self, x: Tensor) -> tuple[Tensor, Tensor]:
        z = self.to_code(self.encoder_map(x).flatten(1))
        features = F.relu(self.from_code(z)).reshape(-1, 16, 7, 7)
        return self.decoder_map(features), z

@torch.no_grad()
def reconstruction_mse(
    model: nn.Module, inputs: Tensor, targets: Tensor
) -> float:
    model.eval()
    reconstruction, _ = model(inputs)
    return F.mse_loss(reconstruction, targets).item()

def fit_convolutional_autoencoder(
    seed: int, denoising: bool, epochs: int = 30
```

```
) -> tuple[float, float, float, ConvolutionalAutoencoder]:
    torch.manual_seed(seed)
    model = ConvolutionalAutoencoder()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.003)
    order_generator = torch.Generator().manual_seed(seed + 100_000)
    noise_generator = torch.Generator().manual_seed(seed + 200_000)
    best_validation = float("inf")
    best_state = deepcopy(model.state_dict())

    for _ in range(epochs):
        model.train()
        order = torch.randperm(len(fit_images), generator=order_generator)
        for indices in order.split(100):
            targets = fit_images[indices]
            if denoising:
                noise = torch.randn(targets.shape, generator=noise_generator)
                inputs = (targets + noise_sd * noise).clamp(0.0, 1.0)
            else:
                inputs = targets
            reconstruction, _ = model(inputs)
            loss = F.mse_loss(reconstruction, targets)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        validation_inputs = noisy_validation if denoising else validation_images
        validation_loss = reconstruction_mse(
            model, validation_inputs, validation_images
        )
        if validation_loss < best_validation:
            best_validation = validation_loss
            best_state = deepcopy(model.state_dict())

    model.load_state_dict(best_state)
    clean_mse = reconstruction_mse(model, holdout_images, holdout_images)
    noisy_mse = reconstruction_mse(model, noisy_holdout, holdout_images)
    return best_validation, clean_mse, noisy_mse, model

conv_rows: dict[bool, Tensor] = {}
conv_models: dict[bool, list[ConvolutionalAutoencoder]] = {}
# [3]
for denoising in (False, True):
    fitted = [
        fit_convolutional_autoencoder(seed, denoising)
        for seed in range(6050, 6055)
    ]
    conv_rows[denoising] = torch.tensor([row[:3] for row in fitted])
    conv_models[denoising] = [row[3] for row in fitted]
```

```
plain_model = conv_models[False][0]
denoising_model = conv_models[True][0]
example_indices = [7, 22]
with torch.no_grad():
    plain_noisy_reconstruction, _ = plain_model(noisy_holdout[example_indices])
    denoised_reconstruction, _ = denoising_model(noisy_holdout[example_indices])

# [4]
# A separate operator check: transposed convolution is the convolution's adjoint.
probe_x = torch.randn(2, 1, 13, 13, dtype=torch.float64)
probe_weight = torch.randn(3, 1, 3, 3, dtype=torch.float64)
probe_y = torch.randn(2, 3, 7, 7, dtype=torch.float64)
conv_x = F.conv2d(probe_x, probe_weight, stride=2, padding=1)
transpose_y = F.conv_transpose2d(
    probe_y, probe_weight, stride=2, padding=1
)
left_inner_product = (conv_x * probe_y).sum()
right_inner_product = (probe_x * transpose_y).sum()
adjoint_relative_gap = (
    (left_inner_product - right_inner_product).abs()
    / torch.maximum(left_inner_product.abs(), right_inner_product.abs())
)

print("arm      validation  clean test  noisy-input test")
for denoising, label in [(False, "plain"), (True, "denoising")]:
    table = conv_rows[denoising]
    print(label, "mean", *(f"{value:.6f}" for value in table.mean(0).tolist()))
    print(label, "SD  ", *(f"{value:.6f}" for value in table.std(0).tolist()))
print("corrupted-input test MSE", f"{F.mse_loss(noisy_holdout, holdout_images):.6f}")
print("transposed-convolution adjoint relative gap", f"{adjoint_relative_gap:.2e}")
```

```
arm      validation  clean test  noisy-input test
plain mean 0.022851 0.020701 0.033263
plain SD   0.000719 0.000565 0.003642
denoising mean 0.026020 0.026294 0.023810
denoising SD   0.000858 0.001445 0.000775
corrupted-input test MSE 0.068601
transposed-convolution adjoint relative gap 1.15e-16
```

The denoising arm wins only the question it was trained to answer. It restores this specified corruption better, while the clean-trained arm reconstructs clean inputs better. Calling either model “the better autoencoder” would erase the experimental contract. The important lesson here is narrow: changing the corruption and target changes the function the same architecture is rewarded for learning.

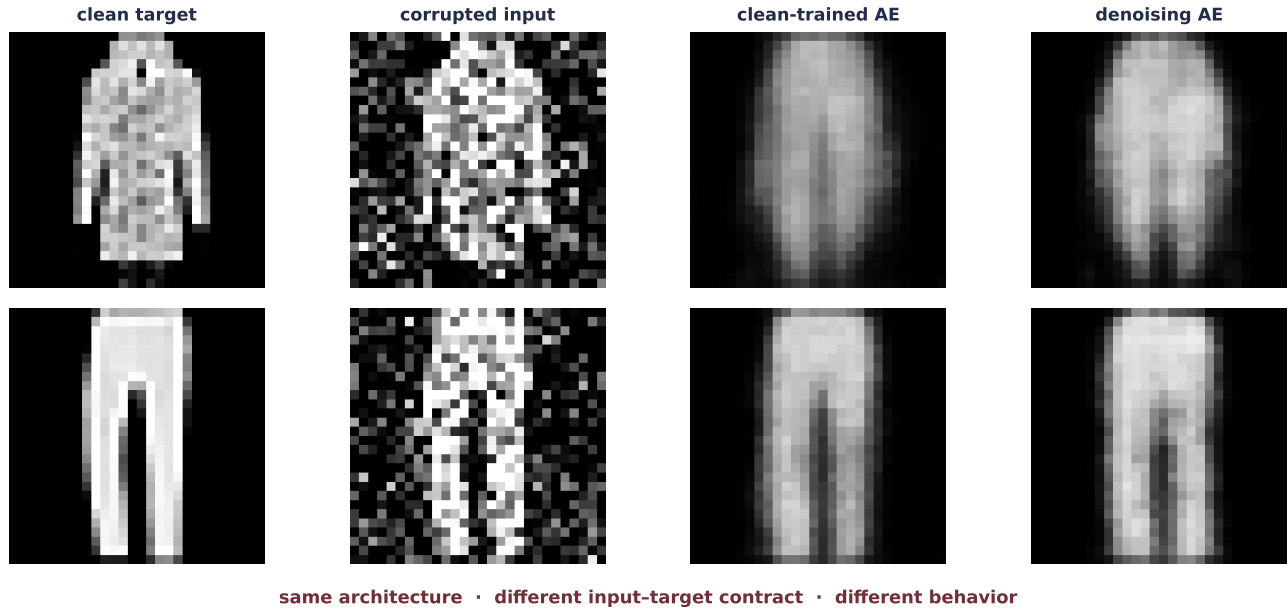


Figure AE.4: The input–target contract determines what the same convolutional autoencoder learns to preserve. The denoising arm gives up some clean reconstruction accuracy to improve on the specified corruption. This CPU Fashion-MNIST-subset study isolates a contract change; it is not a universal ranking of denoisers.

Why a one-shot encoder cannot be the whole sequence model

The detour has earned a useful machine—an encoder can learn a code; a decoder can be trained to reconstruct, denoise, or translate from it. Why not encode an entire sentence into one k -dimensional vector and decode the translation?

For short, bounded objects, that design may work. The trouble is the contract:

- every input must be available before the one-shot encoder can finish;
- the code has the same width for four tokens and four thousand;
- every distinction in content and order must survive the same handoff;
- the decoder cannot revisit an earlier input detail that the code discarded.

Making k larger postpones the argument; it does not remove the fixed maximum or give the computation a streaming rule. The next chapter changes the shape of the problem. It shares one update across time,

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}, \mathbf{x}_t),$$

so the model can accept one element at a time and stop whenever the sequence stops. Recurrence still has a finite state—later chapters will expose that bottleneck—but it turns a fixed-input network into a variable-length process. The autoencoder did not fail us. It made the missing operation visible.

Okay, so — PCA became a network, then the code became a bottleneck

i Two different limitations

“A one-shot encoder is not a variable-length process” asks how memory should update as evidence arrives. “A code is not yet a probability distribution” asks how a genuinely new sample could begin. Do not collapse them: they are different contracts and require different machinery.

i Check yourself

Reconstruct the bridge without looking back.

- Under which restrictions does a linear autoencoder recover the principal subspace?
- What changes when the encoder and decoder become nonlinear?
- Why does successful reconstruction not define a sampling distribution or a variable-length update rule?

Okay, so — PCA became a network, then the code became a bottleneck

1. **An autoencoder learns an input–code–reconstruction contract.** The target says what information the code is rewarded for preserving.
2. **PCA is the linear case under stated conditions.** Center the data, use an undercomplete linear map and squared loss, and compare subspaces or projectors—not arbitrary latent coordinates.
3. **Gradient descent makes the projection learnable.** Add depth and nonlinearity and the reconstruction map can follow curved structure; it gains capacity, not a semantic guarantee.
4. **Convolution preserves the spatial prior.** Transposed convolution is the adjoint shape-changing operator used by the decoder, not an inverse of the encoder.
5. **Same architecture, different contracts, different behavior.** Clean reconstruction, denoising, and translation pair inputs and targets in distinct ways.
6. **A one-shot handoff plants two questions.** Variable-length evidence asks for a step-by-step update. Unsupported latent sampling waits until the book has earned a probabilistic starting rule.

Sources and further reading

- [Baldi and Hornik, *Neural Networks and Principal Component Analysis*](#): the qualified principal-subspace result for linear autoencoders.
- [Vincent et al., *Extracting and Composing Robust Features with Denoising Autoencoders*](#): noisy-input, clean-target representation learning.
- [PyTorch, *ConvTranspose2d*](#): the framework operator and output-shape conventions used in the independently derived implementation.

Exercises

1. **(Pencil.) Untie the linear maps.** Replace $\mathbf{W}\mathbf{W}^\top$ with an encoder $\mathbf{W}_e$ and decoder $\mathbf{W}_d$. Construct an invertible change of latent coordinates that changes both matrices but leaves their product unchanged.

2. **(Audit.) Audit the bend.** Extend the planted curve’s test interval beyond the range used for training. Compare interpolation with extrapolation and state which claim the original interleaving split earned.
3. **(Code.) Change the corruption.** Train the convolutional autoencoder with masking noise instead of Gaussian noise. Evaluate both corruptions without retuning and explain why “denoising ability” is not one scalar property.
4. **(Pencil.) Prove the adjoint check.** Write a one-channel convolution as an explicit matrix for a tiny input. Verify that applying its matrix transpose matches a transposed convolution with the corresponding stride and padding.
5. **(Audit.) Stress the one-shot code.** Design two sequence families that share the same final tokens but require remembering increasingly old information. Predict how a one-shot encoder and a recurrent state will fail differently as length grows.

Part III.

Sequences

10. Sequences and Recurrence

The autoencoder interlude promised you a **one-shot encoder is not a variable-length process**. We began with PCA’s closed-form projection, made its encoder and decoder learnable, and added nonlinearity so the reconstruction could bend. For a bounded object such as a 28×28 image, that is a coherent contract: see the whole object, compress it to $\mathbf{z} \in \mathbb{R}^k$, then decode.

Now let the object keep arriving. A four-token message and a four-thousand-token report are both squeezed through the same k numbers; the encoder cannot finish before the last item appears; the decoder cannot revisit evidence discarded from the code. Increasing k postpones the pressure, but it does not tell a fixed-input network how to accept an unknown number of pieces.

This part is about data that refuses the fixed-size contract. Text, audio, sensor streams, stock prices—they come one piece at a time, stop whenever they please, and carry meaning in the *order* of the pieces. We need a computation that can update as the evidence arrives. Recurrence asks us to make the carried summary learnable: rather than compressing once, the model learns how to revise its memory after each new piece.

Order deserves a moment, because the book has been here before with the opposite verdict. Chapter 6 shuffled every pixel with one fixed permutation and retrained the MLP to the same accuracy: because that coordinate map was invertible, the first-layer weights could absorb it. The trained MLP was still position-specific; it was not a bag model. Now shuffle the words of *this sentence* and read it back. For sequences, no single compensating remap preserves arbitrary order—order is most of the signal. We need models whose computation represents it explicitly.

10.1. Stretching the old toolkit

Before building anything new, the book’s habit: try what we own.

A window. Predict the next element from the last k : slide a width- k window along the sequence and feed each snapshot to an MLP. This should feel familiar twice over: it is autoregression, and the *sliding* is Chapter 7’s one-dimensional convolution — locality, transplanted from space to time. Windows are genuinely useful (they power real forecasting systems), but they inherit locality’s blind spot: anything older than k steps ago does not exist. Widen k and the parameter count grows with it, yet no finite window survives the sentence “The company whose first-day price we recorded back in January finally ...”. Context has no fixed radius.

A function per timestep. Then let the model change over time: learn f_1 for step one, f_2 for step two, each passing a summary forward. We first build this strawman carefully, because its failure names the cure. The parameter count now grows with sequence *length*; a length-500 essay needs 500 learned functions; and the model is **non-stationary** — it insists the rules of language at position 17 differ from the rules at position 18, when the whole point of language is that they do not.

You have watched this book face that exact configuration twice. A template for every image position was wasteful and brittle $\rightarrow$ share the kernel across space (Chapter 7). A detector per location was the

wrong prior $\rightarrow$ share it (Chapter 8). The move is the same here, and it is the chapter’s one big idea: **share the function across time**,

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t), \quad (10.1)$$

one learned rule f , applied at every step, carrying a running summary $\mathbf{h}_t$ — the **hidden state** — from each step to the next. This is the book’s third weight sharing: across examples (every chapter since the first), across space (Part II), and now across *time*. Stationarity is to sequences what translation equivariance was to images: the same pattern means the same thing whenever it occurs.

i The hidden state is a running memory — and a Markov claim

$\mathbf{h}_t$ is whatever f chooses to remember about everything seen so far, compressed into one fixed-size vector. Feeding it back in makes a claim with a classical name, the **Markov property**: $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ together contain everything needed for the next step. Nothing else from the past gets a second look. That claim buys streaming (process tokens as they arrive), any sequence length (the loop doesn’t care), and — later in this chapter — the license to train on chopped-up chunks. Its price is that the memory is *finite*, and this part of the book ends when we finally refuse to pay that price (Chapter 12).

10.2. A network with a loop

The simplest choice of f is Chapter 3’s recipe with the state concatenated in: a linear map of $(\mathbf{h}_{t-1}, \mathbf{x}_t)$, squashed by a tanh:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh} \mathbf{h}_{t-1} + \mathbf{W}_{xh} \mathbf{x}_t + \mathbf{b}_h), \quad \mathbf{o}_t = \mathbf{W}_{hq} \mathbf{h}_t + \mathbf{b}_q, \quad (10.2)$$

the **vanilla recurrent neural network**. Three weight matrices total, reused at every step, regardless of whether the sequence has five tokens or five thousand. The readout $\mathbf{o}_t$ produces logits — Chapter 2’s machinery — and you can collect one at every step (predicting each next character, labeling each word) or only at the end (classifying the whole sequence); both designs appear below.

Per the book’s habit, build the loop by hand, then verify against the framework:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Implement the recurrence, written as the loop it is.
 3. Report or visualize the measured result.
-

CODE

```

import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

# [1]
torch.manual_seed(6050)
V, H, T, B = 8, 16, 12, 4          # vocab, hidden, time, batch
rnn = nn.RNN(V, H, batch_first=True)
x = torch.randn(B, T, V)
out_ref, _ = rnn(x)

W_xh, W_hh = rnn.weight_ih_l0, rnn.weight_hh_l0
b = rnn.bias_ih_l0 + rnn.bias_hh_l0
h = torch.zeros(B, H)              # h_0 = 0, the standard start
outs = []
# [2]
for t in range(T):                  # the loop IS the architecture
    h = torch.tanh(x[:, t] @ W_xh.T + h @ W_hh.T + b)
    outs.append(h)
manual = torch.stack(outs, 1)
# [3]
print(f"manual loop vs. nn.RNN: max |diff| = {(manual - out_ref).abs().max():.1e}")

```

```
manual loop vs. nn.RNN: max |diff| = 1.2e-07
```

Notice what the shapes say. `nn.RNN` eats (batch, time, features) — a third axis has joined Chapter 8’s NCHW discipline — and the loop runs over that middle axis carrying `h` along. Unroll the loop on paper and the RNN is an ordinary feedforward network that happens to be T layers deep with every layer’s weights tied. Hold that thought; it is about to matter enormously.

10.3. Blame through time

Tied depth means Chapter 5’s rules apply verbatim. To train on a sequence, run the loop forward, collect losses at the readouts, and backpropagate through the *unrolled* network, a procedure called **backpropagation through time** (BPTT). Blame that lands on $\mathbf{h}_T$ must travel back through every application of Equation 10.2 to reach an early input. Define $\mathbf{D}_t = \text{diag}(\tanh'(\cdot))$ at step t . The forward Jacobian for one hop is $\mathbf{D}_t \mathbf{W}_{hh}$, so the full Jacobian is

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_1} = (\mathbf{D}_T \mathbf{W}_{hh}) \cdots (\mathbf{D}_2 \mathbf{W}_{hh}), \quad \mathbf{g}_{t-1} = \mathbf{W}_{hh}^\top \mathbf{D}_t \mathbf{g}_t. \quad (10.3)$$

A product of T near-identical matrices. The second expression shows the reverse-mode order explicitly: the derivative gate acts on incoming blame, then $\mathbf{W}_{hh}^\top$ carries it to the previous state. Chapter 5 taught you to fear this shape. If the factors shrink the signal, the product dies exponentially; if they grow it, it explodes exponentially. Since $\tanh' \leq 1$ and $\mathbf{W}_{hh}$ repeats at every step, keeping every relevant direction

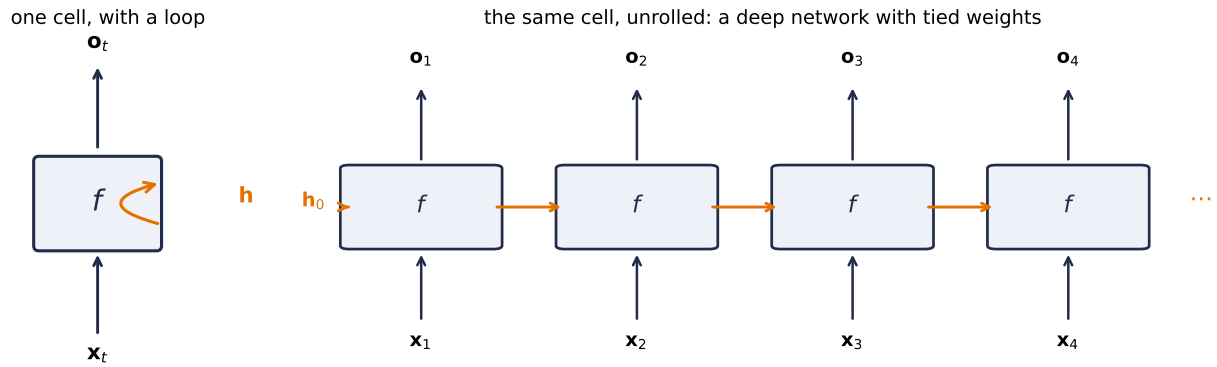


Figure 10.1.: Two drawings of the same machine. Left: one cell whose hidden state loops back into itself. Right: the loop unrolled — a chain of copies of the *same* cell, each consuming one input and passing the state along. Read vertically it is Chapter 3’s network; read horizontally it is a network as deep as the sequence is long, with every layer’s weights tied. Both halves of that sentence are about to matter.

near unit gain over a long sequence is a fragile special case. Depth was measured in layers there; here it is measured in *time*, and a paragraph of text is a hundred layers deep. Watch the collapse, in the spirit of Chapter 5’s depth experiment:

PLAN

1. Define the reusable `grad_at_first_input` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Gradient reaching the first input vs. lag.
-

CODE

```
# [1]
def grad_at_first_input(rec: nn.Module, lags: list[int],
                        vocab: int = 4) -> list[float]:
    norms = []
    for T in lags:
        generator = torch.Generator().manual_seed(100 + T)
        x = torch.randn(8, T, vocab, generator=generator, requires_grad=True)
        o, _ = rec(x)
        o[:, -1].sum().backward()
        norms.append(x.grad[:, 0].norm().item())    # blame on input #1
    return norms

# [2]
lags = [1, 5, 10, 20, 40, 60]
torch.manual_seed(0)
vanilla = nn.RNN(4, 32, batch_first=True)
```

```

g_rnn = grad_at_first_input(vanilla, lags)
# [3]
print("lag:      " + " ".join(f"{l:>7d}" for l in lags))
print("|grad|:    " + " ".join(f"{g:.1e}" for g in g_rnn))

plt.figure(figsize=(6.2, 3.2))
plt.semilogy(lags, g_rnn, "o-", color="#E57200", ms=5, label="vanilla RNN")
plt.xlabel("lag $T$ between output and first input")
plt.ylabel(r"$\|\partial L / \partial \mathbf{x}_1\|$")
plt.legend(); plt.tight_layout(); plt.show()

```

```

lag:      1      5      10      20      40      60
|grad|:  2.9e+00  6.8e-01  2.6e-02  7.2e-05  3.4e-10  1.1e-15

```

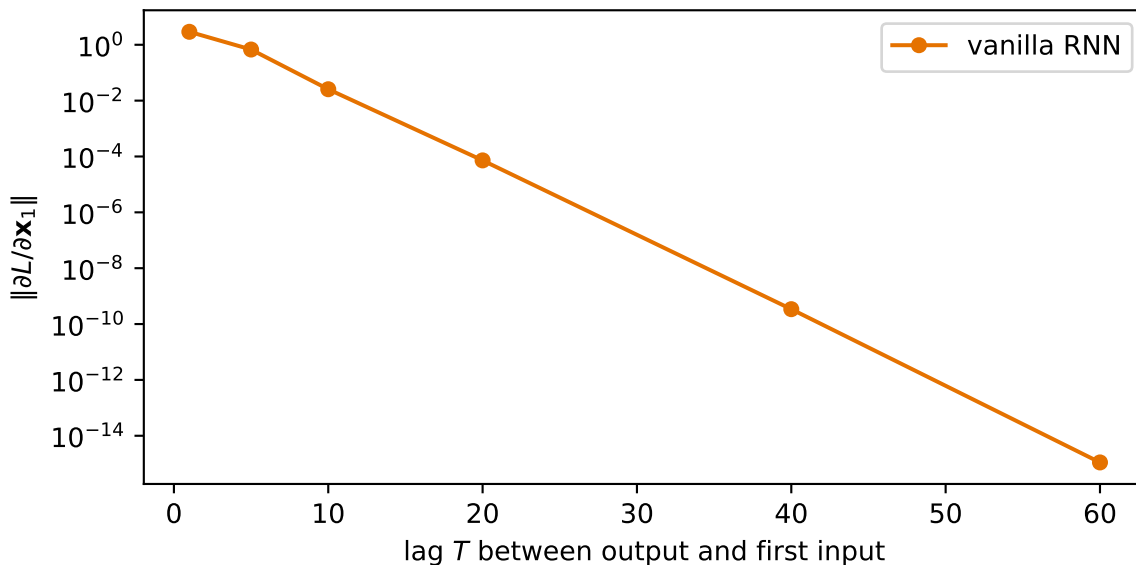


Figure 10.2.: How much a freshly initialized vanilla RNN’s output can blame its first input, as the lag between them grows. Fifteen orders of magnitude gone by sixty steps: the same exponential decay Chapter 5 measured down a deep network, now measured across time — because an unrolled RNN *is* a deep network with tied weights.

By lag 60 the gradient reaching the first input is 10^{-15} : at initialization, whatever happened at the start supplies almost no usable learning signal. That makes success fragile rather than mathematically impossible — a later experiment catches one lucky seed. The mirror-image failure, explosion, is just as real — one large spike in Equation 10.3’s product can make an update destabilize training.

⚠ Gradient clipping is a seatbelt, not a cure

Practice handles explosion with **gradient clipping**: if the gradient’s norm exceeds a threshold, rescale it down to the threshold before stepping (`nn.utils.clip_grad_norm_` — it appears in every training loop below). This tames the spikes so training survives, at the cost of damping legitimately large steps — and it does *nothing* for vanishing. You cannot clip your way to a longer

memory.

10.3.1. Training with a finite horizon

BPTT has a second, blunter problem: unrolling a book-length sequence costs book-length memory and compute per step. The standard fix leans on the Markov claim from the callout above. In **truncated BPTT**, contiguous chunks carry the recurrent state forward but detach it at each boundary. The values cross the boundary; the gradient graph does not. This bounds memory while preserving running context.

The final experiment below uses a simpler finite-horizon recipe: it samples random 100-character windows and starts each from zero state. That is **fixed-window training**, not truncated BPTT, because no state is carried between windows. It is cheap and appropriate for this small demonstration, but the model can neither see nor learn dependencies beyond 100 characters. In production character models, contiguous chunks of roughly 20–50 steps with carried-and-detached state are common. In either recipe, clipping handles spikes; neither creates gradients beyond the truncation horizon.

10.4. Engineering a memory: the LSTM

So the vanilla RNN’s memory dies of repeated multiplication. A useful way to stage the fix is as a design-by-constraints exercise, and it is worth walking because you already own the key move. We want a memory pathway that is:

1. **Preserved by default.** Repeated *multiplication* killed the signal $\rightarrow$ make the carried state **additive**: $\mathbf{c}_t = \mathbf{c}_{t-1} + \Delta_t$. A conveyor belt: information rides along untouched unless something deliberately intervenes. You met this exact maneuver one chapter ago — [Chapter 9](#)’s residual connection, $H(x) = F(x) + x$, whose Jacobian’s identity term let gradients cross forty layers. Same trick, laid across time.
2. **Forgettable on command.** Memory is finite; some of it should expire. Install a valve: $\mathbf{f}_t \in (0, 1)$ multiplying $\mathbf{c}_{t-1}$.
3. **Writable on command.** New information enters through a second valve $\mathbf{i}_t$ scaling a candidate $\tilde{\mathbf{c}}_t$.
4. **Readable on command.** Expose only what the current step needs, through a third valve $\mathbf{o}_t$.

Each valve is a small learned layer squashed by a sigmoid — Chapter 2’s machine, reinterpreted: a number in $(0, 1)$ read as *what fraction passes*. The candidate content is shaped by $\tanh$, keeping it bounded and centered. Assembled, this is the **Long Short-Term Memory** cell (Hochreiter & Schmidhuber, 1997):

$$\begin{aligned}
 \mathbf{f}_t &= \sigma(\mathbf{W}_f [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) & \mathbf{i}_t &= \sigma(\mathbf{W}_i [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \\
 \tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) & \mathbf{o}_t &= \sigma(\mathbf{W}_o [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \\
 \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t & \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t).
 \end{aligned} \tag{10.4}$$

Two states now travel: the cell state $\mathbf{c}_t$ (long-term memory, the conveyor belt) and the hidden state $\mathbf{h}_t$ (working memory, playing the vanilla RNN’s old role). The line that changes everything is the $\mathbf{c}_t$ update. Read its gradient along the cell chain:

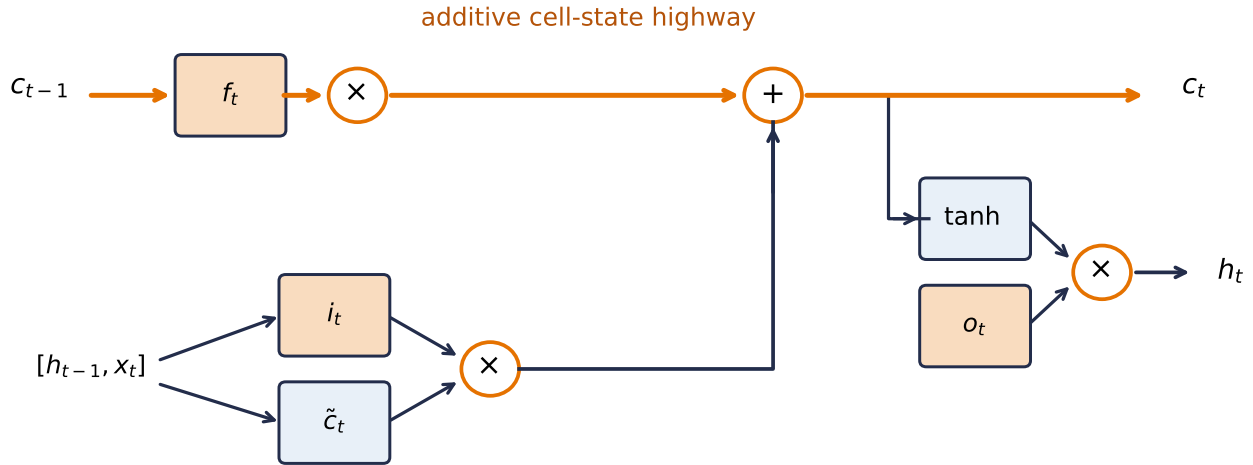


Figure 10.3.: The LSTM cell as a conveyor belt with valves. The forget gate f_t controls how much old cell state continues; the input gate i_t controls how much candidate content is added; the output gate o_t controls what becomes visible as h_t . The upper additive route is the long-memory highway.

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} \approx \mathbf{f}_t \quad \Rightarrow \quad \frac{\partial L}{\partial \mathbf{c}_{t-k}} \approx \left(\prod_{j=t-k+1}^t \mathbf{f}_j \right) \odot \frac{\partial L}{\partial \mathbf{c}_t}. \quad (10.5)$$

Still a product — but of *learned, data-dependent gates*, not a fixed weight matrix. Where Chapter 9’s residual highway kept an unconditional identity lane open, the LSTM’s lane has a valve the network controls: hold $\mathbf{f}_j$ near 1 and gradients ride the conveyor belt across dozens of steps; drop it toward 0 and the memory (deliberately) expires. The practical corollary: *initialize the forget-gate bias positive* (say +1), so the cell starts life remembering by default. PyTorch stores separate input-hidden and hidden-hidden bias vectors, then adds them. The code sets their **sum** to +1; setting both slices to +1 would silently create an effective bias of +2. Watch what that buys, on the same axes as before:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Measure the LSTM gradient norm across lag.
-

CODE

```
# [1]
torch.manual_seed(0)
lstm = nn.LSTM(4, 32, batch_first=True)
with torch.no_grad():
    n = lstm.bias_ih_l0.shape[0] // 4
```

```

lstm.bias_ih_l0[n:2 * n].fill_(1.0)
lstm.bias_hh_l0[n:2 * n].zero_()           # the two bias slices sum to +1
# [2]
g_lstm = grad_at_first_input(lstm, lags)

```

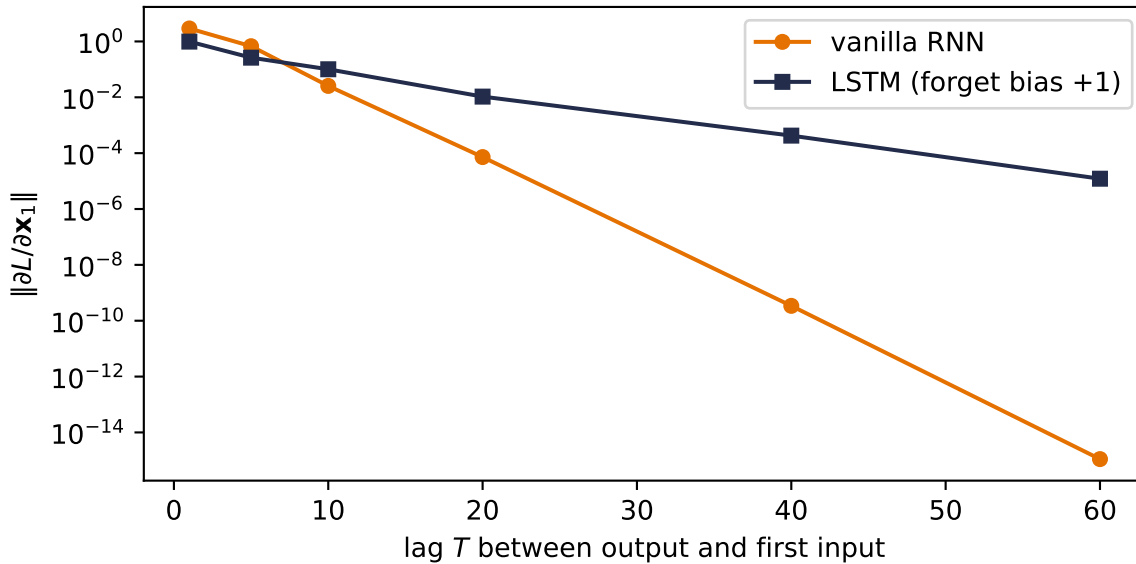


Figure 10.4.: The gradient-vs-lag experiment, rerun with an LSTM whose effective forget-gate bias is initialized to +1. The vanilla RNN’s derivative signal dies exponentially; the LSTM’s rides the cell-state conveyor belt and arrives at lag 60 attenuated but alive, about ten orders of magnitude stronger. Chapter 9 built this highway across layers; the LSTM builds it across time, with a learned valve on the on-ramp.

10.4.1. The memory test

Gradients at initialization are promissory notes; training is where they get cashed. The decisive task is a two-company story: Company A’s prices open at 0, Company B’s at 1, then both trade identically for days — and the correct day-5 prediction is whatever day 1 said. Everything hinges on carrying one early bit across a long stretch of identical noise. Here is that story as an executable task: sequences of 80 random tokens, where the label to predict at the end is simply *the first token*. Chance is 25%. Three contenders, three seeds each: the vanilla RNN, the LSTM as PyTorch hands it to you, and the LSTM with the remember-by-default initialization. Predict the three rows before running — in particular, does the *architecture* alone earn its keep?

PLAN

1. Define the reusable helpers: `make_recall`, `SeqClassifier`, and `train_recall`.
2. Prepare the inputs and fixed settings for the example.
3. Recall-the-first-token at lag 80, three configurations.

CODE

```
# [1]
def make_recall(n: int, T: int, vocab: int = 4,
               seed: int = 0) -> tuple[torch.Tensor, torch.Tensor]:
    g = torch.Generator().manual_seed(seed)
    X = torch.randint(0, vocab, (n, T), generator=g)
    return F.one_hot(X, vocab).float(), X[:, 0]      # label = first token

class SeqClassifier(nn.Module):
    def __init__(self, cell: str, vocab=4, hidden=32, forget_bias=None):
        super().__init__()
        self.rec = {"rnn": nn.RNN, "lstm": nn.LSTM}[cell](vocab, hidden,
                                                         batch_first=True)

        if forget_bias is not None:
            with torch.no_grad():
                n = self.rec.bias_ih_l0.shape[0] // 4
                self.rec.bias_ih_l0[n:2 * n].fill_(forget_bias)
                self.rec.bias_hh_l0[n:2 * n].zero_()
        self.out = nn.Linear(hidden, vocab)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        o, _ = self.rec(x)
        return self.out(o[:, -1])                  # read out at the END only

def train_recall(cell: str, T: int, seed: int, forget_bias: float | None = None,
               epochs: int = 80) -> float:
    torch.manual_seed(seed)
    net = SeqClassifier(cell, forget_bias=forget_bias)
    X_tr, y_tr = make_recall(2000, T, seed=1)
    X_te, y_te = make_recall(500, T, seed=2)
    opt = torch.optim.Adam(net.parameters(), lr=3e-3)
    for _ in range(epochs):
        perm = torch.randperm(len(X_tr))
        for i in range(0, len(X_tr), 128):
            idx = perm[i:i + 128]
            loss = F.cross_entropy(net(X_tr[idx]), y_tr[idx])
            opt.zero_grad(); loss.backward()
            nn.utils.clip_grad_norm_(net.parameters(), 1.0)
            opt.step()
    with torch.no_grad():
        return (net(X_te).argmax(1) == y_te).float().mean().item(), net
```



```

# [2]
T = 80
configs = [("vanilla RNN", "rnn", None),
            ("LSTM, default init", "lstm", None),
            ("LSTM, forget bias +1", "lstm", 1.0)]
probe_nets: dict[str, nn.Module] = {} # kept for the diagnostic below
print(f"recall accuracy at lag {T} (chance 25%), seeds 0 / 1 / 6050:")
# [3]
for label, cell, fb in configs:
    results = [train_recall(cell, T, s, fb) for s in [0, 1, 6050]]
    accs = [a for a, _ in results]
    probe_nets[label] = results[-1][1] # seed 6050's model
    print(f" {label:22s} " + " ".join(f"{a:.0%}" for a in accs))

```

recall accuracy at lag 80 (chance 25%), seeds 0 / 1 / 6050:

vanilla RNN	25%	100%	25%
LSTM, default init	24%	26%	26%
LSTM, forget bias +1	100%	100%	100%

Three rows, one lesson each. The vanilla RNN is a *lottery*: two seeds sit at chance, one cracks the task. Figure 10.2 says the teaching gradient is astronomically attenuated, not exactly zero — so once in a while optimization stumbles onto a solution anyway, and you cannot build a system on once-in-a-while. The default-initialized LSTM loses on *every* seed — worth a long pause, because it says the architecture alone is not the cure. A fresh LSTM's forget gates hover near $\sigma(0) = \frac{1}{2}$, and $0.5^{80} \approx 10^{-24}$: a half-closed valve, compounded eighty times, is as fatal as no valve. The third row flips the single bias that opens the valve, and the task is solved outright, every seed. The highway was built by the architecture; *initialization opened it* — the same lesson Chapter 5 taught about signal scales at birth, now with a memory attached. Appendix C separates this mathematical attenuation from the later question of whether a chosen dtype can still represent the resulting value or update. And the deliverable is not raw capability but **reliability**: what the gates buy is that memory stops being a lottery. (For shorter stretches the drama disappears: at lag 20 all three rows solve the task under this budget. The gates earn their keep precisely where Equation 10.3 says the vanilla cell must fail.)

10.4.2. Watching the valves

The table's story makes a checkable claim about mechanism: if the forget-bias model solves the task by *holding its valve open* across the eighty-step gap, the gates themselves should show it. The falsifying control is already trained: the default-initialized LSTM has the identical architecture, saw the identical data, and sits at chance. Prediction, per Equation 10.5: the solver's mean forget gate should hold clearly above the half-open point, step after step; the failed twin's should sag near $\sigma(0) = \frac{1}{2}$.

PLAN

1. Define the reusable `forget_gate_trace` helper.
2. Prepare the inputs and fixed settings for the example.
3. Replay the trained LSTMs and read their forget gates.

CODE

```
# [1]
@torch.no_grad()
def forget_gate_trace(net: SeqClassifier, T: int = 80) -> list[float]:
    X_probe, _ = make_recall(64, T, seed=3)          # fresh probe sequences
    lstm = net.rec
    W_ih, W_hh = lstm.weight_ih_l0, lstm.weight_hh_l0
    b = lstm.bias_ih_l0 + lstm.bias_hh_l0
    hidden = W_hh.shape[1]
    h = torch.zeros(64, hidden)
    c = torch.zeros(64, hidden)
    means = []
    for t in range(T):                               # replayed by hand
        z = X_probe[:, t] @ W_ih.T + h @ W_hh.T + b
        i_gate, f_gate, g_cand, o_gate = z.chunk(4, dim=1)
        f = torch.sigmoid(f_gate)
        c = f * c + torch.sigmoid(i_gate) * torch.tanh(g_cand)
        h = torch.sigmoid(o_gate) * torch.tanh(c)
        means.append(f.mean().item())
    out_ref, _ = lstm(X_probe)                        # replay must match the module
    assert torch.allclose(h, out_ref[:, -1], atol=1e-5)
    return means

# [2]
plt.figure(figsize=(6.2, 3.2))

# [3]
for label, color in [("LSTM, forget bias +1", "#232D4B"),
                     ("LSTM, default init", "#E57200")]:
    plt.plot(range(1, T + 1), forget_gate_trace(probe_nets[label]),
             color=color, lw=2, label=label)
plt.axhline(0.5, ls=":", color="#B8B8A8")
plt.ylim(0, 1.02)
plt.xlabel("timestep $t$")
plt.ylabel("mean forget gate")
plt.legend()
plt.tight_layout(); plt.show()
```

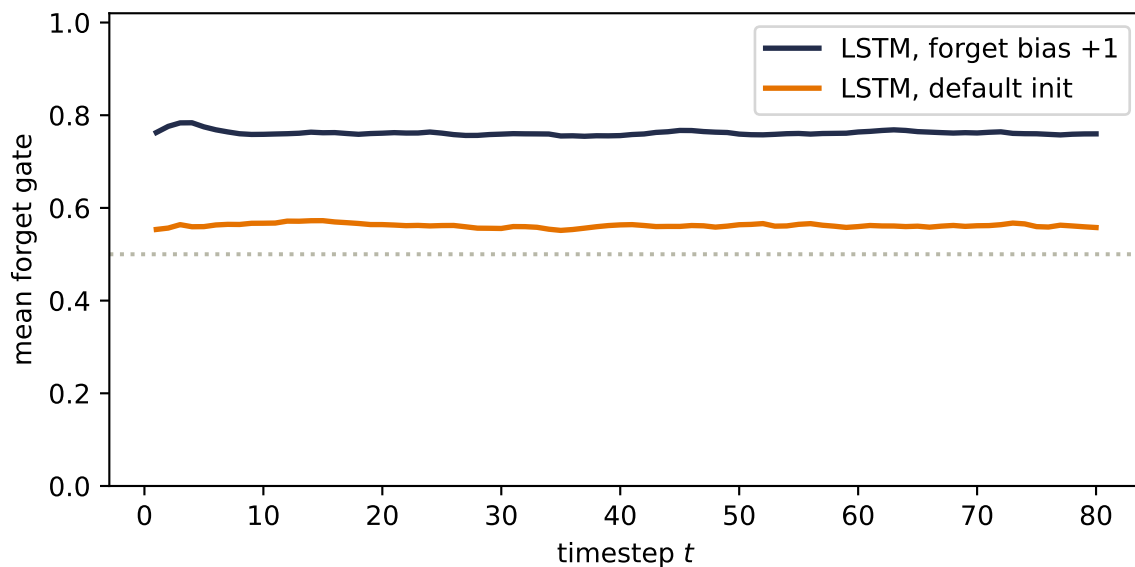


Figure 10.5.: The valve, observed. Mean forget-gate activation per timestep on 64 fresh probe sequences, replayed through the two trained seed-6050 LSTMs from the table above. The task-solving model (navy) holds its gates flat around 0.76 for all eighty steps — the conveyor belt stays open, and the first token survives to the readout. The default-initialized model (orange), same architecture and data, hovers near $\sigma(0) \approx \frac{1}{2}$ at about 0.56: a half-closed valve compounded eighty times, and chance accuracy. One task, one seed each, means over units and probes — a diagnostic of this experiment, not a general theory of trained LSTMs.

The replay is Equation 10.4 executed by hand with the trained weights — the assert guarantees it reproduces the module’s own output — so the curves are the model’s actual gates, not a cartoon. Architecture proposed the highway; initialization opened it; here is the valve position, measured.

10.5. GRU: the streamlined cousin

The **Gated Recurrent Unit** (Cho et al., 2014) folds the same ideas into a smaller box: cell and hidden state merged into one $\mathbf{h}_t$, forget and input valves merged into a single **update gate** $\mathbf{z}_t$ (what you keep you don’t overwrite, and vice versa), plus a **reset gate** $\mathbf{r}_t$ that can blank the past when a fresh start is called for:

$$\begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t]), & \mathbf{r}_t &= \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t]), \\ \tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t]), & \mathbf{h}_t &= \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t. \end{aligned} \quad (10.6)$$

This follows Cho et al.’s original convention: $\mathbf{z}_t$ weights the old state, so it reads as **keep**. Some later references reverse the symbol and let $\mathbf{z}_t$ weight the candidate instead; compare the interpolation equation, not the gate’s name alone.

Two gates instead of three, one state instead of two, comparable performance in most settings. We skip the derivation; with the equations on record and Exercise 3 waiting, you can build one yourself and let it compete on the memory test.

One convention matters before you compare code. Equation 10.6 applies the reset gate to $\mathbf{h}_{t-1}$ **before** the candidate’s hidden-state linear map. PyTorch’s `nn.GRU` uses a reset-after form, $\tanh(W_{in}x + b_{in} + r \odot (W_{hn}h + b_{hn}))$, for implementation efficiency. The gates serve the same purpose, but the two cells are not algebraically identical for the same weights. An exact manual-loop verification must implement PyTorch’s ordering rather than Equation 10.6’s.

i Historical bridge (non-examinable): before contextual states

Word2Vec learned a fixed vector for each word type through local prediction: skip-gram predicts nearby words from a center word, while CBOW predicts the center from its neighborhood. **GloVe** used corpus-wide co-occurrence counts, fitting vector dot products and biases to their logarithmic structure. Both made distributional meaning learnable, but both returned a static lookup: the stored vector for *bank* did not change between a river and a loan.

The recurrent state in this chapter changes that contract. Its h_t depends on the visible prefix, so the representation at a position is contextual even when its input token embedding began as a lookup. Later we will build a different mechanism that lets each position mix information from other visible positions directly. First, the recurrent route deserves to be understood on its own terms.

10.6. The book reads itself

Time to put fixed windows, clipping, gates, and Chapter 2’s softmax over a vocabulary to work on real text. In keeping with a book whose every experiment runs on its own pages, the training corpus is *the nine chapters you have already read*: their prose, stripped of code cells, about 150,000 characters. We commit that code-stripped text as a benchmark snapshot; later copyedits must not silently change the task. The task is the oldest one in language modeling: read a character, predict the next.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Run a character-level LSTM trained on Chapters 1–9.
 3. Report or visualize the measured result.
 4. Define the `CharLSTM` module.
-

CODE

```
import hashlib
from pathlib import Path

# [1]
corpus_path = Path("../data/book-corpus-ch1-9.txt")
text = corpus_path.read_text(encoding="utf-8")
# [2]
assert len(text) == 148_594
assert hashlib.sha256(text.encode("utf-8")).hexdigest() == (
```

```

    "b0fc23a513e37e7bcd78da04a03877345f6ebc850b91dbae260656d9924fb299"
)
chars = sorted(set(text))
stoi = {c: i for i, c in enumerate(chars)}
data = torch.tensor([stoi[c] for c in text])
split = int(0.9 * len(data))
train_data, valid_data = data[:split], data[split:]
# [3]
print(f"corpus: 9 chapters, {len(text):,} characters, vocab {len(chars)}")
print(f"deterministic split: {len(train_data):,} train / {len(valid_data):,} held
    ↪ out")

# [4]
class CharLSTM(nn.Module):
    def __init__(self, vocab: int, hidden: int = 128):
        super().__init__()
        self.vocab = vocab
        self.lstm = nn.LSTM(vocab, hidden, batch_first=True)
        self.out = nn.Linear(hidden, vocab)
    def forward(
        self, x: torch.Tensor,
        state: tuple[torch.Tensor, torch.Tensor] | None = None,
    ) -> tuple[torch.Tensor, tuple[torch.Tensor, torch.Tensor]]:
        o, state = self.lstm(F.one_hot(x, self.vocab).float(), state)
        return self.out(o), state # logits at EVERY step

```

```

corpus: 9 chapters, 148,594 characters, vocab 104
deterministic split: 133,734 train / 14,860 held out

```

The training loop below is the book’s **canonical next-token trainer** — Listing 10.1. Chunk sampling *is* truncated backpropagation through time: fresh random 100-character windows every step, gradients clipped, no state carried across chunk boundaries. The listing lives in the support module as tested source (`code/dlbook/training.py`); what you read here is included from that file, and the cell after it imports and runs the *same artifact* — printed code and executed code cannot drift apart.

Listing 10.1 — the canonical next-token trainer:

PLAN

1. Declare the shared next-token training interface and its protocol controls.
 2. Configure the optimizer and requested step budget.
 3. Draw or reuse chunk starts, then construct shifted input–target pairs.
 4. Predict every next token and average cross-entropy across batch and time.
 5. Backpropagate, clip the gradient, and take one optimizer step.
 6. Retain a sparse learning curve and return the trained artifact.
-

CODE

```

"""Next-token training loop - Chapter 10's listing, importable.

The loop is printed and taught in Chapter 10 (truncated-BPTT chunk sampling,
gradient clipping); later chapters import it and print only their deltas.
"""

import torch
import torch.nn.functional as F
from torch import nn

# [1]
def fit_next_token(
    model: nn.Module,
    data: torch.Tensor,
    *,
    vocab: int,
    context: int = 100,
    batch: int = 64,
    steps: int = 2501,
    lr: float = 2e-3,
    clip: float = 1.0,
    schedule: list[torch.Tensor] | None = None,
    log_every: int = 500,
    log_decimals: int = 2,
) -> tuple[nn.Module, list[tuple[int, float]]]:
    """Train `model` to predict data[t+1] from data[t-context+1 .. t].

    With `schedule=None`, chunk starts are drawn fresh each step (Chapter 10's
    truncated-BPTT sampling, consuming the global RNG exactly as printed there).
    A precomputed `schedule` of start tensors makes the minibatch order an
    explicit, shareable part of the protocol (Chapter 14's paired comparison).
    """

    # [2]
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    curve: list[tuple[int, float]] = []
    n_steps = len(schedule) if schedule is not None else steps
    # [3]
    for step in range(n_steps):
        starts = (
            schedule[step]
            if schedule is not None
            else torch.randint(0, len(data) - context - 1, (batch,))
        )
        xb = torch.stack([data[j : j + context] for j in starts])
        yb = torch.stack([data[j + 1 : j + context + 1] for j in starts])
        # [4]
        logits = model(xb)
        if isinstance(logits, tuple):
            # recurrent models return state

```

```

        logits = logits[0]
    loss = F.cross_entropy(logits.reshape(-1, vocab), yb.reshape(-1))
    # [5]
    opt.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(model.parameters(), clip)
    opt.step()
    # [6]
    if step % log_every == 0:
        curve.append((step, loss.item()))
        print(f"step {step:4d}    loss {loss.item():.{log_decimals}f}")
    return model, curve

```

Listing 10.2 — the fixed-window evaluation protocol (state reset at every window boundary, so train and held-out numbers stay comparable across chapters):

PLAN

1. Declare inference-only fixed-window evaluation.
 2. Switch to evaluation mode and initialize token-weighted totals.
 3. Traverse consecutive non-overlapping windows.
 4. Predict with recurrent state reset at each window boundary.
 5. Sum loss within windows and count all scored tokens.
 6. Divide once to obtain mean next-token cross-entropy.
-

CODE

```

"""Fixed-window evaluation loss - Chapter 10's protocol, importable.

Scores a sequence in consecutive `window`-sized chunks with the recurrent (or
attention) state reset at each boundary, so train and held-out numbers are
comparable across chapters and architectures.
"""

import torch
import torch.nn.functional as F
from torch import nn

# [1]
@torch.no_grad()
def fixed_window_loss(
    net: nn.Module,
    sequence: torch.Tensor,
    *,
    vocab: int,
    window: int = 100,

```

```

) -> float:
    """Mean next-token cross-entropy over non-overlapping windows."""
    # [2]
    net.eval()
    total_loss, n_tokens = 0.0, 0
    # [3]
    for start in range(0, len(sequence) - 1, window):
        stop = min(start + window, len(sequence) - 1)
        xb = sequence[start:stop].unsqueeze(0)
        yb = sequence[start + 1 : stop + 1]
        # [4]
        logits = net(xb)
        if isinstance(logits, tuple):          # recurrent models return state
            logits = logits[0]                 # state resets at each window
        # [5]
        total_loss += F.cross_entropy(
            logits.reshape(-1, vocab), yb, reduction="sum"
        ).item()
        n_tokens += yb.numel()
    # [6]
    return total_loss / n_tokens

```

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Run char lm train run.
 3. Report or visualize the measured result.
-

CODE

```

from dlbook.training import fit_next_token    # Listing 10.1, imported

# [1]
torch.manual_seed(6050)
model = CharLSTM(len(chars))
# [2]
model, _ = fit_next_token(model, train_data, vocab=len(chars))

from dlbook.evaluation import fixed_window_loss    # Listing 10.2, imported

# [3]
held_out_loss = fixed_window_loss(model, valid_data, vocab=len(chars))
print(f"held-out fixed-window loss {held_out_loss:.2f}")

```

```

step    0    loss 4.62
step  500    loss 2.28

```



```

step 1000    loss 1.93
step 1500    loss 1.75
step 2000    loss 1.57
step 2500    loss 1.42
held-out fixed-window loss 1.89

```

The loss falls from $\ln(V) \approx 4.6$ for the printed vocabulary size (uniform guessing) to 1.42 on the last sampled training minibatch. The deterministic held-out tail is harder, at 1.89, but both are far below uniform guessing. The model has learned a great deal about what this book's next character tends to be. To *hear* what it learned, generate: feed a prompt, sample the next character from the softmax (divided by a **temperature** that sharpens or flattens it, Chapter 2's dial between hard max and uniform), feed that character back in, repeat. The hidden state streams; nothing is recomputed.

PLAN

1. Define the reusable `sample` helper.
 2. Sample from the model, one character at a time.
-

CODE

```

# [1]
@torch.no_grad()
def sample(prompt: str, n: int = 300, temp: float = 0.8) -> str:
    model.eval()
    idx = torch.tensor([[stoi.get(c, 0) for c in prompt]])
    logits, state = model(idx)
    out = prompt
    for _ in range(n):
        p = F.softmax(logits[0, -1] / temp, -1)
        nxt = torch.multinomial(p, 1)[None]
        out += chars[nxt.item()]
        logits, state = model(nxt, state)           # consume each sample once
    return out

# [2]
print(sample("The gradient ", n=300))

```

The gradient mupolition both
lates
chood each the burk's map into a mack from the bupfured way
intene agmenten. Pyout nothing buith and why the dirge melies to exactly. What
↪ algien is

```

$$
\logit_i).
ixsections.

```

```
## Exercise reseds the barkwarper from to tve optimized $\rightarrow$ and
↪ classitivations, in t
```

Read the output honestly, because both halves of the verdict teach. What it *got*: words, spacing, sentence rhythm, markdown furniture (asterisks, pipes, colons), and the book's working vocabulary (training, batch, layer, convolution in various misspellings), all learned from raw characters by a one-layer recurrence reading 100-character chunks. What it *lacks*: meaning. Clauses trail off; equations open and never close; the prose has the book's accent with none of its intent. Some of that is scale (a 150k-character corpus and 128 hidden units is a *tiny* language model), and some of it is the finite state. By the end of a sentence, the beginning has been squeezed through the LSTM's two 128-dimensional states, (**h**, **c**): 256 scalars in all. Both diagnoses point down the road this part of the book is traveling.

10.7. A full-scale rematch: words, layers, and corpus size

The character model answered the chapter's mechanism question on a CPU-sized snapshot. It did not answer what happens when tokens become words, the corpus grows past two million training tokens, and recurrent depth increases. We ran that rematch on WikiText-2 with a 33,279-word training vocabulary. Three tied-embedding LSTMs share the same optimizer and 12-epoch budget; validation selects one checkpoint before the test split is opened.

PLAN

1. Load the nine pinned WikiText-2 Rivanna records.
 2. Summarize parameter count and held-out perplexity by model size.
-

CODE

```
# [1]
import json
from pathlib import Path

wikitext_root = Path("../.. /experiments/rivanna/results/wikitext")
wikitext_records = [
    json.loads(path.read_text()) for path in sorted(wikitext_root.glob("*.json"))
]
wikitext_order = ["small", "medium", "large"]

# [2]
wikitext_summary = {}
for size in wikitext_order:
    rows = [row for row in wikitext_records if row["size"] == size]
    perplexities = torch.tensor([row["test_perplexity"] for row in rows])
    wikitext_summary[size] = {
        "parameters": rows[0]["parameter_count"],
        "values": perplexities,
        "mean": perplexities.mean().item(),
```

```

"sd": perplexities.std(unbiased=True).item(),
}

```

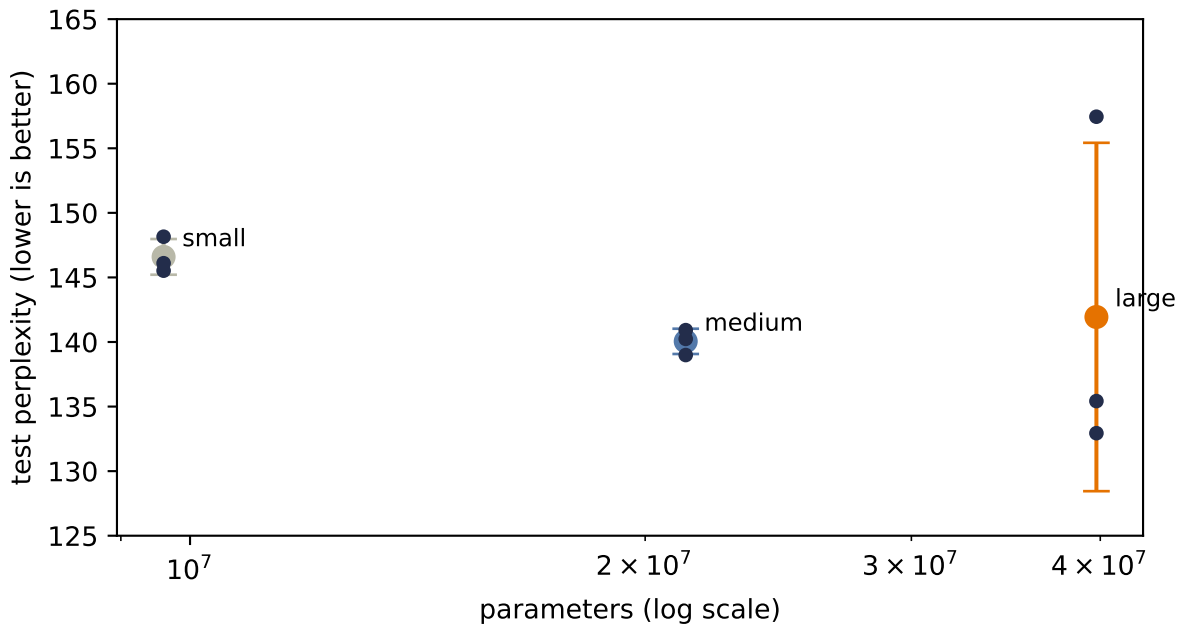


Figure 10.6.: Word-level LSTM scale on WikiText-2. Across seeds 6050–6052, the 9.61M-parameter model reaches test perplexity 146.59 ± 1.38 , the 21.27M model 140.05 ± 0.98 , and the 39.77M model 141.94 ± 13.49 . The medium model improves reliably; the largest has two strong runs and one poor run, so more parameters did not buy a stable improvement under the shared 12-epoch recipe. Points are seed-level tests; bars are means $\pm$ one sample standard deviation.

The change in tokenization is audible: generated samples now contain full words and longer grammatical fragments, though their facts and discourse still drift. The numbers make the scale lesson more precise. Moving from 9.61M to 21.27M parameters improves every seed. Moving again to 39.77M does not improve the mean reliably: two runs are strongest, while seed 6050 ends at perplexity 157.44. Capacity, optimization, and data must scale together. “Bigger” is not a result until the training contract makes it one.

10.8. What a fixed-size state cannot hold

This chapter closes with the bridge this part of the book now crosses. If a recurrent cell can read a sequence into a state, then two of them can translate: an **encoder** reads the source sentence into its final state, a **decoder** spins that state back out as words in another language. That architecture, plus its training tricks, is [Chapter 11](#). But notice what the design asks of one fixed-size state: the *entire sentence*, compressed into $(\mathbf{h}, \mathbf{c})$, no matter how long the sentence runs. The char-LM’s trailing clauses were this bottleneck in miniature. The honest fix is not a bigger vector; it is admitting that the decoder should be allowed to *look back at all of the encoder’s states*, not just the last. Choosing where to look, on the fly, is a similarity computation you have been prepared for since Chapter 1’s dot products. The next

chapter will first make the fixed-code failure visible in translation. Part IV will then build an alternative from that need.

The fixed-size state is about to face that rematch: the alternative will keep the evidence available instead of squeezing all of it through one vector. Much later the state will return—not as a defeated design, but as a deliberate point on a memory tradeoff whose price we can finally name.

i Check yourself

Close the book for one minute and unroll the recurrence.

- What is shared across timesteps, and what changes at each step?
- How does an LSTM cell create a less obstructed path for information and gradients?
- Why does a fixed-width recurrent state become a bottleneck as evidence grows?

10.9. Okay, so — weight sharing moved into time

1. **Order is the signal.** Chapter 6’s MLP could absorb one fixed pixel permutation when retrained; shuffle a sentence into arbitrary orders and its meaning changes. Sequences add variable length and streaming, and the fixed-size toolkit of Parts I–II has no honest answer.
2. **The third weight sharing:** one function f , applied at every timestep, carrying a hidden state (Equation 10.1, Equation 10.2). This is stationarity as inductive bias, exactly as sharing across space was in Part II. The unrolled RNN is a deep network with tied weights.
3. **Tied depth in time = Chapter 5’s nightmare:** BPTT multiplies T near-identical Jacobians (Equation 10.3) → vanishing or exploding, unless their gains are carefully controlled. Clipping belts in the explosions; truncation caps the cost, but neither lengthens memory. Windows of roughly 20–50 steps are a common practical horizon, not a universal vanilla-RNN ceiling.
4. **The LSTM engineers the memory:** an additive cell-state conveyor belt, Chapter 9’s residual highway laid across time, with three sigmoid valves (forget, input, output; Equation 10.4). Its gradient rides a product of *gates*, not weights (Equation 10.5).
5. **Architecture proposes, initialization disposes:** at lag 80 the default LSTM scores chance ($\sigma(0)^{80} \approx 10^{-24}$); set the forget bias to +1 and the task is solved on every seed. GRU packs the same ideas into two gates and one state (Equation 10.6).
6. **A one-layer character LSTM learns this book’s accent** (words, rhythm, LaTeX tics) but not its meaning: part scale, part the finite-state bottleneck. Cramming a whole sequence into 256 recurrent-state scalars is the debt Part III carries forward. The next chapters will earn the mechanism that repays it.

Sources and further reading

- Merity et al., *Pointer Sentinel Mixture Models*: introduces the WikiText corpora used by the chapter’s pinned word-level language-model scale study.
- Elman, *Finding Structure in Time*: the classic simple recurrent network and its view of hidden state as temporal context.
- Hochreiter & Schmidhuber, *Long Short-Term Memory*: the original gated-memory architecture and constant-error pathway.

- Cho et al., *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*: introduces the gated recurrent unit in its encoder–decoder setting.
- Pascanu, Mikolov, & Bengio, *On the Difficulty of Training Recurrent Neural Networks*: geometric analysis of vanishing/exploding gradients and gradient clipping.
- Mikolov et al., *Efficient Estimation of Word Representations in Vector Space*: the CBOW and skip-gram predictive objectives.
- Pennington, Socher, & Manning, *GloVe: Global Vectors for Word Representation*: static embeddings learned from global word co-occurrence statistics.

Exercises

1. **(Pencil.)** Count parameters: a width- k window MLP mapping k one-hot tokens (vocabulary V) through hidden width H to V logits, versus the vanilla RNN of Equation 10.2 with the same V and H . Which count depends on how much context the model can use, and why is that the whole argument of this chapter in two numbers?
2. **(Pencil.)** From Equation 10.3, show that if $\tanh$ operated in its linear regime ($\tanh' \approx 1$), the gradient norm grows or decays like the spectral radius of $\mathbf{W}_{hh}$ raised to T . What does $\tanh'(0) = 1$ say about the moment right after initialization, when $\mathbf{h} \approx \mathbf{0}$?
3. **(Code.)** Implement the reset-before GRU in Equation 10.6 as a manual loop and enter it in the lag-80 memory test. Then change only the candidate update to PyTorch’s reset-after convention and verify that version against `nn.GRU`. Does either need an initialization trick, and which bias would you set?
4. **(Code.)** The chapter starts every sequence with $\mathbf{h}_0 = \mathbf{0}$. Make $\mathbf{h}_0$ a learned `nn.Parameter` in the recall task and retrain. Where does its gradient come from, and when could a learned start plausibly matter?
5. **(Code.)** Sample the character model at temperatures 0.2, 0.8, and 1.5, and with the temperature near zero compare against picking the argmax at every step. Connect what you see to Chapter 2’s hard-max-versus-softmax discussion: which failure mode of the hard max does greedy decoding reproduce?

11. Encoder–Decoder, Teacher Forcing, Beam Search

Chapter 10 ended with a promise: if one recurrent cell can read a sequence *into* a state, two cells can translate: one reads, one writes. This chapter builds that machine and the craft around it: how to train it fast (teacher forcing), what poisonous bookkeeping to avoid (padding), and how to get answers back out of it (greedy versus beam search). It ends where Part III must: staring at the fixed-size recurrent state in the middle.

The names are not new. In the autoencoder interlude, an encoder compressed a fixed object and a decoder reconstructed it. Here both maps become recurrent processes. The encoder consumes however many source tokens arrive; the decoder produces however many target tokens are needed. What survives is the handoff: every source detail still has to cross through one fixed-size context.

Before recurrence, every model lived on a **fixed-size contract**: 784 pixels in, 10 logits out; 28×28 garments in, one verdict out. The interesting world breaks that contract on both ends at once: translation maps a sentence of *any* length to a sentence of *some other* length, in a different vocabulary. Chapter 10's recurrent cell broke the contract on the input side; today we break it on both.

11.1. Two RNNs and a handoff

The architecture is one idea long. An **encoder** RNN reads the source sequence and keeps nothing but its final state, the entire input compressed into fixed-size memory, the **context**. A **decoder** RNN starts *from* that state and generates the output token by token, feeding each token back in, exactly like Chapter 10's character model. A language model, in other words, but one with something to say.

Our laboratory task keeps the core mechanics of translation and none of the downloads: **date normalization**. The source language is human-written dates in several dialects; the target language is ISO 8601:

march 5, 2021	-> 2021-03-05
26 september 2024	-> 2024-09-26
12/15/1950	-> 1950-12-15

Variable-length input, a different output alphabet, and real long-range structure (the year arrives *last* in most sources but must be emitted *first*). The target is deliberately fixed at ten characters, so this laboratory does **not** test variable-length target padding. It does include a dialect ambiguity that gives beam search something genuine to reveal: numeric dates are written **mm/dd/yyyy** by 70% of our imaginary writers (the US convention) and **dd/mm/yyyy** by 30%, and nothing in 03/05/2021 says which. Machine translation in miniature.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable `make_date` helper.
3. Implement the two date languages, generated on the fly.
4. Report or visualize the measured result.

CODE

```

import random, torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

# [1]
MONTHS = ["january", "february", "march", "april", "may", "june", "july",
          "august", "september", "october", "november", "december"]
DAYS = dict(zip(MONTHS, [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]))

# [2]
def make_date(rng: random.Random) -> tuple[str, str]:
    mo = rng.randrange(12)
    d = rng.randrange(1, DAYS[MONTHS[mo]] + 1)
    y = rng.randrange(1950, 2026)
    iso = f"{y:04d}-{mo+1:02d}-{d:02d}"
    style = rng.random()
    if style < 0.35:
        src = f"{MONTHS[mo]} {d}, {y}"
    elif style < 0.6:
        src = f"{d} {MONTHS[mo]} {y}"
    elif rng.random() < 0.7:
        src = f"{mo+1:02d}/{d:02d}/{y}"          # US convention: mm/dd
    else:
        src = f"{d:02d}/{mo+1:02d}/{y}"          # EU convention: dd/mm
    return src, iso

rng = random.Random(6050)
pairs, seen_sources = [], set()
# [3]
while len(pairs) < 9000:
    pair = make_date(rng)
    if pair[0] not in seen_sources:
        seen_sources.add(pair[0])
        pairs.append(pair)
train_pairs = pairs[:8000]
valid_pairs = pairs[8000:8500]
test_pairs = pairs[8500:]

```

```
# [4]
print(*pairs[:4], sep="\n")
train_sources = {s for s, _ in train_pairs}
valid_sources = {s for s, _ in valid_pairs}
test_sources = {s for s, _ in test_pairs}
print("exact-source overlaps: "
      f"train/valid {len(train_sources & valid_sources)}, "
      f"train/test {len(train_sources & test_sources)}, "
      f"valid/test {len(valid_sources & test_sources)}")

PAD, BOS, EOS = 0, 1, 2 # special tokens first
SRC = sorted(set("".join(s for s, _ in pairs)))
TGT = sorted(set("".join(t for _, t in pairs)))
src_stoi = {c: i + 3 for i, c in enumerate(SRC)}
tgt_stoi = {c: i + 3 for i, c in enumerate(TGT)}
tgt_itos = {i: c for c, i in tgt_stoi.items()}
V_src, V_tgt = len(SRC) + 3, len(TGT) + 3
print(f"source vocab {V_src}, target vocab {V_tgt}")
```

```
('26 september 2024', '2024-09-26')
('april 7, 1962', '1962-04-07')
('12/15/1950', '1950-12-15')
('14 february 1997', '1997-02-14')
exact-source overlaps: train/valid 0, train/test 0, valid/test 0
source vocab 37, target vocab 14
```

The rejection loop is evaluation hygiene, not decoration. It removes duplicate source strings **before** the 8,000/500/500 train/validation/test split. We use validation sources for every learning curve and reserve test sources for one final audit after the design is fixed. The same underlying date may appear in another written dialect, which is the structural generalization we want; exact-string memorization is no longer available.

Three special tokens run the show, the same conventions as the course’s full translation pipeline: `<pad>` fills the short sequences of a batch out to a rectangle, `<bos>` tells the decoder “begin,” and `<eos>` lets the *model* say “I’m done” — remember, the machine must be free to choose its output length.

One more new tool, and it earns a pause. Chapter 10 fed characters in as one-hot vectors. This chapter gives each token a learned vector instead: `nn.Embedding`, a table with one trainable row per vocabulary entry. There is no new math — an embedding is exactly a one-hot vector times a weight matrix, Chapter 1’s linear map wearing a lookup table’s clothes — but the reading matters: *the address is hard, the content is learned*. Row 17 is fetched by index, rigidly; what lives in row 17 is whatever training decides that token should mean. Keep the phrase “hard address, learned content” nearby: in [Chapter 13](#) we soften the address too, and that will be the whole story.

PLAN

1. Encoder, decoder, and the handoff.

CODE

```
# [1]
class Seq2Seq(nn.Module):
    def __init__(self, v_src: int, v_tgt: int, embed: int = 32,
                  hidden: int = 128, packed: bool = True):
        super().__init__()
        self.packed = packed
        self.src_emb = nn.Embedding(v_src, embed, padding_idx=PAD)
        self.tgt_emb = nn.Embedding(v_tgt, embed, padding_idx=PAD)
        self.encoder = nn.LSTM(embed, hidden, batch_first=True)
        self.decoder = nn.LSTM(embed, hidden, batch_first=True)
        self.out = nn.Linear(hidden, v_tgt)

    def encode(
        self, S: torch.Tensor, lengths: torch.Tensor,
    ) -> tuple[torch.Tensor, torch.Tensor]:
        e = self.src_emb(S)                                # (B, T_src, embed)
        if self.packed:                                    # stop at each true length
            e = nn.utils.rnn.pack_padded_sequence(
                e, lengths, batch_first=True, enforce_sorted=False)
        _, state = self.encoder(e)
        return state                                       # (h, c): the context

    def forward(self, S: torch.Tensor, lengths: torch.Tensor,
                 T_in: torch.Tensor) -> torch.Tensor: # teacher-forced pass
        o, _ = self.decoder(self.tgt_emb(T_in), self.encode(S, lengths))
        return self.out(o)                                # (B, T_tgt, V_tgt) logits
```

The whole architecture is that `encode` return value: the LSTM’s final **(h, c)**, handed to the decoder as its *initial* state. With hidden width 128, that bridge carries two 128-dimensional vectors, or **256 scalars**. Everything the decoder will ever know about the source crosses it.

11.2. The padding trap

Before the machinery works, an honest detour through the way it *fails*, because experience says everyone hits it once. Batching variable-length sequences means padding them to a rectangle, and padding poisons two places:

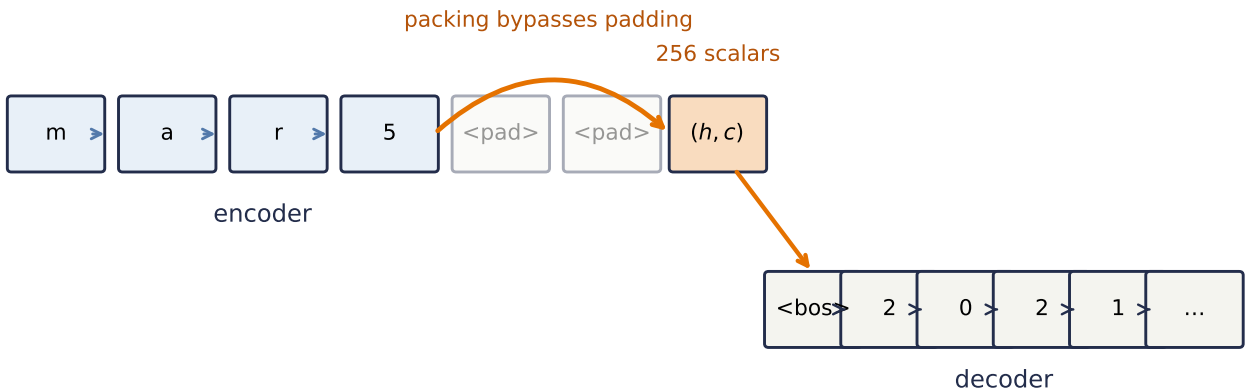


Figure 11.1.: The encoder–decoder handoff. Packing stops each encoder at its last real source token, rather than marching through right-padding. Only the final (h, c) pair crosses the bridge to initialize the decoder; the earlier encoder states are discarded. Part IV will keep those states and let the decoder look back.

1. **The loss.** Positions holding `<pad>` are not predictions to be graded. `F.cross_entropy(..., ignore_index=PAD)` excludes them. Our ISO targets all have the same length, so this branch is dormant here, but it makes the loop correct for genuinely variable-length targets. (In the vocabulary of [Chapter 4](#) this is Case 1 over *real* tokens — and which tokens count, the denominator, is part of the objective’s definition, so it must be stated.)
2. **The encoder.** Subtler and far worse: an LSTM does not know your zeros are padding. It keeps stepping through them, and every pad step smears the final state, the context pair, for every sequence shorter than the batch’s longest. Worse still, at inference you feed single sequences with *no* padding, so the model is tested in a regime it never trained in.

Watch what that costs. Same model, same data, same budget; the only difference is whether the encoder is told the true lengths (`pack_padded_sequence`, which runs the recurrence only through each sequence’s real tokens):

PLAN

1. Define the reusable helpers: `batchify`, `train_seq2seq`, and `translate`.
 2. Define the reusable helpers: `exact_match` and `unambiguous`.
 3. Prepare the inputs and fixed settings for the example.
 4. Train twice — naive right-padding vs. packed sequences.
 5. Report or visualize the measured result.
-

CODE

```

# [1]
def batchify(prs: list[tuple[str, str]], idx: list[int]) -> tuple[
    torch.Tensor, torch.Tensor, torch.Tensor
]:
    srcs = [torch.tensor([src_stoi[c] for c in prs[i][0]]) for i in idx]
    lengths = torch.tensor([len(s) for s in srcs])
    tgts = [torch.tensor([BOS] + [tgt_stoi[c] for c in prs[i][1]] + [EOS])
            for i in idx]
    S = nn.utils.rnn.pad_sequence(srcs, batch_first=True, padding_value=PAD)
    T = nn.utils.rnn.pad_sequence(tgts, batch_first=True, padding_value=PAD)
    return S, lengths, T

def train_seq2seq(mode: str = "tf", packed: bool = True, epochs: int = 25,
                  seed: int = 6050, batch: int = 128,
                  checkpoints: tuple[int, ...] = ()) -> tuple[
    Seq2Seq, dict[int, float]
]:
    torch.manual_seed(seed)
    model = Seq2Seq(V_src, V_tgt, packed=packed)
    opt = torch.optim.Adam(model.parameters(), lr=1e-3)
    n, curve = len(train_pairs), {}
    for ep in range(1, epochs + 1):
        model.train()
        perm = torch.randperm(n)
        for i in range(0, n, batch):
            S, L, T = batchify(train_pairs, perm[i:i + batch].tolist())
            if mode == "tf":
                # gold rail: one fused call
                logits = model(S, L, T[:, :-1])
            else:
                # free-running: its own rail
                state = model.encode(S, L)
                tok, outs = T[:, :1], []
                for t in range(T.shape[1] - 1):
                    o, state = model.decoder(model.tgt_emb(tok), state)
                    logit = model.out(o)
                    outs.append(logit)
                    tok = logit.argmax(-1)
                    # feed its own prediction
                logits = torch.cat(outs, 1)
            loss = F.cross_entropy(logits.reshape(-1, V_tgt),
                                   T[:, 1:].reshape(-1), ignore_index=PAD)
            opt.zero_grad(); loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), 5.0)
            opt.step()
        if ep in checkpoints:
            curve[ep] = exact_match(model, valid_unamb[:400])
    return model, curve

@torch.no_grad()
def translate(model: Seq2Seq, src: str, maxlen: int = 12) -> str:

```

```

model.eval()
S = torch.tensor([[src_stoi[c] for c in src]])
state = model.encode(S, torch.tensor([len(src)]))
tok, out = torch.tensor([[BOS]]), ""
for _ in range(maxlen):
    o, state = model.decoder(model.tgt_emb(tok), state)
    tok = model.out(o).argmax(-1) # greedy: best next char
    if tok.item() == EOS:
        break
    out += tgt_itos.get(tok.item(), "?")
return out

# [2]
@torch.no_grad()
def exact_match(model: Seq2Seq, prs: list[tuple[str, str]]) -> float:
    return sum(translate(model, s) == t for s, t in prs) / len(prs)

def unambiguous(prs: list[tuple[str, str]]) -> list[tuple[str, str]]:
    return [(s, t) for s, t in prs
            if "/" not in s or int(s[:2]) > 12 or int(s[3:5]) > 12]

# [3]
valid_unamb = unambiguous(valid_pairs)
test_unamb = unambiguous(test_pairs)

cps = (2, 4, 6, 8, 12, 16, 20, 25)
# [4]
naive, _ = train_seq2seq(packed=False)
packed, packed_curve = train_seq2seq(packed=True, checkpoints=cps)
# [5]
print(f"naive right-padding:  validation exact match "
      f"{exact_match(naive, valid_unamb):.1%}")
print(f"packed (true lengths): validation exact match "
      f"{exact_match(packed, valid_unamb):.1%}")

```

```

naive right-padding:  validation exact match 37.3%
packed (true lengths): validation exact match 95.7%

```

Validation exact match jumps from 37.3% to 95.7%, a 58.4-point gap. Not from a better architecture, optimizer, or dataset; from telling the encoder where the sentences actually end. Let the gap sink in, because this is the cheapest lesson in the book: the two runs are identical except for bookkeeping, and the bookkeeping was worth more than every design decision in Part III so far. The sealed test set remains untouched until both training-mode designs below are fixed.

The mechanism is worth *seeing* once. Take the naive model and one short validation source, and watch its encoder state as the input runs past the last real character into the padding:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Measure encoder-state drift through the padding tail.

CODE

```
# [1]
short_src = min((s for s, _ in valid_pairs), key=len)
ids = torch.tensor([src_stoi[c] for c in short_src])
n_pads = 8
padded_ids = torch.cat([ids, torch.zeros(n_pads, dtype=torch.long)])
with torch.no_grad():
    states, _ = naive.encoder(naive.src_emb(padded_ids[None]))
states = states.squeeze(0) # (len + n_pads, hidden)
T = len(ids)
# [2]
drift = (states[T - 1:] - states[T - 1]).norm(dim=1)
```

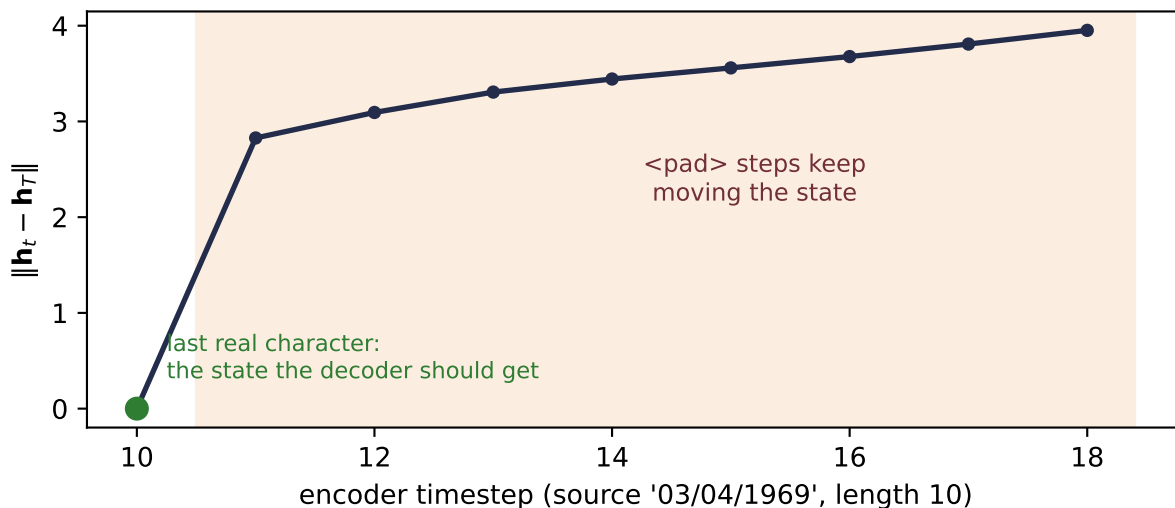


Figure 11.2.: Why naive right-padding hurts: the encoder state does not stop when the sentence does. One short validation source is encoded by the naive model; at its last real character (green) the state is exactly what the decoder should receive. A *single* step already throws the state most of the way to its final drift, and each further step compounds it (navy) — the decoder is handed the endpoint of the shaded excursion, a state the unpadded inference-time encoder never produces. Packing simply stops the clock at the green dot.

⚠️ Trap: the model will happily read your padding

Nothing in the tensor marks `<pad>` as meaningless; meaning is something *you* enforce, in two places: `ignore_index` for the loss, `pack_padded_sequence` (or an explicit mask) for the encoder. When a

sequence model underperforms mysteriously, audit the padding before touching the architecture. And file the general shape of this tool away: telling a model *which positions it may not look at* is called **masking**, and it returns center-stage in [Chapter 14](#), where a *causal* mask is what turns a transformer into a language model.

11.3. Teacher forcing: training on the gold rail

Look back at the two `mode` branches in the training loop; they are this section. When the decoder is trained, what should it see as *its own previous output*?

Free-running (“fr”) is the honest answer: feed the model’s own predictions back in, exactly as at inference. But its flaws are structural. Early in training those predictions are garbage → the decoder learns conditioned on garbage → every sequence must be generated token-by-token in a Python loop, because step t ’s input does not exist until step $t - 1$ is computed. Slow, and unstable exactly when training is young.

Teacher forcing (“tf”) conditions each step on the *ground-truth* previous token instead, the gold rail. Every conditioning input is then known in advance → the whole target can be submitted in **one fused decoder call** (the `model(S, L, T[:, :-1])` line), rather than a Python feedback loop. The LSTM still processes time recurrently inside that call; teacher forcing does not make its timesteps mathematically parallel. Every step nevertheless learns from a sensible context from epoch one. Watch both effects:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Train the free-running comparison.
 3. Audit all three models once on the final test split.
-

CODE

```
# [1]
tf_model, tf_curve = packed, packed_curve
# [2]
fr_model, fr_curve = train_seq2seq("fr", checkpoints=cps)
# [3]
print("one final test audit (unambiguous sources):")
for label, candidate in [("naive padding", naive),
                        ("packed / teacher forced", tf_model),
                        ("packed / free-running", fr_model)]:
    print(f" {label:25s} {exact_match(candidate, test_unamb):.1%}")
```

```
one final test audit (unambiguous sources):
naive padding          34.3%
packed / teacher forced 93.1%
packed / free-running  99.1%
```

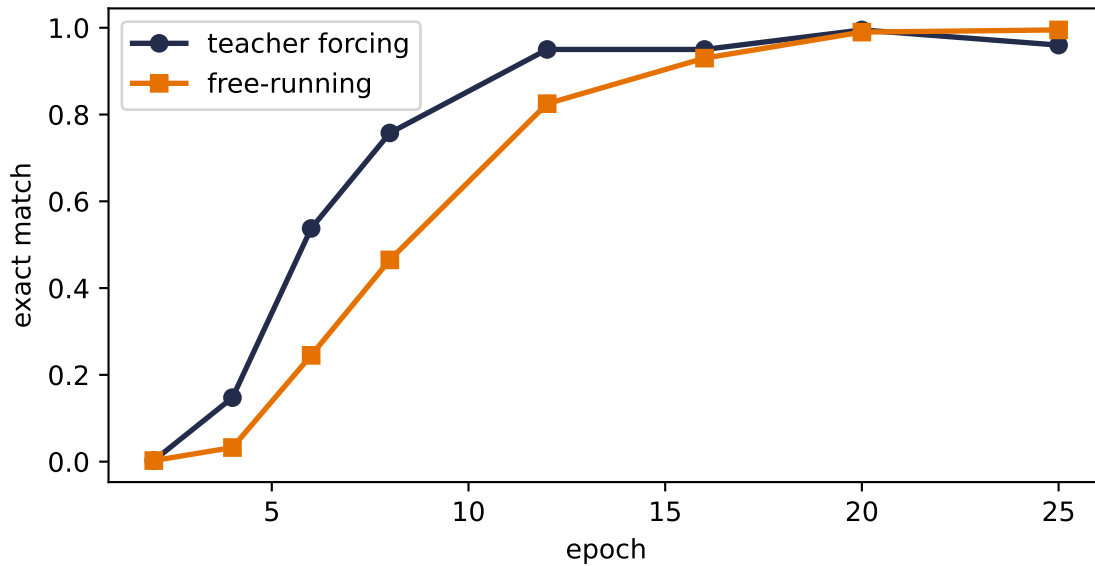


Figure 11.3.: Learning curves on a fixed subset of 400 unambiguous validation source strings, disjoint from both training and test. Teacher forcing leads through the early and middle epochs (53.8% vs. 23.8% at epoch 6; 95.0% vs. 81.2% at epoch 12) and uses one fused decoder call per batch where free-running needs a Python token loop. By epoch 20 the validation curves meet; the single final test audit favors free-running.

So teacher forcing is the pragmatic default: fused, stable, and fast out of the gate. What it costs has a name. A teacher-forced decoder has *only ever seen perfect histories*. At inference it must condition on its own imperfect predictions — a distribution it never trained on. This mismatch is **exposure bias**: the model never learned to recover from its own mistakes, so one early slip can send the rest of the sequence somewhere strange. You can see the shape of this failure in our model’s rare residual errors:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Implement the residual errors, up close.
-

CODE

```
# [1]
errs = [(s, t, translate(tf_model, s))
        for s, t in test_unamb if translate(tf_model, s) != t]
print(f"{len(errs)} errors on {len(test_unamb)} unambiguous test dates; a sample:")
# [2]
for s, t, g in errs[:3]:
    print(f" {s!r} -> {g!r}    (truth {t})")
```

```
30 errors on 437 unambiguous test dates; a sample:
'03/21/1958' -> '1958-03-22'    (truth 1958-03-21)
```

```
'01/22/1996' -> '1996-02-12'    (truth 1996-01-22)
'21 november 2013' -> '2013-11-22'    (truth 2013-11-21)
```

The errors are plausible-looking digit substitutions and order slips. The decoder has learned a date-shaped language, but no hard mechanism forces each output field to copy the corresponding source fact. Off the gold rail, one wrong character can alter the context for every character after it. Hold that thought two sections.

i Scheduled sampling: renting the rail

The engineered compromise (Bengio et al., 2015): during training, at *each timestep* flip a biased coin — probability ϵ of the gold token, $1 - \epsilon$ of the model’s own prediction — and anneal ϵ from 1 toward 0 as training matures. Early epochs enjoy the gold rail; late epochs practice recovery. One design detail from the original paper worth repeating: flip **per token, not per sequence** — committing a whole sequence to model predictions early in training lets consecutive errors amplify. Exercise 3 has you build it.

11.4. Getting answers out: search

Training is settled; now the neglected half. At inference the decoder defines a probability for every possible output sequence, and we want the best one. That is a *search problem* over a tree where every node branches V_{tgt} ways; exhaustive search is V^T , hopeless. Everything practical is a heuristic walk through that tree.

Greedy search, which `translate` above already performs, takes the single most probable token at each step and never looks back. Complexity $O(V \cdot T)$, quality: usually fine, occasionally myopic. The failure mode is structural: a token that looks best *right now* can commit the decoder to a mediocre continuation, when a slightly worse token now would have opened a much better path. Maximizing each step is not maximizing the sequence. You met this exact gap in [Chapter 2](#), where the hard max’s stepwise confidence hid the alternatives softmax kept alive.

Beam search widens the walk: carry the k best partial sequences (the *beam*), expand each by all V tokens, keep the top k by *cumulative* log-probability, repeat. Complexity $O(k \cdot V \cdot T)$, a small multiple of greedy, nothing like exhaustive.

PLAN

1. Beam search, k candidate rails at once.
-

CODE

```
# [1]
@torch.no_grad()
def beam_search(model: Seq2Seq, src: str, k: int = 5,
                maxlen: int = 12) -> list[tuple[str, float]]:
    model.eval()
```



```

S = torch.tensor([[src_stoi[c] for c in src]])
state = model.encode(S, torch.tensor([len(src)]))
beams = [([], 0.0, state, False)] # (tokens, logp, state, done)
for _ in range(maxlen):
    cand = []
    for seq, lp, st, done in beams:
        if done:
            cand.append((seq, lp, st, True)); continue
        tok = torch.tensor([seq[-1] if seq else BOS])
        o, st2 = model.decoder(model.tgt_emb(tok), st)
        logp = F.log_softmax(model.out(o)[0, -1], -1)
        top = torch.topk(logp, k)
        for lg, ix in zip(top.values, top.indices):
            cand.append((seq + [ix.item()], lp + lg.item(), st2,
                          ix.item() == EOS))
    beams = sorted(cand, key=lambda b: -b[1])[:k]
    if all(b[3] for b in beams):
        break
return ["".join(tgt_itos.get(i, "?") for i in seq if i > 2), lp]
        for seq, lp, _, _ in beams]

```

Now the payoff, and it is exactly where we buried it: the ambiguous dates. For 03/05/2021, “the one best answer” is the wrong question. The training world itself was split 70/30 between two readings. Greedy silently hands you one sequence; beam search shows which complete alternatives the model ranks highly:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Beam search on genuinely ambiguous dates.
-

CODE

```

# [1]
amb = [(s, t) for s, t in test_pairs
        if "/" in s and int(s[:2]) <= 12 and int(s[3:5]) <= 12]
# [2]
for s, t in amb[:4]:
    print(f"{s!r}    truth: {t!r}    greedy: {translate(tf_model, s)!r}")
    for text, lp in beam_search(tf_model, s, k=4)[:2]:
        print(f"    beam: {text!r}    joint log score = {lp:.2f}")
    print()

```

```

'09/02/1995'    truth: '1995-09-02'    greedy: '1995-09-02'
    beam: '1995-09-02'    joint log score = -0.23
    beam: '1995-09-20'    joint log score = -1.84

```

```
'12/06/2006'   truth: '2006-06-12'   greedy: '2006-12-06'
    beam: '2006-12-06'   joint log score = -0.21
    beam: '2006-06-12'   joint log score = -2.51

'08/01/1989'   truth: '1989-08-01'   greedy: '1989-08-01'
    beam: '1989-08-01'   joint log score = -0.14
    beam: '1989-01-08'   joint log score = -2.51

'01/03/1991'   truth: '1991-01-03'   greedy: '1991-01-03'
    beam: '1991-01-03'   joint log score = -0.54
    beam: '1991-03-01'   joint log score = -1.23
```

Each displayed score is the sum of token log-probabilities along that one sequence, including `<eos>` when it was reached. These are **raw joint log scores**, not a normalized distribution over the printed beam, so exponentiating two of them and calling the result a 70/30 posterior would be wrong. Some pairs correspond cleanly to the month/day and day/month readings. Others include a high-scoring hybrid that matches neither convention, an honest reminder that beam search searches the model; it does not install a calendar constraint. The ranking still exposes alternatives that greedy search hides.

Honesty requires the other half of the report:

PLAN

1. How often beam actually changes the answer.
 2. Report or visualize the measured result.
-

CODE

```
# [1]
sub = random.Random(0).sample(test_pairs, 200)
n_diff = sum(beam_search(tf_model, s, k=5)[0][0] != translate(tf_model, s)
              for s, _ in sub)
# [2]
print(f"beam-5 disagrees with greedy on {n_diff} of {len(sub)} test dates")
```

```
beam-5 disagrees with greedy on 1 of 200 test dates
```

On this small, well-trained model, beam search changes almost nothing. Beam search earns its keep at the margins, but real translation, with vocabularies of tens of thousands and long sentences, lives entirely at those margins, which is why beam decoding became a standard tool of the seq2seq era.

Length normalization

Summed log-probabilities shrink as sequences grow $\rightarrow$ a raw beam quietly favors short outputs. The standard correction divides the score by L^α with $\alpha \approx 0.6\text{--}0.75$. Our dates all have the same length, so the bias is largely invisible for correct outputs here (Exercise 4 makes you check the remaining error cases); in free-length generation it matters enormously.

11.5. The fixed-size state in the middle

Time to stare at the handoff. Every property of the source that the decoder will ever use crossed one bridge: $(\mathbf{h}, \mathbf{c})$, two 128-dimensional vectors and therefore 256 scalars. Our task fits because a date is three facts, yet its residual digit and order errors are consistent with a decoder reconstructing from a compressed summary rather than consulting source positions directly. They do not, by themselves, prove that compression caused each error. Now scale the ambition: a forty-word sentence, clause structure, names, negations, all crammed through the same fixed pipe, *regardless of length*. Chapter 10 called this the price of the Markov claim; here the bill arrives itemized.

And notice what the architecture throws away: the encoder *had* a state at every step, a whole trajectory of summaries, one per source position, and we kept only the last. The fix that defines Part IV is almost embarrassingly direct: keep them all, and let the decoder *look back* at whichever ones matter for the token it is writing right now. Choosing where to look, by similarity, on the fly: you have owned the primitive since Chapter 1’s dot product, and Chapter 2 promised you the “soft lookup” that makes it differentiable. [Chapter 12](#) builds the classical version; [Chapter 13](#) makes it learnable and comes back for this very date task with an alignment map to show where the decoder looks.

Check yourself

Close the book for one minute and reconstruct the sequence pipeline from memory.

- Which two places must exclude padding, and what tool handles each one?
- What mismatch does teacher forcing create between training and inference?
- What does beam search change, and what part of the learned model does it leave unchanged?

11.6. Okay, so — the fixed-size handoff is the bottleneck

1. **Encoder–decoder = two RNNs and a handoff:** read the source into a final state, hand it over, generate from it. `<bos>` starts the decoder, `<eos>` lets it choose its length, embeddings replace one-hots (hard address, learned content).
2. **Padding poisons twice:** the loss (`ignore_index`) and the encoder (`pack_padded_sequence`). On never-seen source strings, the same model and budget score 34.3% naive versus 93.1% packed. Target padding is included for reuse but not exercised by these fixed-length ISO targets. Audit bookkeeping before architecture.
3. **Teacher forcing** trains the decoder on the gold rail: one fused call with no Python feedback loop and stable early learning, at the cost of **exposure bias**, the train/inference mismatch that free-running avoids by paying with speed and early instability. Scheduled sampling anneals between them, flipping per token.
4. **Decoding is search.** Greedy maximizes steps, not sequences; beam search carries k rails for $O(kVT)$ and surfaces high-scoring alternatives. Its raw joint log scores are rankings, not a normalized posterior, and a beam can contain invalid hybrids. Length normalization (L^α) keeps long outputs competitive.
5. **The bottleneck is now the story.** One fixed-size $(\mathbf{h}, \mathbf{c})$ pair, 256 scalars here, connects reader and writer regardless of input length. The encoder’s per-step states were there all along. Part IV is what happens when we stop throwing them away.

Sources and further reading

- Sutskever, Vinyals, & Le, *Sequence to Sequence Learning with Neural Networks*: the canonical fixed-state LSTM encoder–decoder and source-reversal result.
- Bengio et al., *Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks*: the original exposure-bias intervention used in Exercise 3.
- Bahdanau, Cho, & Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*: removes the single-state bottleneck by learning where to look; Chapter 13 performs the full harvest.

Exercises

1. **(Pencil.)** Count this chapter’s parameters: two embeddings, two LSTMs, one output layer ($V_{\text{src}} = 37$, $V_{\text{tgt}} = 14$, embed 32, hidden 128; an LSTM holds $4[(e + h)h + 2h]$ — derive that first). Where do most parameters live, and why is the answer different from Chapter 8’s LeNet audit?
2. **(Pencil.)** With vocabulary 10^4 and output length 10, compare the number of sequences examined by exhaustive search, greedy, and beam-5. Then explain why beam search with $k = V^T$ is exact and with $k = 1$ is greedy — and why neither endpoint is usable.
3. **(Code.)** Implement scheduled sampling: per token, use the gold input with probability ϵ , the model’s own prediction otherwise, annealing ϵ linearly $1 \rightarrow 0$ over 25 epochs. Compare its learning curve with both curves in Figure 11.3. Does it inherit teacher forcing’s fast start *and* free-running’s finish?
4. **(Code.)** Add length normalization ($/L^\alpha$) to `beam_search` and sweep $\alpha \in \{0, 0.7, 1\}$ on the date task. Explain why correct ten-character outputs are unaffected, inspect whether any erroneous beams change, and design the smallest change to the *task* that would make α matter.
5. **(Code.)** Sutskever et al. (2014) famously improved translation by feeding the source sentence *reversed*. Try it here. Whatever you observe, explain it with this chapter’s tools: which source-target dependencies does reversal shorten in their setting, and which does it shorten — or lengthen — for dates, where the year sits at the end of the source but the start of the target?

Part IV.

Attention: Learning the Similarity

12. Kernel Regression: Attention Before It Was Learnable

Chapter 11 ended with a machine that could translate, and one uncomfortable picture: every source fact had to cross a single fixed-size bridge before the decoder wrote its first token. The encoder had computed a state at every source position. Then we threw all but the last one away.

The repair begins with an almost embarrassing question: why throw them away? Keep the whole sequence of encoder states as a **memory bank**. At each decoding step, ask what the decoder needs, compare that query with every stored state, and retrieve a weighted mixture of the relevant content. Keeping a bank does not make its entries lossless, but it removes the rule that *all* source information must pass through the last entry alone.

One problem remains: how should a query decide the mixture? A nearest-neighbor rule changes its chosen slot abruptly. We first need a smooth similarity-weighted rule, and statistics had one in 1964, half a century before modern neural attention.

Let us start with three observed pairs:

observation	location x_i	response y_i
1	1	1.5
2	3	2.8
3	5	1.8

For a query $q = 3.5$, a Gaussian similarity with bandwidth $h = 0.6$ assigns weights $(0.0002, 0.9413, 0.0585)$. Their weighted mixture is 2.7412. The second observation dominates because its location is closest, the third contributes a little, and the distant first observation is nearly silent. No similarity function was learned; the chosen metric and bandwidth did the routing.

12.1. Chapter 1 already mixed old answers

There is a piece of unfinished business from Chapter 1. Linear regression makes a prediction by mixing the observed targets. It hid that fact inside the normal equation.

Let the augmented design matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times (d+1)}$ include the intercept column, let $\tilde{\mathbf{q}} \in \mathbb{R}^{d+1}$ be a new query with its leading 1, and let $\mathbf{y} \in \mathbb{R}^n$ contain the observed targets. Assuming full column rank,

$$\hat{\boldsymbol{\beta}} = (\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^\top \mathbf{y}.$$

The prediction at the query is therefore

$$\hat{f}(\mathbf{q}) = \tilde{\mathbf{q}}^\top \hat{\boldsymbol{\beta}} = \underbrace{[\tilde{\mathbf{X}}(\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{q}}]}_{\boldsymbol{\lambda}(\mathbf{q})} \mathbf{y}. \quad (12.1)$$

 **Trap:** weighted combination does not mean weighted average

Chapter 1's promise was exact: $\hat{f}(\mathbf{q})$ is a weighted combination of the training targets. With an intercept, the weights sum to one because OLS reproduces a constant response. But **weighted combination** is not the same as **weighted average**. The entries of $\boldsymbol{\lambda}$ may be negative. For keys $(-2, -1, 0, 1, 2)$ and query $q = 1.5$, the OLS weights are

$$(-0.10, 0.05, 0.20, 0.35, 0.50).$$

They sum to one, yet the first is negative. If the targets are $(1, 0, 0, 0, 0)$, OLS predicts -0.10 , outside the observed range $[0, 1]$. Linear regression is allowed to leave the observed targets' convex hull even when the query lies inside the observed key range. For local smoothing, we want a stricter contract: nearby evidence should receive more weight, every weight should be nonnegative, and the weights should sum to one.

12.2. The magnifying glass becomes a kernel

A **smoothing kernel** is a similarity rule. It looks at a query and a stored key and returns a nonnegative affinity: large when they are close, small when they are far. To avoid confusing the kernel function with the matrix of keys later, we write it as κ_h . The Gaussian choice is

$$\kappa_h(\mathbf{q}, \mathbf{k}) = \exp\left(-\frac{\|\mathbf{q} - \mathbf{k}\|^2}{2h^2}\right), \quad h > 0. \quad (12.2)$$

The bandwidth h sets the width of the magnifying glass. A small value sees only the closest keys; a large value blends a broad neighborhood.

Given observed pairs $(\mathbf{x}_i, y_i)$, normalize the affinities,

$$\alpha_i(\mathbf{q}) = \frac{\kappa_h(\mathbf{q}, \mathbf{x}_i)}{\sum_{j=1}^n \kappa_h(\mathbf{q}, \mathbf{x}_j)}, \quad \hat{f}_h(\mathbf{q}) = \sum_{i=1}^n \alpha_i(\mathbf{q}) y_i. \quad (12.3)$$

This is the **Nadaraya–Watson estimator**, proposed independently by Nadaraya and Watson in 1964. For the strictly positive Gaussian, each $\alpha_i > 0$ and $\sum_i \alpha_i = 1$. Scalar predictions remain between the smallest and largest observed values. Unlike OLS, this operator smooths locally while keeping every prediction inside the observed values' convex hull.

The endpoint behavior is worth stating precisely. As $h \rightarrow 0^+$, the weight concentrates on the unique nearest key; exact nearest-key ties split the mass. At $h = 0$ the formula is undefined. As $h \rightarrow \infty$, every finite distance looks alike and the weights approach $1/n$.

⚠ Trap: the kernel supplies the weights; the constant fit supplies the average

It is tempting to say that attention solves an unrestricted regression problem with zero regularization. That is too loose. An unrestricted function could interpolate the stored values and remain undetermined elsewhere; nothing in that statement forces an average at a new query. The exact claim is local and smaller. For a fixed query $\mathbf{q}$, fit one constant vector $\mathbf{c}$:

$$\mathbf{c}^*(\mathbf{q}) = \arg \min_{\mathbf{c}} \frac{1}{2} \sum_i \kappa(\mathbf{q}, \mathbf{k}_i) \|\mathbf{c} - \mathbf{v}_i\|^2. \quad (12.4)$$

Setting its gradient to zero gives $\sum_i \kappa_i(\mathbf{c}^* - \mathbf{v}_i) = \mathbf{0}$, so $\mathbf{c}^* = \sum_i \kappa_i \mathbf{v}_i / \sum_i \kappa_i$. With $\kappa(\mathbf{q}, \mathbf{k}) = \exp(\mathbf{q}^\top \mathbf{k} / \sqrt{d})$, that is exactly scaled dot-product softmax attention. The kernel supplies nonnegative weights; the local-constant model is what makes their solution an average. That distinction preserves this chapter's line between a weighted *combination* and a weighted *average*.

i This lens is now a research program

The regression view you have just built is not only a teaching bridge. Work from 2024–26 uses a forward pass as **test-time regression**: the prefix supplies key–value training pairs, the current token supplies a query, and a memory operator fits or updates a predictor before answering. Test-Time Training, Delta-style fast weights, Titans, MesaNet, and Wang, Yang, Vidal and colleagues' *Beyond Test-Time Memory* differ in which part they change: the learned views of a token, the weighting of its history, the model or regularizer, or the online solver.

That shared objective is a design language, not a claim that the architectures are equivalent. Softmax attention performs a query-dependent local fit while retaining its key–value rows. Other methods compress the history into a fixed-size parametric state and accept a different statistical contract. The [test-time-regression interlude](#) derives that price list from Equation 12.4 rather than opening an architecture zoo.

Now let us change the names. Locations $\mathbf{x}_i$ become **keys** $\mathbf{k}_i$, observed responses become **values** $\mathbf{v}_i$, and $\mathbf{q}$ remains the **query**. The formula does not change. Here is the whole operator in NumPy. The shapes come before the multiplication: queries are (m, d) , keys are (n, d) , values are (n, p) , and the returned mixture is (m, p) .

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `row_softmax` and `gaussian_attention`.
 3. Fixed Gaussian attention, from scores to mixtures.
-

CODE

```
from __future__ import annotations

import matplotlib.pyplot as plt
import numpy as np
import torch
```



```

# [1]
torch.manual_seed(6050)
np.set_printoptions(precision=4, suppress=True)

# [2]
def row_softmax(scores: np.ndarray) -> np.ndarray:
    """Normalize each row after a stability-preserving shift."""
    shifted = scores - scores.max(axis=1, keepdims=True)
    weights = np.exp(shifted)
    return weights / weights.sum(axis=1, keepdims=True)

def gaussian_attention(
    queries: np.ndarray,
    keys: np.ndarray,
    values: np.ndarray,
    bandwidth: float,
) -> tuple[np.ndarray, np.ndarray]:
    """Fixed Gaussian attention for Q:(m,d), K:(n,d), V:(n,p)."""
    if not np.isfinite(bandwidth) or bandwidth <= 0:
        raise ValueError("bandwidth must be positive and finite")
    if queries.ndim != 2 or keys.ndim != 2 or values.ndim != 2:
        raise ValueError("queries, keys, and values must be matrices")
    if queries.shape[1] != keys.shape[1] or keys.shape[0] != values.shape[0]:
        raise ValueError("Q/K feature dimensions and K/V row counts must agree")

    distances2 = ((queries[:, None, :] - keys[None, :, :]) ** 2).sum(axis=2)
    log_scores = -distances2 / (2.0 * bandwidth**2) # (m, n)
    weights = row_softmax(log_scores) # (m, n)
    return weights @ values, weights # (m, p), (m, n)

keys3 = np.array([[1.0], [3.0], [5.0]])
values3 = np.array([[1.5], [2.8], [1.8]])
query3 = np.array([[3.5]])
pred3, weights3 = gaussian_attention(query3, keys3, values3, bandwidth=0.6)
# [3]
print("weights:", weights3[0])
print(f"prediction: {pred3.item():.4f}")

```

```

weights: [0.0002 0.9413 0.0585]
prediction: 2.7412

```

The same code makes the OLS distinction concrete:

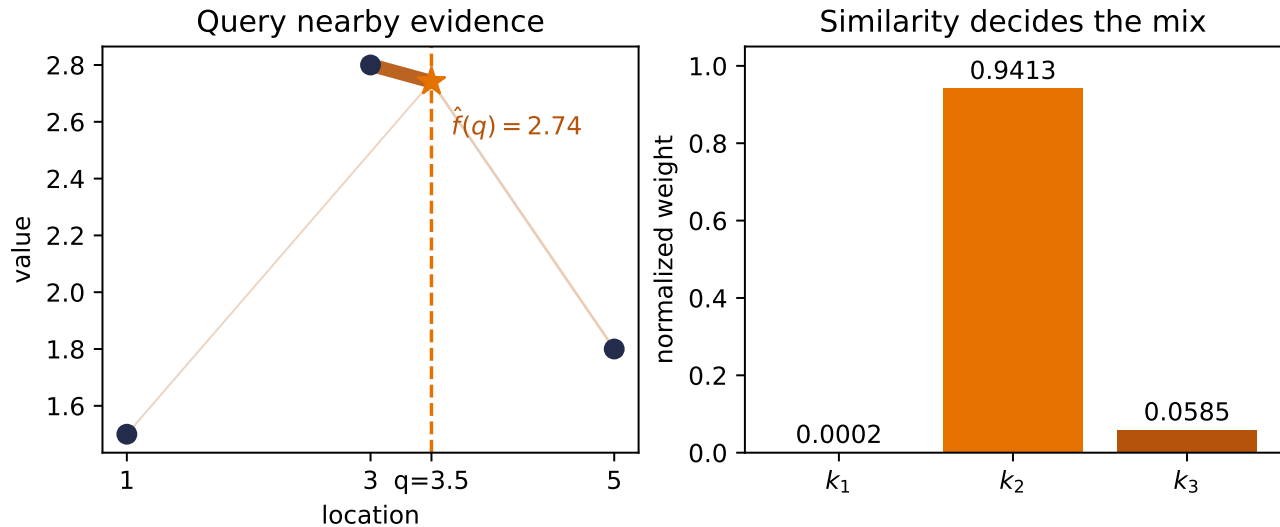


Figure 12.1.: A fixed Gaussian lookup. The bars give the normalized weights, and the connecting lines show which observations feed the query. The query lands close to the second location, so its response dominates the mixture; no similarity function was learned.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Signed OLS weights versus convex kernel weights.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
ols_keys = np.array([-2.0, -1.0, 0.0, 1.0, 2.0])
ols_query = 1.5
design = np.column_stack([np.ones_like(ols_keys), ols_keys])
query_design = np.array([1.0, ols_query])
# [2]
ols_weights = query_design @ np.linalg.solve(design.T @ design, design.T)

witness_values = np.array([[1.0], [0.0], [0.0], [0.0], [0.0]])
nw_pred, nw_weights = gaussian_attention(
    np.array([ols_query]), ols_keys[:, None], witness_values, bandwidth=1.0
)
# [3]
print("OLS weights:", ols_weights, "sum:", ols_weights.sum())
print("NW weights:", nw_weights[0], "sum:", nw_weights.sum())
print(f"witness predictions: OLS {ols_weights @ witness_values[:, 0]:.4f}, "
      f"NW {nw_pred.item():.4f}; target range [0, 1]")
```

OLS weights: [-0.1 0.05 0.2 0.35 0.5] sum: 1.0
 NW weights: [0.001 0.0206 0.152 0.4132 0.4132] sum: 1.0
 witness predictions: OLS -0.1000, NW 0.0010; target range [0, 1]

i Which kernels make a convex average?

Only nonnegative smoothing kernels make this convex average. A Gaussian gives positive weights; a compact-support kernel may give zeros. Other objects also bear the name *kernel*. Equation 12.3 needs nonnegative numerators and a positive denominator.

12.3. Chapter 2's scores-to-weights machine

Now harvest the second promise. Chapter 2 introduced softmax as the machine that turns arbitrary scores into nonnegative weights summing to one. Nadaraya–Watson has exactly that form. For any strictly positive kernel,

$$\alpha_i(\mathbf{q}) = \frac{\kappa_h(\mathbf{q}, \mathbf{k}_i)}{\sum_j \kappa_h(\mathbf{q}, \mathbf{k}_j)} = \text{softmax}_i(\log \kappa_h(\mathbf{q}, \mathbf{k}_i)). \quad (12.5)$$

The **score** is the log-kernel. With the Gaussian,

$$a_i(\mathbf{q}) = -\frac{\|\mathbf{q} - \mathbf{k}_i\|^2}{2h^2}.$$

This also makes the temperature connection exact. If the unscaled score is $s_i = -\|\mathbf{q} - \mathbf{k}_i\|^2$, then

$$\boldsymbol{\alpha}(\mathbf{q}) = \text{softmax}\left(\frac{\mathbf{s}}{\tau}\right), \quad \tau = 2h^2. \quad (12.6)$$

Small bandwidth $\rightarrow$ low temperature $\rightarrow$ sharp lookup. Large bandwidth $\rightarrow$ high temperature $\rightarrow$ diffuse averaging. The factor 2 belongs to this score convention; if we had defined $s_i = -\|\mathbf{q} - \mathbf{k}_i\|^2/2$, the temperature would be h^2 instead.

Notice the implementation never computes κ_h and then takes its logarithm. It computes the log-score directly, subtracts the largest score in each row, and only then exponentiates. That is Chapter 2's stable-softmax hygiene, now paying rent again.

12.4. Bandwidth: the honesty gate

The three-point example makes a narrow bandwidth look wonderful. That is not evidence that narrower is better. A tiny neighborhood follows every noisy bump; a huge neighborhood washes real structure away. Chapter 1 called this bias versus variance. Here the bandwidth controls the trade.

Because this is a synthetic laboratory, we can do something ordinary observed data does not permit: hold 60 key locations fixed, redraw the response noise 1,000 times, and compare every prediction with the known function

$$f(x) = \sin(1.5x) + 0.3 \cos(4x), \quad y_i = f(x_i) + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, 0.25^2).$$

The key locations are seeded uniform draws on $[-3, 3]$. The 241 query locations lie on $[-2.9, 2.9]$ and share no exact coordinate with a key. Holding the design fixed lets us separate squared bias from variance over independent noise draws. The population identity

$$\text{MSE} = \text{bias}^2 + \text{variance}$$

has no irreducible-noise term because we evaluate against the noiseless f ; predicting a fresh noisy response would add the irreducible variance 0.25^2 to every row. In our finite 1,000-world ensemble, the analogous empirical decomposition closes to floating-point roundoff.

PLAN

1. Define the reusable `truth` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Fixed-design bias/variance sweep over 1,000 noisy worlds.
 4. Report or visualize the measured result.
-

CODE

```
# [1]
def truth(x: np.ndarray) -> np.ndarray:
    return np.sin(1.5 * x) + 0.3 * np.cos(4.0 * x)

# [2]
data_rng = np.random.default_rng(605012)
keys = np.sort(data_rng.uniform(-3.0, 3.0, size=60))
observed_rng = np.random.default_rng(605013)
observed_values = truth(keys) + observed_rng.normal(0.0, 0.25, size=keys.size)
queries = np.linspace(-2.9, 2.9, 241)

mc_rng = np.random.default_rng(605014)
responses = truth(keys)[None, :] + mc_rng.normal(
    0.0, 0.25, size=(1_000, keys.size)
)
bandwidths = np.array([0.05, 0.08, 0.12, 0.18, 0.28, 0.42, 0.65, 1.00])

rows = []
# [3]
for bandwidth in bandwidths:
    _, weights = gaussian_attention(
        queries[:, None], keys[:, None], observed_values[:, None], bandwidth
    )
    predictions = responses @ weights.T # (world, query)
    mean_prediction = predictions.mean(axis=0)
```

```

squared_bias = np.mean((mean_prediction - truth(queries)) ** 2)
variance = np.mean(np.var(predictions, axis=0))
mse = np.mean((predictions - truth(queries)[None, :]) ** 2)
rows.append((bandwidth, 2 * bandwidth**2, squared_bias, variance, mse))

# [4]
print(" h      2h^2    bias^2  variance    MSE")
for row in rows:
    print(f"{row[0]:.2f}    {row[1]:.4f}    {row[2]:.4f}    "
          f"{row[3]:.4f}    {row[4]:.4f}")
best = rows[int(np.argmin([row[-1] for row in rows]))]
print(f"grid minimum: h={best[0]:.2f}, MSE={best[-1]:.4f}")

```

h	2h ²	bias ²	variance	MSE
0.05	0.0050	0.0119	0.0343	0.0462
0.08	0.0128	0.0101	0.0247	0.0348
0.12	0.0288	0.0090	0.0169	0.0259
0.18	0.0648	0.0127	0.0109	0.0236
0.28	0.1568	0.0304	0.0069	0.0373
0.42	0.3528	0.0747	0.0047	0.0794
0.65	0.8450	0.1721	0.0031	0.1753
1.00	2.0000	0.3142	0.0022	0.3164

grid minimum: h=0.18, MSE=0.0236

The variance falls at every step, from 0.0343 at $h = 0.05$ to 0.0022 at $h = 1.00$. Bias squared is not perfectly monotone at the sharp end of this sparse, random design; it bottoms at 0.0090 for $h = 0.12$, then rises to 0.3142 at $h = 1.00$. Their sum is the U-shaped quantity we care about. On this predeclared grid, $h = 0.18$ has the smallest MSE, 0.0236.

One noisy world shows what those numbers mean:

💡 Selecting h when the truth is hidden

The known black curve is a laboratory privilege. On real data, choose the bandwidth using a validation set, then report once on a sealed test set. Training error alone will usually reward an extremely narrow kernel for copying each noisy target back to itself. The generalization protocol from Chapter 6 still applies even when nothing is trained by gradient descent.

12.5. From one query to an attention matrix

Now write the operator in the vocabulary that the rest of Part IV will keep. Let

$$\mathbf{Q} \in \mathbb{R}^{m \times d}, \quad \mathbf{K} \in \mathbb{R}^{n \times d}, \quad \mathbf{V} \in \mathbb{R}^{n \times p}$$

hold m queries, n keys, and one p -dimensional value per key. The score and weight matrices are

$$S_{ri} = -\frac{\|\mathbf{q}_r - \mathbf{k}_i\|^2}{2h^2}, \quad \mathbf{A} = \text{rowsoftmax}(\mathbf{S}) \in \mathbb{R}^{m \times n}. \quad (12.7)$$

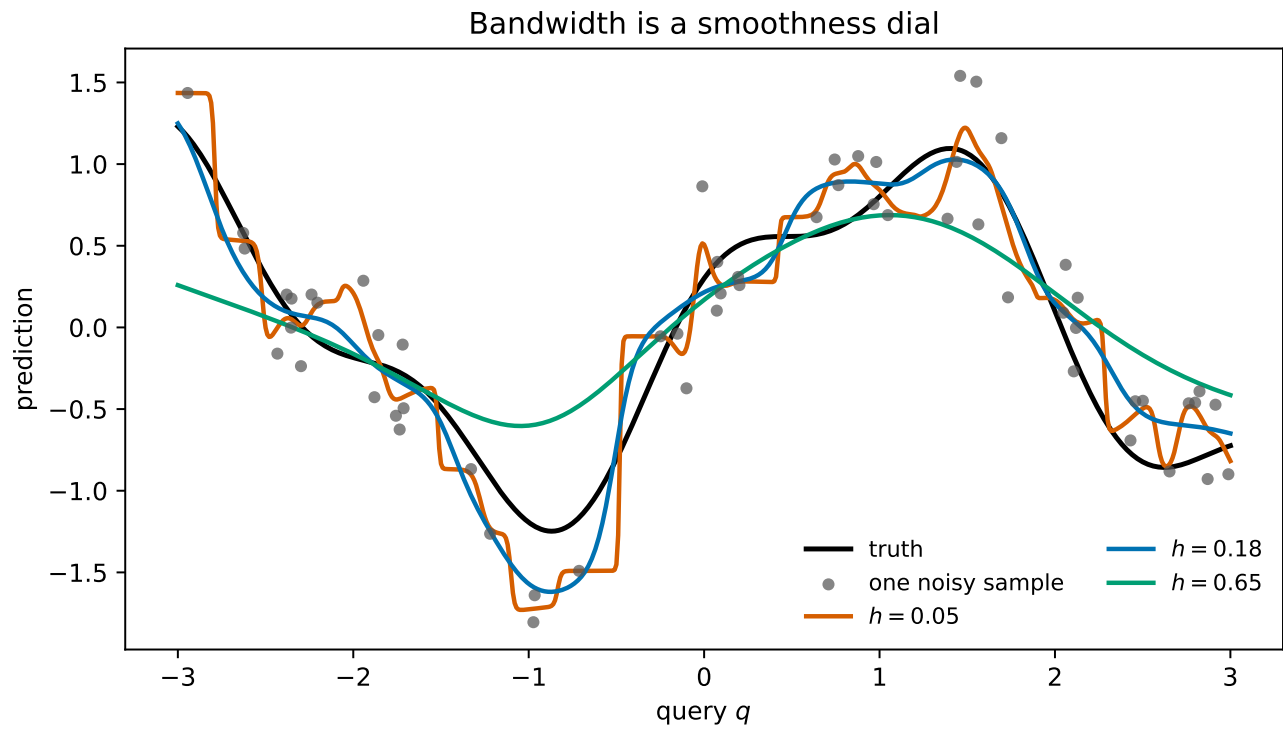


Figure 12.2.: One fixed noisy sample under three bandwidths. The narrow rule follows local noise, the middle rule tracks the signal, and the broad rule erases the function’s smaller-scale structure. The blue bandwidth is the grid minimum in the 1,000-world audit, not a hand-picked attractive curve.

Every row of $\mathbf{A}$ sums to one; columns generally do not. The pooled outputs are one matrix multiplication,

$$\text{Attention}_h(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{A}\mathbf{V} \in \mathbb{R}^{m \times p}. \quad (12.8)$$

This is the book's first **attention-weight matrix**. It is not a learned alignment. It is the allocation imposed by Euclidean distance and the chosen bandwidth.

PLAN

1. Form the first fixed attention map and audit its shapes and row sums.
-

CODE

```
# [1]
map_queries = np.linspace(-2.9, 2.9, 31)
map_predictions, attention_map = gaussian_attention(
    map_queries[:, None], keys[:, None], observed_values[:, None], best[0]
)
print(f"Q {map_queries[:, None].shape}, K {keys[:, None].shape}, "
      f"V {observed_values[:, None].shape}, A {attention_map.shape}, "
      f"AV {map_predictions.shape}")
print("maximum row-sum error:",
      f"{np.max(np.abs(attention_map.sum(axis=1) - 1)):.2e}")
```

```
Q (31, 1), K (60, 1), V (60, 1), A (31, 60), AV (31, 1)
maximum row-sum error: 2.22e-16
```

Read the bright band as a moving magnifying glass. Each query row reallocates one unit of weight across the stored keys. Where keys are dense, that unit is divided among several neighbors. Where keys are sparse, one or two slots carry most of it. Changing h changes the width of the band, exactly as changing temperature changes the spread of a softmax distribution.

There is one numerical trap. A Gaussian is mathematically positive everywhere, but a distant query and small bandwidth can make every directly exponentiated kernel value underflow to zero. Normalizing those zeros produces 0/0. The log-score version still has a largest entry, so the shifted softmax remains finite. This is one of the stable- algorithm case studies reunited in Appendix C:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify the log-kernel identity and survive underflow.
-

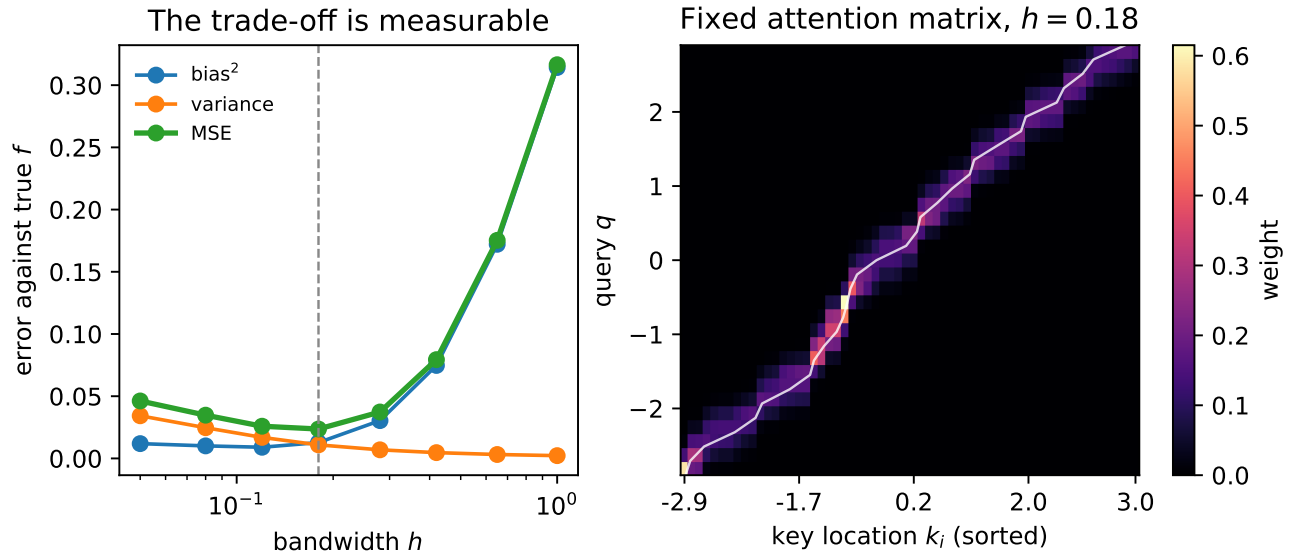


Figure 12.3.: Left: the fixed-design bias–variance audit. Right: the same Gaussian rule written as a 31-query by 60-key attention matrix at $h = 0.18$. Bright cells receive more of a row’s unit mass; the white trace marks the $k = q$ locus in sorted-key coordinates. The map is local and interpretable, but it was specified rather than learned.

CODE

```
# [1]
identity_query, identity_h = 0.37, 0.28
log_kernel = -0.5 * ((identity_query - keys) / identity_h) ** 2
direct_kernel = np.exp(log_kernel)
direct_weights = direct_kernel / direct_kernel.sum()
softmax_weights = row_softmax(log_kernel[None, :])[0]

far_log_kernel = -0.5 * ((1_000.0 - keys) / 0.1) ** 2
far_kernel = np.exp(far_log_kernel)
with np.errstate(divide="ignore", invalid="ignore"):
    naive_far_weights = far_kernel / far_kernel.sum()
stable_far_weights = row_softmax(far_log_kernel[None, :])[0]

# [2]
print("max |normalize(K)-softmax(log K)|:",
      f"{np.max(np.abs(direct_weights - softmax_weights)):.2e}")
print(f"far query: naive denominator={far_kernel.sum():.1f}, "
      f"all finite={np.all(np.isfinite(naive_far_weights))}")
print(f"stable: sum={stable_far_weights.sum():.1f}, "
      f"nearest key={keys[np.argmax(stable_far_weights)]:.4f}")
```

```
max |normalize(K)-softmax(log K)|: 2.78e-17
far query: naive denominator=0.0, all finite=False
stable: sum=1.0, nearest key=2.9886
```


⚠ Trap: similarity weights are not uncertainty probabilities

The entries of $\mathbf{A}$ are probability-like allocations: nonnegative and row-normalized. They are not calibrated probabilities that the corresponding key is “correct,” and a sharp row is not automatically confidence. It may simply mean the bandwidth is small.

12.6. Return the fixed operator to the memory bank

The regression names now return to Chapter 11’s memory bank. Each one names a different job:

job	kernel regression	sequence memory preview
query	location $\mathbf{q}$ where we need an answer	what the decoder needs now
keys	stored coordinates $\mathbf{k}_i$	descriptors used to match encoder memory slots
values	observed responses $\mathbf{v}_i$	content retrieved from those slots
weights	normalized fixed similarities $\alpha_i(\mathbf{q})$	how much each slot contributes
output	$\sum_i \alpha_i \mathbf{v}_i$	a query-specific context vector

Queries specify what is needed; keys provide what to compare against; values provide what to retrieve. In the simplest sequence model, the same encoder state can play both roles, just as our one-dimensional example stores a location and its response together. Later, learned projections will let keys and values carry different representations.

Chapter 1 also planted one more similarity primitive, the dot product:

$$\mathbf{q}^\top \mathbf{k} = \|\mathbf{q}\| \|\mathbf{k}\| \cos \theta.$$

It is pure angular similarity only when the norms are controlled. If $\mathbf{q} = (1, 0)$, $\mathbf{k}_1 = (1, 0)$, and $\mathbf{k}_2 = (2, 0)$, both keys point in the same direction as the query, but their dot products are 1 and 2. Magnitude participates. That is not a defect; it is a reminder that the score defines what “similar” means.

Our score today is a hand-written negative squared Euclidean distance, and h is selected rather than learned. The operator stores the data, computes a fixed map, and pools the values. It shares attention’s normalized similarity-pooling algebra, but it has no learned compatibility function.

Now the Part II rhyme returns. Chapter 7 ended with a fixed image kernel and asked what would happen if the filter became learnable. Chapter 8 answered with a CNN. We end Part IV’s opening chapter with the same question:

💡 What if the similarity itself were learnable?

The memory bank is ready. Query, keys, values, scores, row-wise softmax, and weighted pooling are ready. Only one hand-designed piece remains: the rule that decides whether a query and key belong together. Chapter 13 makes that rule learnable, then returns to Chapter 11’s date task so

the decoder can show us where it looked.

i Check yourself

Close the book for one minute and reconstruct the weighted prediction.

- How do scores become nonnegative weights that sum to one?
- What does bandwidth change in a kernel smoother?
- Which component is still hand designed at the end of this chapter?

12.7. Okay, so — attention is normalized memory mixing

1. **Chapter 1’s linear prediction already mixed training targets.** OLS gives $\hat{f}(\mathbf{q}) = \boldsymbol{\lambda}(\mathbf{q})^\top \mathbf{y}$, but its weights may be signed, so the prediction can leave the targets’ convex hull.
2. **Nadaraya–Watson turns locality into a convex mixture.** A nonnegative kernel scores query–key similarity; normalization makes the weights sum to one; their weighted values form the prediction.
3. **Bandwidth is temperature in geometric clothing.** For Gaussian distance scores, $\tau = 2h^2$: narrow is sharp and variable, wide is diffuse and biased. Our fixed audit reached its grid-minimum MSE of 0.0236 at $h = 0.18$.
4. **Chapter 2’s softmax is the normalization exactly.** Positive-kernel weights equal `softmax(log kernel)`. Computing log-scores directly and shifting each row avoids the distant-query 0/0 trap.
5. **Fixed attention is AV.** Queries score keys, row-softmax produces an $m \times n$ allocation matrix, and that matrix pools the values. The algebra is complete; learning the score is the next chapter’s move.

Sources and further reading

- [Nadaraya, *On Estimating Regression*](#): the 1964 two-page paper introducing one side of the estimator now bearing Nadaraya and Watson’s names.
- [Watson, *Smooth Regression Analysis*](#): the independent 1964 development of smooth regression by local weighting.
- [Bahdanau, Cho, & Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*](#): the neural-attention step that replaces one fixed context with learned soft alignment; Chapter 13 performs that harvest.
- [Wang, Shi, & Fox, *Test-Time Regression: A Unifying Framework for Designing Sequence Models with Associative Memory* \(2025\)](#): develops test-time regression as a common design language for sequence memory.
- [Sun et al., *Learning to \(Learn at Test Time\): RNNs with Expressive Hidden States* \(2024 preprint; 2025 proceedings\)](#): treats a sequence model’s hidden state as a learner updated by a self-supervised objective during the forward pass.
- [Schlag, Irie, & Schmidhuber, *Linear Transformers Are Secretly Fast Weight Programmers* \(2021\)](#): connects linear attention to fast-weight updates and the delta rule.
- [Behrouz et al., *Titans: Learning to Memorize at Test Time* \(2025\)](#): adds a learned long-term memory updated at test time.

- von Oswald et al., *MesaNet: Sequence Modeling by Locally Optimal Test-Time Training* (2025): derives sequence layers from locally solved test-time objectives.
- Wang, Yang, Vidal et al., *Beyond Test-Time Memory: State-Space Optimal Control for LLM Reasoning* (2026): extends the regression framing toward test-time control; the epilogue treats that proposal as a frontier bet rather than settled equivalence.

Exercises

1. **(Pencil.)** Prove that Equation 12.3 is a convex combination when $\kappa_h \geq 0$ and its denominator is positive. Then prove that a scalar prediction lies in $[\min_i v_i, \max_i v_i]$. Which step fails for the OLS weights in Equation 12.1?
2. **(Pencil.)** For keys $(-2, -1, 0, 1, 2)$, derive the five OLS target weights at $q = 1.5$ without using code. Verify that they sum to one, identify the negative weight, and construct a second target vector for which OLS leaves the target range.
3. **(Code.)** Split a fresh noisy sample from this chapter’s function into training, validation, and sealed test sets. Select h on validation data, report the test MSE once, and compare your selected curve with the synthetic-truth grid minimum above. Explain why the two procedures need not select the same bandwidth.
4. **(Pencil.)** (a) Derive Equation 12.6. **(Code.)** (b) Use keys $[0, 0, 1]$, query 0, and successively smaller positive bandwidths. What weights does stable softmax assign to the exact nearest-key tie, and why is $h = 0$ still invalid?
5. **(Code.)** Extend `gaussian_attention` to two-dimensional keys whose coordinates have very different units. Compare the attention map before and after standardizing each feature using training-set statistics. What does this reveal about the metric hidden inside a fixed kernel?
6. **(Pencil.)** Differentiate Equation 12.4 with respect to the vector $\mathbf{c}$. Solve the stationarity equation, then substitute $\kappa(\mathbf{q}, \mathbf{k}_i) = \exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})$. Why would the stronger claim “an unrestricted M with no regularizer must average” be false?

13. Attention: Making the Kernel Learnable

Chapter 12 built the whole attention operator except for one piece. It had a query, a bank of keys and values, a score for every query–key pair, softmax weights, and a weighted read. But Euclidean distance decided what counted as similar. The task could tune the bandwidth; it could not invent a better notion of relevance.

Chapter 11 left us a more pointed promise: **hard address, learned content**. An embedding can learn what lives at row 17, but fetching row 17 is still a rigid indexing operation. Chapter 2 told us how to soften that hard choice. Replace the single address with a differentiable distribution over addresses, let the loss send blame through every score, and learn which memory locations deserve weight. The behavior is now concrete: compare one request with every stored candidate, normalize those comparison scores, and blend the stored values with the resulting weights. Chapter 2 gave us those ingredients before we needed their name. This content-dependent weighted read is **attention**.

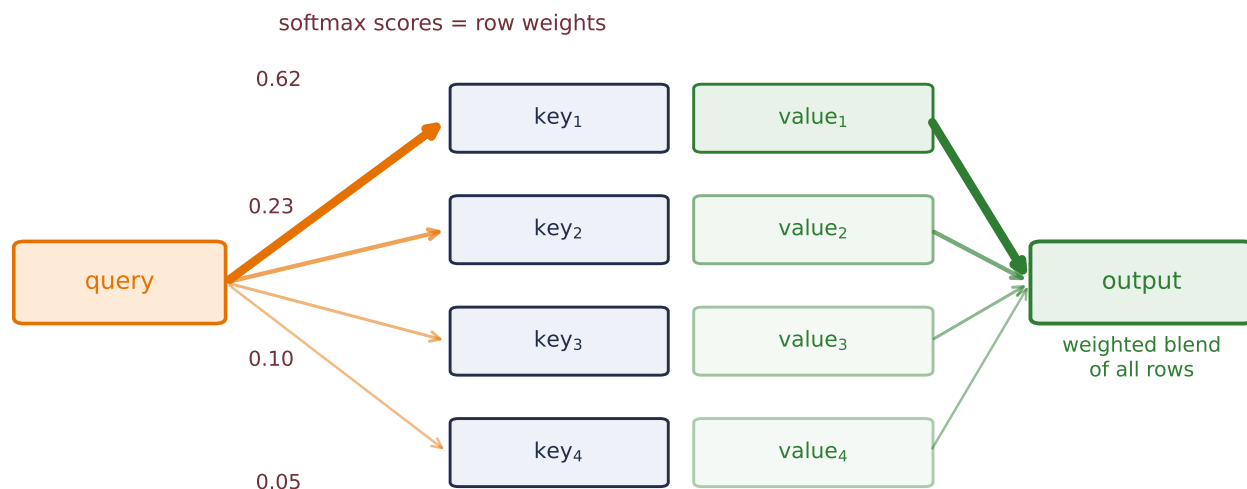


Figure 13.1.: The lookup table, softened. A hard lookup would follow exactly one arrow to one row. Attention scores the query against every key, softmax turns the scores into row weights (arrow thickness; the numbers here are illustrative), and the output is the weighted blend of all values. Every arrow is differentiable, so the loss can teach the scoring itself — this diagram is the entire chapter.

The result is visible in the date task before we name all its parts. For the fixed validation source

```
may 17, 1971 -> 1971-05-17
```

the trained decoder emits the year first even though those four characters sit at the end of the source. Across the fixed 400-example validation audit, its first four attention rows place **97.5%** of their mass

on the four contextual encoder states indexed by the source-year characters. When the example above emits its second character, **9**, the largest weight is 0.856 on the state indexed by the source’s 9; when it emits **7**, the largest weight is 0.985 on the state indexed by the 7 in the year region. Those weights were not supervised as alignments. The date loss taught the model where useful information lived.

13.1. The scores-to-weights machine, one more time

Let the encoder retain its full memory

$$\mathbf{H} = [\mathbf{h}_1, \dots, \mathbf{h}_n]^\top \in \mathbb{R}^{n \times d_e}.$$

At decoder step t , use the previous decoder hidden state $\mathbf{s}_{t-1} \in \mathbb{R}^{d_d}$ as the query. A learned compatibility function a_θ produces one score per source position:

$$e_{t,i} = a_\theta(\mathbf{s}_{t-1}, \mathbf{h}_i).$$

For each padded batch item b , let $M_{b,i} = 0$ at real source positions and $M_{b,i} = -\infty$ at padding. Suppress the batch index below. Then

$$\alpha_{t,i} = \text{softmax}_i(e_{t,i} + M_i), \quad \mathbf{c}_t = \sum_{i=1}^n \alpha_{t,i} \mathbf{h}_i. \quad (13.1)$$

This is Chapter 12’s fixed matrix $\mathbf{AV}$ with a learned score. It is also Chapter 2’s **scores** $\rightarrow$ **weights** machine in exactly the form that chapter advertised. A hard source-position choice would use argmax and one-hot indexing. The soft weights move continuously when the scores move, so backpropagation can tell the scorer how each valid address affected the loss. We have **softened the hard**.

Because the query comes from the decoder and the memory comes from the encoder, this is **cross-attention**. It bridges two sequences. The model still initializes the decoder from the encoder’s final state, but that fixed bridge is no longer the only path from source to output: every decoder step receives its own context $\mathbf{c}_t$. Chapters 10 and 11 called the old handoff a **finite-state bottleneck**. Chapter 12 kept the memory bank; this chapter completes that relay by learning its access rule.

Chapter 8 made one other promise. A CNN buys global sight by stacking local operations until its receptive field spans the input. One cross-attention read can score every encoder position in a single layer. It buys global access directly, paying for all query–key comparisons and, for now, for the RNNs that created those states sequentially.

13.2. Additive attention: learn the comparison itself

The most literal answer to “make similarity learnable” is a small neural network. Choose an alignment width d_a . For one decoder query and one encoder state,

$$e_{t,i} = \mathbf{v}_a^\top \tanh(\mathbf{W}_q \mathbf{s}_{t-1} + \mathbf{W}_k \mathbf{h}_i), \quad (13.2)$$

where

13. Attention: Making the Kernel Learnable

$$\mathbf{W}_q \in \mathbb{R}^{d_a \times d_d}, \quad \mathbf{W}_k \in \mathbb{R}^{d_a \times d_e}, \quad \mathbf{v}_a \in \mathbb{R}^{d_a}.$$

The two projections put objects of different native sizes into one alignment space. The tanh creates match features, and $\mathbf{v}_a$ selects which of those features matter for the scalar score. The same parameters are reused for every decoder step t and source position i . Remember Chapter 7’s sliding filter: one learned rule, applied everywhere. Attention shares a compatibility rule across pairs.

For a batch, let $\mathbf{S}_{t-1} \in \mathbb{R}^{B \times 1 \times d_d}$ collect the queries. The row-major shapes make the broadcasting explicit:

$$\mathbf{S}_{t-1} \mathbf{W}_q^\top : (B, 1, d_a), \quad \mathbf{H} \mathbf{W}_k^\top : (B, n, d_a).$$

Their sum produces (B, n, d_a) match features. Reduction by $\mathbf{v}_a$ produces (B, n) scores, softmax runs over the n source positions, and the weighted read returns a (B, d_e) context. That last axis choice is not bookkeeping: softmax over the wrong dimension answers the wrong question.

We freeze one simple parameter snapshot so we can watch the arithmetic. With identity projections and an all-ones selector, three source states receive scores (1.015, 1.562, 1.124) and therefore weights (0.260, 0.450, 0.290):

PLAN

1. Compute the three-key additive scores and softmax weights.
-

CODE

```
from __future__ import annotations

import math
import random

import matplotlib.pyplot as plt
from matplotlib.patches import FancyBboxPatch
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# [1]
torch.manual_seed(6050)

query_snapshot = np.array([0.5, 0.5, 0.2, -0.1])
key_snapshot = np.array([
    [-0.3, 0.1, 0.2, 0.0],
    [0.6, 0.5, 0.1, -0.2],
    [0.0, -0.4, 0.3, 0.2],
```

```

])
snapshot_scores = np.tanh(query_snapshot + key_snapshot).sum(axis=1)
snapshot_weights = np.exp(snapshot_scores - snapshot_scores.max())
snapshot_weights /= snapshot_weights.sum()

```

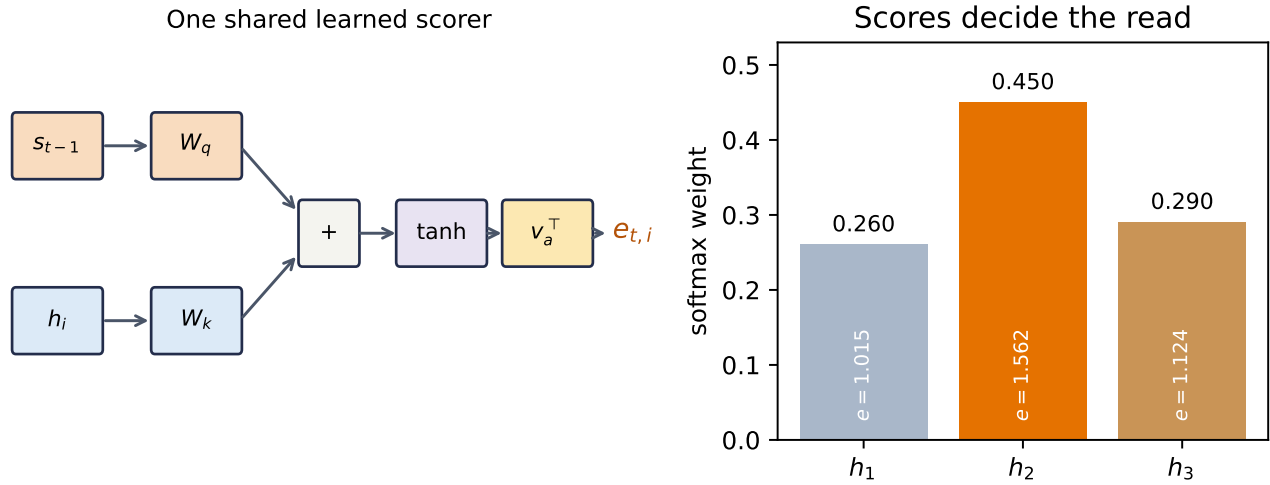


Figure 13.2.: Additive attention learns a compatibility function. Query and key enter separate projections, their alignment features interact through tanh, and a learned selector reduces those features to one score. The right panel freezes one parameter setting to show the resulting scores-to-weights calculation; it is arithmetic, not a trained result.

i “Learned kernel” is an analogy

Exponentiating a learned score and normalizing it reproduces the Nadaraya–Watson algebra. It does not make every attention score a classical kernel. Separate query and key projections can make $a_\theta(q, k)$ asymmetric and not positive semidefinite. What survives exactly is the operational pattern: compare $\rightarrow$ normalize $\rightarrow$ mix.

13.3. Scaled dot product: learn the spaces, simplify the score

For m decoder queries, additive attention is vectorized, but it forms a (B, m, n, d_a) intermediate, applies an elementwise tanh, and reduces it. Dense matrix multiplication has a simpler computational path. Chapter 1 introduced the dot product as a similarity score against a template. Here the key is the template, and training learns the space in which that comparison is useful.

First walk one four-dimensional query past three keys. Its raw dot products are $(1.10, -0.60, 0.15)$. Since $\sqrt{4} = 2$, scaling gives $(0.55, -0.30, 0.075)$, and softmax gives $(0.488, 0.209, 0.303)$:

 PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Run one query through dot product, scaling, and softmax.
-

CODE

```
# [1]
query_walk = torch.tensor([0.9, -0.3, 0.2, 0.6])
key_walk = torch.tensor([
    [0.8, -0.2, 0.1, 0.5],
    [-0.5, 0.9, 0.0, 0.2],
    [0.3, 0.0, 0.6, -0.4],
])
raw_walk = key_walk @ query_walk
scaled_walk = raw_walk / math.sqrt(query_walk.numel())
weight_walk = torch.softmax(scaled_walk, dim=0)
# [2]
print("raw scores:", raw_walk.numpy())
print("scaled scores:", scaled_walk.numpy())
print("weights:", weight_walk.numpy(), "sum:", weight_walk.sum().item())
```

```
raw scores: [ 1.0999999 -0.6          0.15       ]
scaled scores: [ 0.54999995 -0.3          0.075       ]
weights: [0.4879715  0.20856632 0.3034622 ] sum: 1.0
```

Now let raw decoder features $\mathbf{D} \in \mathbb{R}^{B \times m \times d_d}$ and encoder memory $\mathbf{H} \in \mathbb{R}^{B \times n \times d_e}$ enter learned projections:

$$\begin{aligned}\mathbf{Q} &= \mathbf{D}\mathbf{W}_Q, & \mathbf{W}_Q &\in \mathbb{R}^{d_d \times d_k}, \\ \mathbf{K} &= \mathbf{H}\mathbf{W}_K, & \mathbf{W}_K &\in \mathbb{R}^{d_e \times d_k}, \\ \mathbf{V} &= \mathbf{H}\mathbf{W}_V, & \mathbf{W}_V &\in \mathbb{R}^{d_e \times d_v}.\end{aligned}$$

The resulting tensors have shapes

$$\mathbf{Q} \in \mathbb{R}^{B \times m \times d_k}, \quad \mathbf{K} \in \mathbb{R}^{B \times n \times d_k}, \quad \mathbf{V} \in \mathbb{R}^{B \times n \times d_v}.$$

Only the projected query and key widths must agree. The raw encoder and decoder widths need not. Let $\mathbf{M} \in \mathbb{R}^{B \times 1 \times n}$ broadcast across queries. **Scaled dot-product attention** is

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} + \mathbf{M}\right) \mathbf{V}, \quad (13.3)$$

where $\mathbf{K}^\top$ transposes the last two axes within each batch. Softmax runs over the last, key-position axis. The hot path is two dense products: $\mathbf{Q}\mathbf{K}^\top$ and $\mathbf{A}\mathbf{V}$. Additive attention remains useful and

accelerator-compatible; scaled dot product simply maps more directly to the matrix operations modern numerical libraries optimize heavily. Appendix C turns that intuition into a conditional performance model—dense products can earn high arithmetic intensity through reuse, but shapes, data movement, implementation, and hardware decide the actual regime (Appendix C).

Luong and colleagues compared unscaled dot, general, and concat scores in recurrent translation. Vaswani and colleagues later introduced the $1/\sqrt{d_k}$ scaling in the Transformer formulation. The mechanisms belong in one conceptual family, not a single inevitable historical derivation.

The dot product is not magically semantic. $\mathbf{W}_Q$ and $\mathbf{W}_K$ learn which features to compare, including their magnitudes, while the surrounding RNN has already built nonlinear sequence features. In Chapter 14, a feedforward network will help provide that nonlinear feature transformation after recurrence disappears.

13.3.1. Why divide by $\sqrt{d_k}$?

Suppose, for the moment, that projected coordinates q_ℓ and k_ℓ are independent, mean zero, and variance one, and that their products are independent across coordinates. Then

$$\mathbb{E}[\mathbf{q}^\top \mathbf{k}] = 0, \quad \text{Var}(\mathbf{q}^\top \mathbf{k}) = \sum_{\ell=1}^{d_k} \text{Var}(q_\ell k_\ell) = d_k.$$

The score’s standard deviation grows as $\sqrt{d_k}$. Dividing by that quantity returns the idealized variance to one. This is a variance-control rationale, not a promise that trained projections remain independent or unit variance.

Let us test both the variance and what softmax sees. For each width, draw 20,000 queries and 16 independent keys per query:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Audit score variance and softmax concentration versus width.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
scaling_gen = torch.Generator().manual_seed(6050132)
scaling_rows = []
# [2]
for width in (8, 32, 128, 512):
    raw_parts, raw_max_parts, scaled_max_parts = [], [], []
    for _ in range(20_000 // 500):
        q = torch.randn(500, 1, width, generator=scaling_gen)
        k = torch.randn(500, 16, width, generator=scaling_gen)
        raw = (q * k).sum(-1)
```

```

raw_parts.append(raw.reshape(-1))
raw_max_parts.append(torch.softmax(raw, dim=-1).max(-1).values)
scaled_max_parts.append(
    torch.softmax(raw / math.sqrt(width), dim=-1).max(-1).values
)
raw_scores = torch.cat(raw_parts)
scaling_rows.append((
    width,
    raw_scores.var(unbiased=True).item(),
    (raw_scores / math.sqrt(width)).var(unbiased=True).item(),
    torch.cat(raw_max_parts).mean().item(),
    torch.cat(scaled_max_parts).mean().item(),
))

# [3]
print("d_k    raw var    scaled var    raw max-w    scaled max-w")
for row in scaling_rows:
    print(f"{row[0]:3d}    {row[1]:8.3f}    {row[2]:10.3f}    "
          f"{row[3]:9.3f}    {row[4]:12.3f}")

```

d_k	raw var	scaled var	raw max-w	scaled max-w
8	8.023	1.003	0.564	0.241
32	32.142	1.004	0.779	0.246
128	127.985	1.000	0.889	0.246
512	510.347	0.997	0.944	0.247

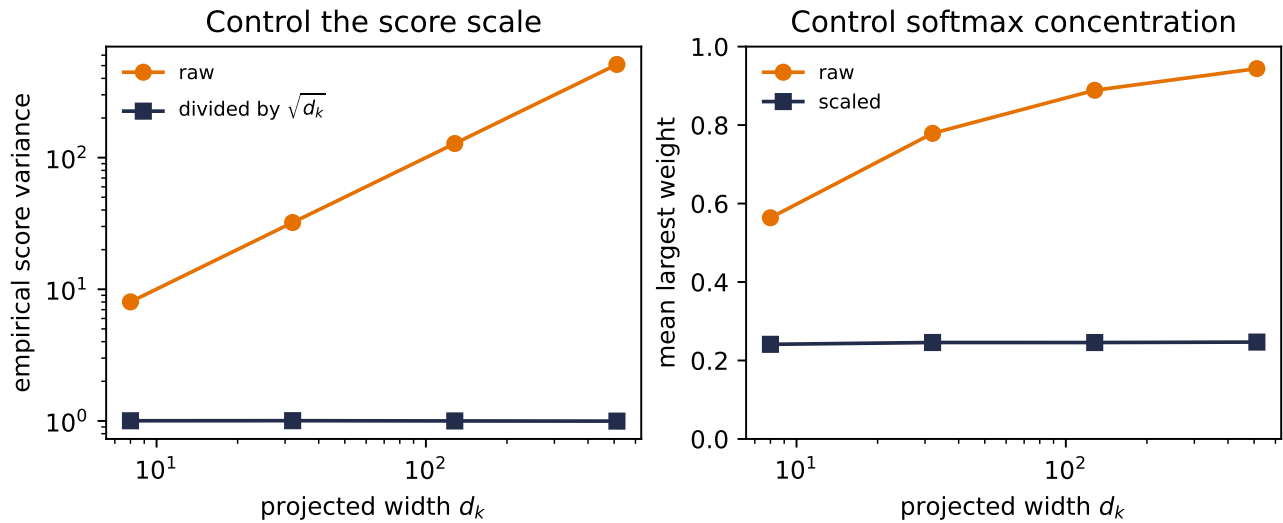


Figure 13.3.: The scaling argument under its stated random-feature assumptions. Left: raw dot-product variance grows with width, while division by the square root of the width keeps it near one. Right: without scaling, a 16-key softmax becomes nearly one-hot as width grows; scaling keeps its mean largest weight near one quarter (0.24 in this draw). These are seeded synthetic draws about concentration, not a claim about every trained layer’s gradients.

Raw variance rises from 8.02 to 510.35 as d_k grows from 8 to 512. Scaled variance stays between

0.997 and 1.004. Over the same range, the average largest unscaled softmax weight climbs from 0.564 to 0.944; the scaled value stays near 0.24. This is Chapter 2's **temperature dial** returning with a dimension-dependent setting. Scaling corrects the dimension-induced spread; it does not normalize the vectors.

13.4. Mask before softmax, and mask the right thing

Chapter 11 showed that padding can poison an encoder's final state. Attention creates a third place to enforce sequence lengths: the score matrix. A padded key must not receive a merely mediocre score. It must be excluded before normalization. Setting its score to zero is wrong because $\exp(0) = 1$ still earns weight.

Here is an independently written scaled-dot operator. It accepts already projected queries, keys, and values, then audits a batch whose second example has two padded keys:

PLAN

1. Define the reusable `scaled_dot_attention` helper.
 2. Run and audit scaled dot-product attention with a source-padding mask.
-

CODE

```
# [1]
def scaled_dot_attention(
    queries: torch.Tensor,
    keys: torch.Tensor,
    values: torch.Tensor,
    key_is_valid: torch.Tensor,
) -> tuple[torch.Tensor, torch.Tensor]:
    """Attend with Q:(B,m,d_k), K:(B,n,d_k), V:(B,n,d_v)."""
    if queries.ndim != 3 or keys.ndim != 3 or values.ndim != 3:
        raise ValueError("Q, K, and V must be three-dimensional")
    if queries.shape[-1] != keys.shape[-1]:
        raise ValueError("projected Q and K widths must match")
    if keys.shape[:2] != values.shape[:2]:
        raise ValueError("K and V must have the same batch and key axes")
    if key_is_valid.shape != keys.shape[:2] or not key_is_valid.any(dim=1).all():
        raise ValueError("every example needs at least one valid key")

    scores = queries @ keys.transpose(-2, -1) / math.sqrt(keys.shape[-1])
    scores = scores.masked_fill(~key_is_valid[:, None, :], -torch.inf)
    weights = torch.softmax(scores, dim=-1)
    return weights @ values, weights

# [2]
mask_gen = torch.Generator().manual_seed(6050133)
```

13. Attention: Making the Kernel Learnable

```
Q_demo = torch.randn(2, 2, 4, generator=mask_gen)
K_demo = torch.randn(2, 4, 4, generator=mask_gen)
V_demo = torch.randn(2, 4, 3, generator=mask_gen)
valid_demo = torch.tensor([[True, True, True, True],
                           [True, True, False, False]])
context_demo, mask_weights = scaled_dot_attention(
    Q_demo, K_demo, V_demo, valid_demo
)
assert torch.allclose(
    mask_weights.sum(-1), torch.ones_like(mask_weights[..., 0])
)
assert torch.count_nonzero(mask_weights[1, :, 2:]) == 0
print("Q", tuple(Q_demo.shape), "K", tuple(K_demo.shape),
      "V", tuple(V_demo.shape), "A", tuple(mask_weights.shape),
      "AV", tuple(context_demo.shape))
print("maximum row-sum error:",
      f"{(mask_weights.sum(-1) - 1).abs().max().item():.2e}")
print("maximum padded weight:",
      f"{mask_weights[1, :, 2:].max().item():.1f}")
```

Q (2, 2, 4) K (2, 4, 4) V (2, 4, 3) A (2, 2, 4) AV (2, 2, 3)
maximum row-sum error: 0.00e+00
maximum padded weight: 0.0

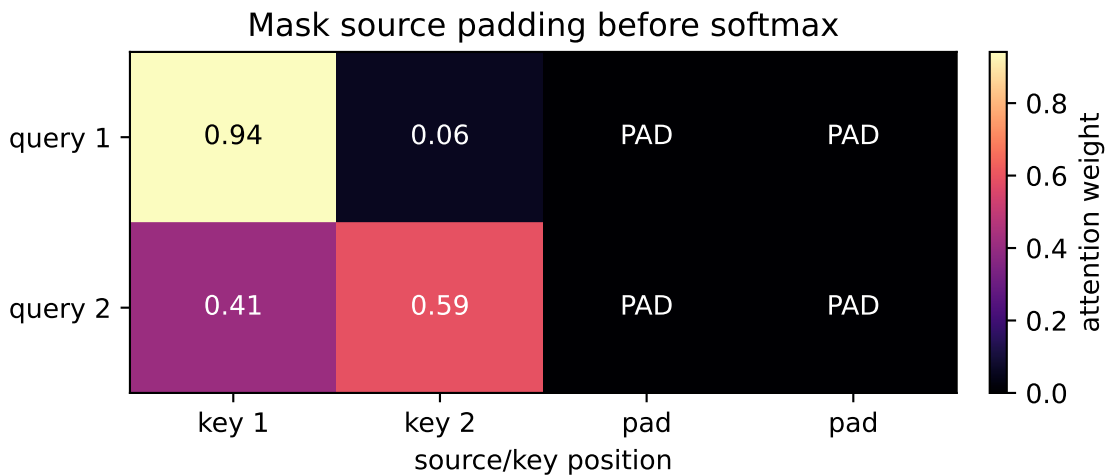


Figure 13.4.: A source-padding mask acts before softmax. The second example has only two real keys; its last two score cells are excluded and receive exactly zero weight. Cross-attention may read every real source position, so no causal triangle belongs here.

⚠️ Trap: three masks are not one mask

The target loss mask says which output positions should not be graded; it remains good general hygiene but is dormant for this fixed-length ISO target. The cross-attention padding mask says which source positions do not exist. A causal mask says which future target positions may not be seen. Our recurrent decoder generates one step at a time and the entire source is already available,

so the source-padding attention mask is the one exercised here. Chapter 14 will use causal masking inside decoder self-attention.

An all-masked row has no sensible distribution and would make softmax of all $-\infty$ undefined. Good hygiene is to assert that every example has at least one real key, as the function does above.

13.5. Return to Chapter 11's date task

The abstraction has earned a real rematch. We will change one architectural idea and keep the experimental contract fixed:

- the same date generator and duplicate-source rejection;
- the same 8,000/500/500 train/validation/test split;
- the same embedding width 32 and LSTM hidden width 128;
- the same Adam learning rate 10^{-3} , batch size 128, 25 epochs, teacher forcing, gradient clip 5, and seed 6050;
- the same first 400 unambiguous validation sources for every learning curve;
- one final scalar audit on the same 437 unambiguous test sources.

The generator and split below are deliberately byte-for-byte copies of Chapter 11. Changing them would end the running benchmark:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable `make_date` helper.
 3. Reconstruct Chapter 11's sealed date benchmark.
 4. Report or visualize the measured result.
-

CODE

```
import random, torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

# [1]
MONTHS = ["january", "february", "march", "april", "may", "june", "july",
          "august", "september", "october", "november", "december"]
DAYS = dict(zip(MONTHS, [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]))

# [2]
def make_date(rng: random.Random) -> tuple[str, str]:
    mo = rng.randrange(12)
    d = rng.randrange(1, DAYS[MONTHS[mo]] + 1)
    y = rng.randrange(1950, 2026)
```

```

iso = f"{y:04d}-{mo+1:02d}-{d:02d}"
style = rng.random()
if style < 0.35:
    src = f"{MONTHS[mo]} {d}, {y}"
elif style < 0.6:
    src = f"{d} {MONTHS[mo]} {y}"
elif rng.random() < 0.7:
    src = f"{mo+1:02d}/{d:02d}/{y}"          # US convention: mm/dd
else:
    src = f"{d:02d}/{mo+1:02d}/{y}"        # EU convention: dd/mm
return src, iso

rng = random.Random(6050)
pairs, seen_sources = [], set()
# [3]
while len(pairs) < 9000:
    pair = make_date(rng)
    if pair[0] not in seen_sources:
        seen_sources.add(pair[0])
        pairs.append(pair)
train_pairs = pairs[:8000]
valid_pairs = pairs[8000:8500]
test_pairs = pairs[8500:]
# [4]
print(*pairs[:4], sep="\n")
train_sources = {s for s, _ in train_pairs}
valid_sources = {s for s, _ in valid_pairs}
test_sources = {s for s, _ in test_pairs}
print("exact-source overlaps: "
      f"train/valid {len(train_sources & valid_sources)}, "
      f"train/test {len(train_sources & test_sources)}, "
      f"valid/test {len(valid_sources & test_sources)}")

PAD, BOS, EOS = 0, 1, 2          # special tokens first
SRC = sorted(set("".join(s for s, _ in pairs)))
TGT = sorted(set("".join(t for _, t in pairs)))
src_stoi = {c: i + 3 for i, c in enumerate(SRC)}
tgt_stoi = {c: i + 3 for i, c in enumerate(TGT)}
tgt_itos = {i: c for c, i in tgt_stoi.items()}
V_src, V_tgt = len(SRC) + 3, len(TGT) + 3
print(f"source vocab {V_src}, target vocab {V_tgt}")

```

('26 september 2024', '2024-09-26')

('april 7, 1962', '1962-04-07')

('12/15/1950', '1950-12-15')

('14 february 1997', '1997-02-14')

exact-source overlaps: train/valid 0, train/test 0, valid/test 0

source vocab 37, target vocab 14

Numeric dates can be ambiguous, so we retain the same unambiguous subsets as Chapter 11. Design

decisions and the heatmap use validation only. The test set remains a single number at the end.

13.5.1. Put the learned read inside the recurrent decoder

The packed encoder now returns two things: its final $(\mathbf{h}, \mathbf{c})$ state, which still initializes the decoder, and a padded memory tensor containing every real encoder output plus a validity mask that excludes the padded slots. At step t :

1. use decoder hidden state $\mathbf{s}_{t-1}$ as the query;
2. compute additive scores over the encoder memory and mask padding;
3. form the context $\mathbf{c}_t$;
4. concatenate $\mathbf{c}_t$ with the embedding of the previous target token;
5. advance one decoder LSTM step, then predict from $[\mathbf{s}_t; \mathbf{c}_t]$.

This timing follows the Bahdanau-style formulation. Luong-style variants often score with the current decoder state instead; both are valid, but silently mixing their indices makes an implementation impossible to audit.

For this date model, the shape ledger is concrete:

quantity	shape
previous hidden query	$(B, 128)$
encoder memory	$(B, T_{\text{src}}, 128)$
attention weights	(B, T_{src})
context	$(B, 128)$
token embedding joined with context	$(B, 1, 160)$
decoder output	$(B, 1, 128)$
output-head input $[\mathbf{s}_t; \mathbf{c}_t]$	$(B, 256)$

PLAN

1. Define the reusable helpers: `batchify`, `unambiguous`, and `LearnedAlignment`.
 2. Prepare the inputs and fixed settings for the example.
 3. Run a masked additive-attention LSTM decoder.
 4. Define the reusable helpers: `AttentiveSeq2Seq`, `greedy_batch`, and `exact_match`.
-

CODE

```
# [1]
def batchify(prs: list[tuple[str, str]], idx: list[int]) -> tuple[
    torch.Tensor, torch.Tensor, torch.Tensor
]:
    srcs = [torch.tensor([src_stoi[c] for c in prs[i][0]]) for i in idx]
    lengths = torch.tensor([len(s) for s in srcs])
    tgts = [torch.tensor([BOS] + [tgt_stoi[c] for c in prs[i][1]] + [EOS])
             for i in idx]
```

```

S = nn.utils.rnn.pad_sequence(srcs, batch_first=True, padding_value=PAD)
T = nn.utils.rnn.pad_sequence(tgts, batch_first=True, padding_value=PAD)
return S, lengths, T

def unambiguous(prs: list[tuple[str, str]]) -> list[tuple[str, str]]:
    return [(s, t) for s, t in prs
            if "/" not in s or int(s[:2]) > 12 or int(s[3:5]) > 12]

# [2]
valid_unamb = unambiguous(valid_pairs)
test_unamb = unambiguous(test_pairs)
valid_fixed = valid_unamb[:400]
# [3]
print(f"fixed validation {len(valid_fixed)}; final test {len(test_unamb)}")

class LearnedAlignment(nn.Module):
    def __init__(self, hidden: int = 128, alignment: int = 128):
        super().__init__()
        self.key_map = nn.Linear(hidden, alignment, bias=False)
        self.query_map = nn.Linear(hidden, alignment, bias=False)
        self.selector = nn.Linear(alignment, 1, bias=False)

    def forward(
        self,
        memory: torch.Tensor,
        query: torch.Tensor,
        valid: torch.Tensor,
    ) -> tuple[torch.Tensor, torch.Tensor]:
        match_features = torch.tanh(
            self.key_map(memory) + self.query_map(query).unsqueeze(1)
        ) # (B, T_src, d_a)
        scores = self.selector(match_features).squeeze(-1) # (B, T_src)
        scores = scores.masked_fill(~valid, -torch.inf)
        weights = torch.softmax(scores, dim=-1)
        context = torch.bmm(weights.unsqueeze(1), memory).squeeze(1)
        return context, weights

# [4]
class AttentiveSeq2Seq(nn.Module):
    def __init__(self, v_src: int, v_tgt: int, embed: int = 32,
                 hidden: int = 128, alignment: int = 128):
        super().__init__()
        self.src_emb = nn.Embedding(v_src, embed, padding_idx=PAD)
        self.tgt_emb = nn.Embedding(v_tgt, embed, padding_idx=PAD)
        self.encoder = nn.LSTM(embed, hidden, batch_first=True)
        self.attention = LearnedAlignment(hidden, alignment)
        self.decoder = nn.LSTM(embed + hidden, hidden, batch_first=True)
        self.out = nn.Linear(2 * hidden, v_tgt)

```



```

def encode(
    self, S: torch.Tensor, lengths: torch.Tensor,
) -> tuple[torch.Tensor, tuple[torch.Tensor, torch.Tensor], torch.Tensor]:
    packed = nn.utils.rnn.pack_padded_sequence(
        self.src_emb(S), lengths, batch_first=True, enforce_sorted=False)
    packed_memory, state = self.encoder(packed)
    memory, _ = nn.utils.rnn.pad_packed_sequence(
        packed_memory, batch_first=True, total_length=S.shape[1])
    valid = torch.arange(S.shape[1])[None, :] < lengths[:, None]
    return memory, state, valid

def decode_step(
    self,
    token: torch.Tensor,
    state: tuple[torch.Tensor, torch.Tensor],
    memory: torch.Tensor,
    valid: torch.Tensor,
) -> tuple[torch.Tensor, tuple[torch.Tensor, torch.Tensor], torch.Tensor]:
    query = state[0][-1] # previous hidden state
    context, weights = self.attention(memory, query, valid)
    decoder_input = torch.cat(
        (self.tgt_emb(token), context.unsqueeze(1)), dim=-1
    )
    decoder_output, state = self.decoder(decoder_input, state)
    logits = self.out(torch.cat((decoder_output[:, 0], context), dim=-1))
    return logits, state, weights

def forward(
    self, S: torch.Tensor, lengths: torch.Tensor, T_in: torch.Tensor,
) -> torch.Tensor:
    memory, state, valid = self.encode(S, lengths)
    logits = []
    for t in range(T_in.shape[1]):
        logit, state, _ = self.decode_step(
            T_in[:, t:t + 1], state, memory, valid)
        logits.append(logit)
    return torch.stack(logits, dim=1) # (B, T_tgt, V_tgt)

@torch.inference_mode()
def greedy_batch(
    model: AttentiveSeq2Seq,
    prs: list[tuple[str, str]],
    batch: int = 128,
    maxlen: int = 12,
) -> list[tuple[str, str, torch.Tensor]]:
    model.eval()
    outputs = []
    for start in range(0, len(prs), batch):
        subset = prs[start:start + batch]

```

```

S, lengths, _ = batchify(subset, list(range(len(subset))))
memory, state, valid = model.encode(S, lengths)
token = torch.full((len(subset), 1), BOS)
token_steps, weight_steps = [], []
for _ in range(maxlen):
    logits, state, weights = model.decode_step(
        token, state, memory, valid)
    token = logits.argmax(-1, keepdim=True)
    token_steps.append(token[:, 0])
    weight_steps.append(weights)
predicted = torch.stack(token_steps, dim=1)
attention = torch.stack(weight_steps, dim=1)
for row, (source, _) in enumerate(subset):
    chars = []
    for index in predicted[row].tolist():
        if index == EOS:
            break
        chars.append(tgt_itos.get(index, "?"))
    outputs.append((
        source,
        "".join(chars),
        attention[row, :, :lengths[row]].clone(),
    ))
return outputs

@torch.inference_mode()
def exact_match(
    model: AttentiveSeq2Seq, prs: list[tuple[str, str]],
) -> float:
    generated = greedy_batch(model, prs)
    return sum(
        guess == truth
        for (_, truth), (_, guess, _) in zip(prs, generated)
    ) / len(prs)

```

fixed validation 400; final test 437

Notice what changed relative to Chapter 11. The embeddings and encoder width are untouched. Values are the contextual encoder states themselves; learned projections create match features for keys and queries. The decoder must now advance one target step at a time because its next context depends on its previous state. Teacher forcing still supplies the true previous token, but it no longer permits one fused decoder call.

13.5.2. The matched-schedule rematch

The full Chapter 11 validation curve below is frozen from a byte-identical re-execution of that chapter's published baseline. We do not retrain it here. We train the attentive model once and inspect only validation alignments. Chapter 11's test endpoint is not globally sealed here because that chapter has

13. Attention: Making the Kernel Learnable

already reported it; for this rematch it remains decision-inert, and we request one final scalar only after the attention choices are fixed:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Matched-schedule date rematch and reused-endpoint audit.
3. Report or visualize the measured result.

CODE

```
# [1]
checkpoints = (2, 4, 6, 8, 12, 16, 20, 25)
baseline_curve = {
    2: 0.0025, 4: 0.1475, 6: 0.5375, 8: 0.7575,
    12: 0.9500, 16: 0.9500, 20: 0.9950, 25: 0.9575,
}

torch.manual_seed(6050)
attention_model = AttentiveSeq2Seq(V_src, V_tgt)
optimizer = torch.optim.Adam(attention_model.parameters(), lr=1e-3)
attention_curve = {}

# [2]
for epoch in range(1, 26):
    attention_model.train()
    permutation = torch.randperm(len(train_pairs))
    for start in range(0, len(train_pairs), 128):
        ids = permutation[start:start + 128].tolist()
        S, lengths, T = batchify(train_pairs, ids)
        logits = attention_model(S, lengths, T[:, :-1])
        loss = F.cross_entropy(
            logits.reshape(-1, V_tgt),
            T[:, 1:].reshape(-1),
            ignore_index=PAD,
        )
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(attention_model.parameters(), 5.0)
        optimizer.step()
    if epoch in checkpoints:
        attention_curve[epoch] = exact_match(
            attention_model, valid_fixed)

# [3]
print("epoch    fixed-state    attention")
for epoch in checkpoints:
    print(f"{epoch:2d}          {baseline_curve[epoch]:6.2%}          "
          f"{attention_curve[epoch]:6.2%}")
```

13. Attention: Making the Kernel Learnable

```

validation_outputs = greedy_batch(attention_model, valid_fixed)
source_example, truth_example = valid_fixed[0]          # fixed before training
_, generated_example, alignment_example = validation_outputs[0]

year_mass, year_top1, row_errors = [], [], []
for (source, _), (_, _, weights) in zip(valid_fixed, validation_outputs):
    year_positions = list(range(len(source) - 4, len(source)))
    for step in range(4):
        year_mass.append(weights[step, year_positions].sum().item())
        year_top1.append(int(weights[step].argmax().item() in year_positions))
    row_errors.append((weights[:,10].sum(1) - 1).abs().max().item())

attention_params = sum(p.numel() for p in attention_model.parameters())
baseline_params = 169_326
print(f"parameters: baseline {baseline_params:,}; "
      f"attention {attention_params:,} "
      f"({(attention_params / baseline_params - 1):.1%})")
print(f"validation example: {source_example!r} -> "
      f"{generated_example!r} (truth {truth_example!r})")
print(f"validation year-region mass: {np.mean(year_mass):.3%}; "
      f"top key in region: {np.mean(year_top1):.3%}")
print("maximum validation row-sum error:", f"{max(row_errors):.2e}")

test_exact = exact_match(attention_model, test_unamb)    # one final test audit
print(f"one final test audit ({len(test_unamb)} sources): "
      f"baseline 93.1%; attention {test_exact:.1%}")

```

epoch	fixed-state	attention
2	0.25%	9.50%
4	14.75%	74.50%
6	53.75%	93.25%
8	75.75%	99.00%
12	95.00%	99.75%
16	95.00%	99.75%
20	99.50%	99.75%
25	95.75%	99.75%

parameters: baseline 169,326; attention 269,550 (+59.2%)
validation example: 'may 17, 1971' -> '1971-05-17' (truth 1971-05-17)
validation year-region mass: 97.469%; top key in region: 99.688%
maximum validation row-sum error: 2.38e-07
one final test audit (437 sources): baseline 93.1%; attention 100.0%

At epoch 6, the fixed-state baseline is at 53.8% and the attentive model is at 93.25%, a 39.5-point lead. At epoch 12 the comparison is 95.0% versus 99.75%. The one final test audit is 93.1% versus 100.0% on 437 unambiguous sources.

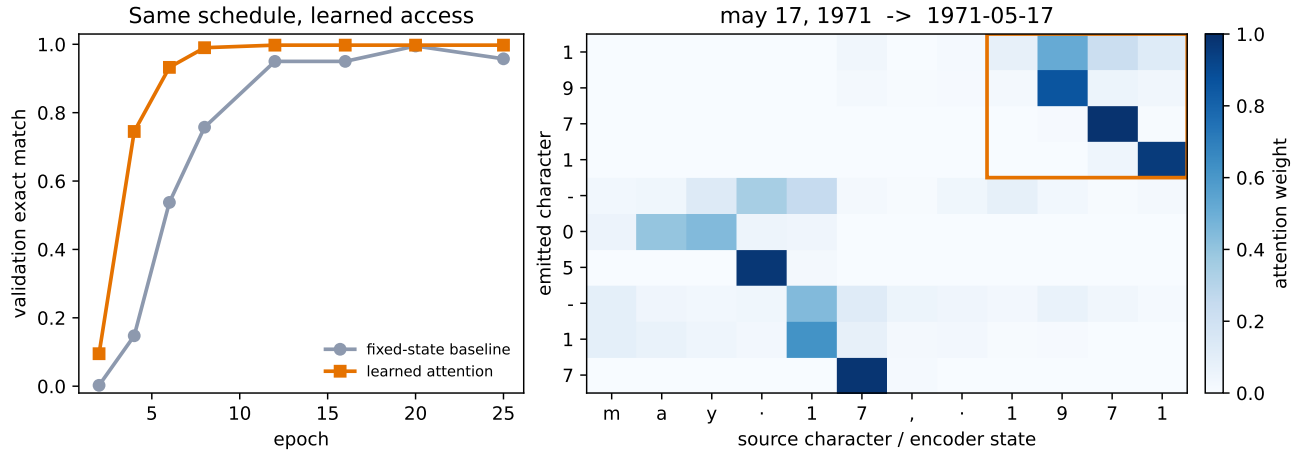


Figure 13.5.: Left: exact-sequence accuracy on Chapter 11’s fixed 400-source validation subset. Both models use the same data, embedding and recurrent hidden widths, optimizer, top-level seed, and epoch schedule; attention adds parameters and a stepwise decoder, so this is not a parameter- or compute-matched ablation. Right: the predeclared first validation example. Rows are emitted ISO characters, columns are contextual encoder states indexed by source character, and the orange box marks the source year. The first four rows concentrate there.

⚠ Faster convergence is not a one-variable proof

We matched examples, embedding and recurrent hidden widths, optimizer, top-level seed, number of updates, and epochs. We did **not** match parameter count, computation, or minibatch order: constructing models with different parameter counts consumes different random draws before their first permutation. The attentive model has 269,550 parameters versus 169,326, an increase of 59.2%, and its input-feeding decoder uses a Python step loop where Chapter 11’s teacher-forced baseline uses one fused target-sequence call. This one seeded synthetic rematch shows that the attention-equipped model reaches higher accuracy in fewer epochs and optimizer updates, not in less wall-clock time. It does not isolate bottleneck removal as the only possible cause.

The heatmap answers a narrower question. On the fixed validation example, the year must travel from the last four source positions to the first four outputs. The orange block catches that movement, and across all 400 fixed validation examples, 97.5% of those four rows’ mass lands on states indexed by the source-year region. The top-weighted state is indexed inside that region 99.7% of the time.

Columns are indexed by source characters, but their values are encoder states. Each state summarizes a recurrent prefix, so a delimiter can legitimately carry information about a field that just ended. Attention weights are internal allocation weights, not calibrated confidence and not a causal explanation of the model. The quantitative audit and the visual alignment complement each other; neither substitutes for the other.

13.6. One learned view, for now

One attention head learns one set of query, key, and value projections. Multi-head attention repeats that operator with several projection sets, concatenates the reads, and learns an output projection. It works for cross-attention as well as self-attention, and different heads *can* learn different relationships without

any guarantee that they will. We keep one head in the date rematch so its alignment stays easy to inspect. Chapter 14 derives the multi-head shapes and implements the full operator in its self-attention setting.

13.7. The bottleneck that remains

Cross-attention has repaired the fixed handoff. A decoder output can read any encoder state through one learned allocation. Yet both surrounding LSTMs remain recurrent: $\mathbf{h}_i$ waits for $\mathbf{h}_{i-1}$, and $\mathbf{s}_t$ waits for $\mathbf{s}_{t-1}$. Within either sequence, training cannot create all recurrent states at once.

The Q/K/V operator itself does not care where its three inputs came from. What if every position in one sequence supplied queries, keys, and values to every other position? Using that operator in place of the RNN can remove recurrence within each layer, but it also removes the RNN's built-in sense of order. Chapter 8 warned that throwing away *where* incurs a debt; Chapter 14 pays it back with positional information. Chapter 11's masking warning also returns there: a causal mask will decide which future positions an autoregressive model may not see.

Check yourself

Close the book for one minute and reconstruct the soft lookup.

- What distinct jobs do queries, keys, and values perform?
- Why does masking belong inside score normalization?
- What did making the compatibility function learnable buy over the fixed kernel?

13.8. Okay, so — attention softens the address

1. **Attention softens the address.** Chapter 11 learned embedding content while keeping lookup indices hard. Chapter 2's differentiable-lookup promise is now paid: learned scores become a distribution over memory locations.
2. **Additive attention learns the compatibility function.** Query and key enter a shared alignment space, tanh creates match features, and one selector produces each score. The same scorer is reused across all decoder/source pairs.
3. **Scaled dot product learns comparison spaces.** Projected $\mathbf{QK}^\top / \sqrt{d_k}$ maps directly to dense matrix products. Under the idealized independent unit-variance model, scaling holds score variance near one instead of letting it grow with d_k .
4. **Masking is part of the operator.** Padding is excluded before softmax; zero is not a mask. Cross-attention sees all real source positions, while causal masking belongs to the self-attentive decoder in Chapter 14.
5. **The date rematch exposes learned access.** Under the shared schedule, attention reaches 93.25% validation exact match at epoch 6 and 100.0% on the final test audit. Its year rows route 97.5% of validation weight to states indexed by the source-year region, with the parameter/compute caveat stated in full.

Sources and further reading

- Bahdanau, Cho, & Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*: conjectures that the fixed-length encoder vector is a bottleneck, then tests differentiable soft search with an additive scorer.
- Luong, Pham, & Manning, *Effective Approaches to Attention-based Neural Machine Translation*: compares dot, general, and concat scoring functions and clarifies a second recurrent-attention timing convention.
- Vaswani et al., *Attention Is All You Need*: introduces scaled dot-product and multi-head attention in the architecture Chapter 14 builds.

Exercises

1. **(Pencil.)** Starting from Equation 13.2, write every batched shape for $B = 32$, source length 17, $d_e = 128$, $d_d = 96$, and $d_a = 64$. Which axis must softmax normalize, and why can the raw encoder and decoder widths differ?
2. **(Code.)** Extend the scaled-dot function with a deliberately all-false mask row. Confirm the guard catches it. Remove the guard and inspect the result of softmax over all $-\infty$. Then replace the mask with zero scores and explain why padding receives weight.
3. **(Pencil.)** (a) Generalize the scaling derivation to independent coordinates with variances σ_q^2 and σ_k^2 . **(Code.)** (b) Modify the seeded audit and check which divisor returns score variance near one.
4. **(Code.)** Replace the date model’s additive scorer with learned projected scaled-dot attention. Keep the split, schedule, and validation subset fixed. Select any design choices on validation, report the reused test endpoint once, and compare both convergence and year-region alignment. Explain why a large weight is neither calibrated confidence nor a causal explanation.
5. **(Pencil.)** The heatmap labels columns by source characters even though each value is a recurrent prefix state. Give two reasons a delimiter state might carry useful month or day information. Then state one conclusion the heatmap supports and one conclusion it cannot support.
6. **(Audit.)** The date model’s `nn.Embedding` learned a vector for each of the characters 0–9. Predict their geometry before looking: a clean number line ordered by numeric value is the *named wrong answer* — tempting, and not what the training pressure rewards. Extract the ten rows, project them to two dimensions, and compare against the same rows from an untrained model (the falsifying control). What structure did training actually build, and which of the model’s jobs — months, days, years — explains it?

14. Self-Attention and the Transformer

Chapter 13 ended with a question that the attention operator itself had already answered. A query, key, and value need not come from an encoder or a decoder. They need not come from a recurrent network at all. If all three come from the same sequence, each token can ask every token—including itself—what matters and route the answers directly. The recurrent state—the last serial bottleneck left standing—becomes optional.

That substitution is the core of a *Transformer*. It is also easy to state too quickly. Removing recurrence creates a position problem. Splitting one attention operation into heads creates a bookkeeping problem. Stacking the resulting operations creates an optimization problem. This chapter solves each one—from the equations outward—then returns to the book-corpus benchmark from Chapter 10 with a deliberately small causal Transformer.

Here is the result we have to explain on the same 14,860-character held-out tail. In one exactly matched seed, positional encoding improves the Transformer’s loss from 2.3405 to 1.9190. Yet Chapter 10’s compact LSTM remains better at 1.8881. The new architecture has learned, and position matters in this run—but at this scale, the LSTM still wins.

14.1. Let the sequence query itself

Suppose the sentence contains the words *bank*, *river*, and *boats*. A useful representation of *bank* should route information from *river* and *boats*, while another occurrence—now near *loan* and *interest*—should route different information. Chapter 13 built exactly this kind of content-dependent routing. Self-attention changes only the source of the three inputs.

The three questions still fit: a query asks “what am I looking for?”, a key advertises “what do I offer?”, and a value carries “what information do I contain?”

Let a batch of token representations be

$$X \in \mathbb{R}^{B \times n \times d},$$

where B is batch size, n is sequence length, and d is model width. Learned projections produce

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

with $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$. Scaled dot-product attention is

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^\top}{\sqrt{d}} \right) V.$$

For one sequence, $QK^\top$ has shape $n \times n$. Row i contains the scores from query position i to every key position j . The row-wise softmax makes each row a distribution, and multiplying by V returns one weighted sum for each query:

$$(n, d)(d, n) \rightarrow (n, n), \quad (n, n)(n, d) \rightarrow (n, d).$$

Chapter 10 named time as the book’s third sharing axis. An RNN shares one transition rule across time. Self-attention keeps that stationarity and extends the shared rule across every ordered pair of positions. The weights in a particular attention row still depend on the content; what is shared is the machinery that produces them.

All query positions can be evaluated together during training because none waits for a hidden state from the preceding position. That is a structural statement—not a timing result. Dense self-attention still forms n^2 pairwise scores, and autoregressive generation still emits one new token at a time.

14.1.1. The position debt

Bare self-attention knows content but not slot number. A precise proof is more useful than the slogan.

Let P be an $n \times n$ permutation matrix, so PX reorders the tokens. Then

$$Q' = PQ, \quad K' = PK, \quad V' = PV$$

and therefore

$$Q'K'^\top = P(QK^\top)P^\top.$$

Write $S = QK^\top/\sqrt{d}$; then $S' = PSP^\top$. Row-wise softmax respects the same simultaneous row-and-column permutation:

$$\text{softmax}(PSP^\top) = P \text{softmax}(S)P^\top.$$

Putting the pieces together gives

$$\text{SA}(PX) = P \text{SA}(X).$$

Self-attention is *permutation equivariant*: permute the inputs and the outputs permute in the same way. It is not permutation invariant—the token outputs do not all become identical. But without another signal, the operator has no basis for distinguishing “first” from “fifth.” A pooled sequence summary can consequently become invariant to order.

Chapter 8 planted this debt when convolution received locality “for free.” A convolution knows which values are neighbors because its kernel is tied to nearby offsets. Global attention abandons that built-in geometry—so we must pay to put position back in.

PLAN

1. Define the reusable helpers: `numpy_attention` and `numpy_positions`.
2. Prepare the inputs and fixed settings for the example.
3. Report the permutation-equivariance and fixed-slot differences.

CODE

```
import math
import numpy as np
import matplotlib.pyplot as plt

# [1]
def numpy_attention(x: np.ndarray) -> np.ndarray:
    scores = x @ x.T / math.sqrt(x.shape[-1])
    scores = scores - scores.max(axis=-1, keepdims=True)
    weights = np.exp(scores)
    weights = weights / weights.sum(axis=-1, keepdims=True)
    return weights @ x

def numpy_positions(length: int, width: int) -> np.ndarray:
    position = np.arange(length)[:, None]
    frequency = np.exp(
        np.arange(0, width, 2) * (-math.log(10_000.0) / width)
    )
    pe = np.zeros((length, width))
    pe[:, 0::2] = np.sin(position * frequency)
    pe[:, 1::2] = np.cos(position * frequency)
    return pe

# [2]
rng = np.random.default_rng(6050)
x = rng.normal(size=(5, 8))
permutation = np.array([2, 4, 0, 1, 3])
bare_error = np.abs(
    numpy_attention(x[permutation]) - numpy_attention(x)[permutation]
).max()

pe = numpy_positions(5, 8)
with_position = numpy_attention(x + pe)
fixed_slot_change = np.abs(
    numpy_attention(x[permutation] + pe) - with_position[permutation]
).max()

# [3]
print(f"bare permutation-equivariance error: {bare_error:.2e}")
print(f"fixed-slot difference after adding position: {fixed_slot_change:.3f}")
```

bare permutation-equivariance error: 2.22e-16
fixed-slot difference after adding position: 1.581

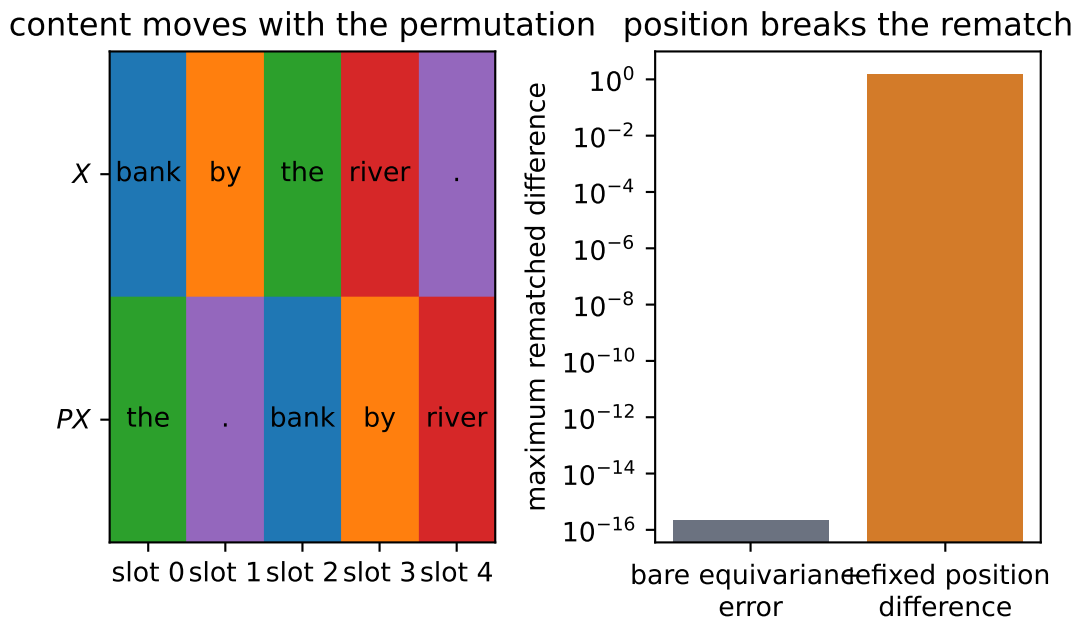


Figure 14.1.: The same content is permuted across slots (left). The audit (right) shows bare self-attention rematches after the same output permutation, while fixed positional signals break that rematch.

The first error is numerical roundoff, about 2.2×10^{-16} . Adding a fixed signal to each slot changes the rematched outputs by about 1.581 in this example. Position has not been inferred from content; it has been supplied.

14.2. Position as a bank of clocks

The simplest additive code repeats $i/(n-1)$ in every coordinate. It supplies order, but every position vector lies on the same ray and differs only in magnitude; attention receives no multidirectional geometry. A binary code supplies more coordinates, yet adjacent integers can flip many bits and have poor locality. These are not impossible designs—they are weak inductive biases for a model that must reason about shifts and distances.

Sinusoidal encoding gives every position a bank of clocks. For even model width d , define

$$\begin{aligned} \text{PE}(i, 2j) &= \sin(i \cdot 10000^{-2j/d}), \\ \text{PE}(i, 2j+1) &= \cos(i \cdot 10000^{-2j/d}). \end{aligned}$$

Small j gives a fast clock; large j gives a slow one. Every sine coordinate is paired with a cosine at the same frequency. The model receives

$$Z_i = \sqrt{d} E_i + \text{PE}(i),$$

where E_i is the token embedding. Scaling the embedding keeps its initial magnitude commensurate with the position vector.

Why pair sine and cosine? For frequency ω , advancing by offset δ is a rotation—an algebraic answer to a geometric question:

$$\begin{bmatrix} \sin((i + \delta)\omega) \\ \cos((i + \delta)\omega) \end{bmatrix} = \begin{bmatrix} \cos(\delta\omega) & \sin(\delta\omega) \\ -\sin(\delta\omega) & \cos(\delta\omega) \end{bmatrix} \begin{bmatrix} \sin(i\omega) \\ \cos(i\omega) \end{bmatrix}.$$

The rotation depends on the offset, not the absolute position. This gives learned linear maps a convenient raw material for representing relative shifts. It is a property of the positional pair itself—not a guarantee that arbitrary learned W_Q and W_K will preserve or use it.

i Show the rotation, then name it

Instead of adding the clock coordinates to the token, rotate each query and key directly. If R_i is the block-diagonal rotation for position i , use $\tilde{\mathbf{q}}_i = R_i \mathbf{q}_i$ and $\tilde{\mathbf{k}}_j = R_j \mathbf{k}_j$. Their score becomes

$$\tilde{\mathbf{q}}_i^\top \tilde{\mathbf{k}}_j = \mathbf{q}_i^\top R_i^\top R_j \mathbf{k}_j = \mathbf{q}_i^\top R_{j-i} \mathbf{k}_j.$$

The positional factor now depends on the relative offset $j - i$. This construction is called **rotary positional embedding (RoPE)**. Content still matters through $\mathbf{q}_i$ and $\mathbf{k}_j$; RoPE changes how position enters their comparison.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify the rotation identity and constant-norm invariant.
-

CODE

```
# [1]
pe = numpy_positions(100, 32)

delta = 7
j = 5
omega = 10_000 ** (-2 * j / 32)
rotation = np.array([
    [np.cos(delta * omega), np.sin(delta * omega)],
    [-np.sin(delta * omega), np.cos(delta * omega)],
])
pair_i = pe[13, 2 * j : 2 * j + 2]
pair_shifted = pe[13 + delta, 2 * j : 2 * j + 2]
rotation_error = np.abs(rotation @ pair_i - pair_shifted).max()
norm_error = np.abs(np.linalg.norm(pe, axis=1) - 4.0).max()

# [2]
```

```
print(f"rotation identity maximum error: {rotation_error:.2e}")
print(f"constant position-vector norm maximum error: {norm_error:.2e}")
```

rotation identity maximum error: 1.11e-16

constant position-vector norm maximum error: 0.00e+00

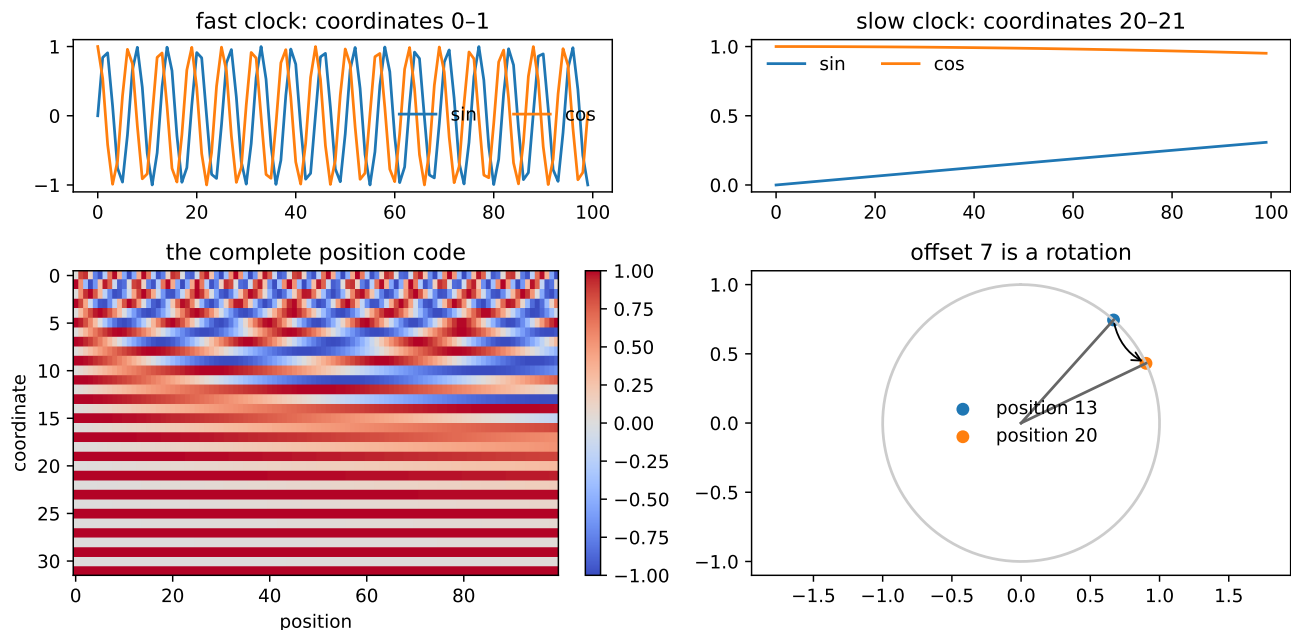


Figure 14.2.: Sinusoidal position is a bank of clocks. Fast coordinates track local offsets; slow coordinates vary across longer ranges. Each sine/cosine pair acts like a clock, and a fixed offset rotates that pair by a start-independent angle.

For $d = 32$, each position vector has norm $\sqrt{d/2} = 4$ because every $\sin^2 + \cos^2$ pair contributes one. Adding this vector does *not* preserve the norm of the combined token representation—the embedding and position vector can reinforce or cancel one another. Layer normalization will soon manage scale at the block level.

14.3. One routing operation, many heads

A single attention distribution must make one compromise about what to retrieve. Multi-head attention lets several smaller routing systems—separate views of the same residual stream—operate in parallel. Choose a number of heads h that divides d , and let $d_h = d/h$. For each head r ,

$$Q_r = XW_Q^{(r)}, \quad K_r = XW_K^{(r)}, \quad V_r = XW_V^{(r)},$$

where $W_Q^{(r)}, W_K^{(r)}, W_V^{(r)} \in \mathbb{R}^{d \times d_h}$. The head output is

$$H_r = \text{softmax} \left(\frac{Q_r K_r^\top}{\sqrt{d_h}} + M \right) V_r.$$

Here M is the visibility mask defined next; $M = 0$ for unmasked attention.

The complete operation concatenates the heads and mixes them:

$$\text{MHA}(X) = \text{Concat}(H_1, \dots, H_h)W_O, \quad W_O \in \mathbb{R}^{d \times d}.$$

The shape ledger is the safest implementation guide:

$$(B, n, d) \rightarrow (B, h, n, d_h) \rightarrow (B, h, n, n) \rightarrow (B, h, n, d_h) \rightarrow (B, n, d).$$

Scaling uses $\sqrt{d_h}$, not $\sqrt{d}$, because a head's dot product sums d_h terms. With fixed d , splitting into heads does not multiply the leading projection count: W_Q, W_K, W_V, W_O together contribute about $4d^2$ weights regardless of h . Heads divide the representational subspace—they do not each receive a fresh width- d model.

14.3.1. The mask is part of the model

Chapter 11 called masking the rule that tells a model *which positions it may not look at*. Here that rule is causal: position i may use tokens at positions $j \leq i$ but must not inspect future tokens $j > i$. Rows are queries and columns are keys, so the forbidden region is the strict upper triangle. We add

$$M_{ij} = \begin{cases} 0, & j \leq i, \\ -\infty, & j > i \end{cases}$$

before softmax. Replacing forbidden probabilities with zero afterward would leave the remaining row unnormalized. Masking every key in a row is invalid—softmax would receive only negative infinity. Here the diagonal remains visible: the representation at a character may use that character while predicting the next one.

PLAN

1. Define the `CausalMultiHeadAttention` module.
 2. Prepare the inputs and fixed settings for the example.
 3. Implement causal multi-head attention and audit its mask.
-

CODE

```
import torch
from torch import nn

# [1]
class CausalMultiHeadAttention(nn.Module):
    def __init__(self, width: int, heads: int) -> None:
        super().__init__()
        if width % heads != 0:
```

```

        raise ValueError("width must be divisible by heads")
    self.heads = heads
    self.head_width = width // heads
    self.qkv = nn.Linear(width, 3 * width)
    self.project = nn.Linear(width, width)
    nn.init.xavier_uniform_(self.qkv.weight)
    nn.init.zeros_(self.qkv.bias)

def forward(self, x: torch.Tensor, return_weights: bool = False):
    batch, length, width = x.shape
    qkv = self.qkv(x)
    qkv = qkv.reshape(
        batch, length, 3, self.heads, self.head_width
    ).permute(2, 0, 3, 1, 4)
    q, k, v = qkv.unbind(dim=0)
    scores = q @ k.transpose(-2, -1) / math.sqrt(self.head_width)
    forbidden = torch.triu(
        torch.ones(length, length, dtype=torch.bool, device=x.device),
        diagonal=1,
    )
    scores = scores.masked_fill(forbidden, float("-inf"))
    weights = torch.softmax(scores, dim=-1)
    routed = weights @ v
    routed = routed.transpose(1, 2).reshape(batch, length, width)
    output = self.project(routed)
    if return_weights:
        return output, weights
    return output

# [2]
torch.manual_seed(6050)
demo_attention = CausalMultiHeadAttention(width=32, heads=4)
demo_x = torch.randn(1, 12, 32)
_, demo_weights = demo_attention(demo_x, return_weights=True)

future = torch.triu(torch.ones(12, 12, dtype=torch.bool), diagonal=1)
# [3]
assert demo_weights[0, :, future].max().item() == 0.0
assert torch.allclose(
    demo_weights.sum(dim=-1),
    torch.ones_like(demo_weights.sum(dim=-1)),
    atol=1e-6,
)

```

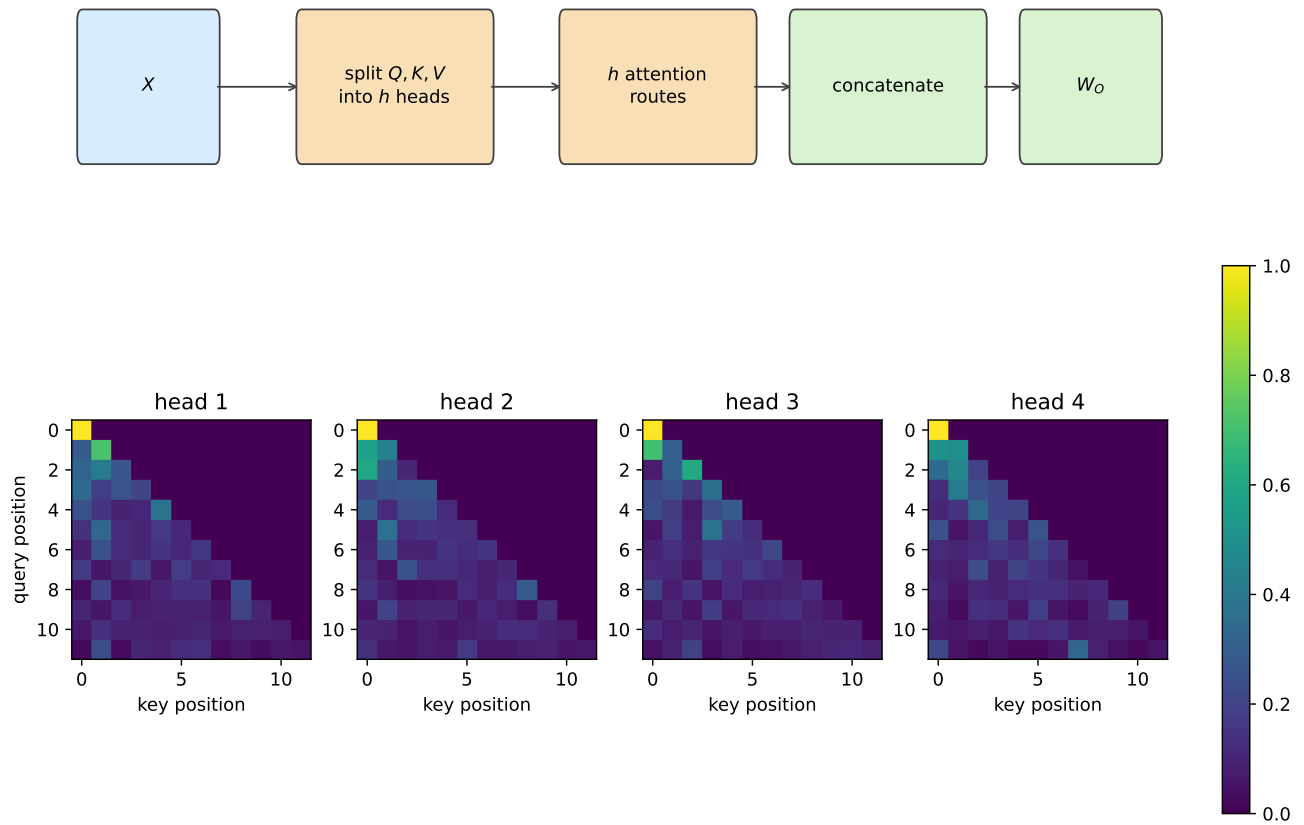



Figure 14.3.: Multi-head attention splits, routes in parallel, concatenates, and projects (top). In the untrained causal heads below, every query row sums to one and every future key has exactly zero weight.

PLAN

1. Report the causal-mask and normalization audit.

CODE

```
# [1]
print("maximum forbidden attention:", demo_weights[0, :, future].max().item())
print(
    "maximum row-sum error:",
    (demo_weights.sum(dim=-1) - 1).abs().max().item(),
)
```

```
maximum forbidden attention: 0.0
maximum row-sum error: 1.1920928955078125e-07
```

The assertions are part of the derivation. A visually plausible triangle can still be transposed, shifted by one, or applied after softmax. Executable shape and normalization checks catch those errors—before training can disguise them.

The mask changes visibility, not dense computation. The implementation still forms all n^2 scores and stores Bhn^2 attention weights, alongside roughly Bnd^2 projection work and $Bn d d_{ff}$ work in the FFN. Self-attention shortens the graph path between distant tokens to one routing step—but “one step away” is not “constant runtime.” Appendix C returns to this n^2 ledger through Roofline analysis and FlashAttention’s I/O-aware schedule.

14.4. Build a Transformer block

Attention routes information between token positions. A Transformer block needs two more ingredients—a stable path through depth and a nonlinear computation at each position.

14.4.1. The residual stream

Chapter 9 introduced the residual stream with one durable sentence: every block reads the stream and writes a correction back. If a sublayer computes $F(x)$, the residual update is

$$x \leftarrow x + F(x).$$

The identity branch provides a direct route for representations and gradients while the learned branch proposes a correction. It makes deep optimization more tractable—it does not guarantee that gradients can never vanish or explode.

Layer normalization controls each token’s feature scale. For one token vector $x \in \mathbb{R}^d$,

$$\mu = \frac{1}{d} \sum_{k=1}^d x_k, \quad \sigma^2 = \frac{1}{d} \sum_{k=1}^d (x_k - \mu)^2,$$

$$\text{LN}(x)_k = \gamma_k \frac{x_k - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta_k, \quad \gamma, \beta \in \mathbb{R}^d.$$

This is Chapter 9’s normalization equation applied along a different axis—the same equation, different axis. Chapter 9’s BatchNorm2d used batch and spatial axes for each channel. LayerNorm flips to the feature axis of each token: it computes statistics across the features within one token, without aggregating statistics across other tokens or examples. Its calculation is the same at training and evaluation time. Before the learned γ and β , the vector is approximately zero mean and unit variance. After applying them, that claim need not remain true.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify LayerNorm centers and scales each token independently.
-

CODE

```
# [1]
audit = torch.tensor([
    [[1.0, 3.0, 5.0, 7.0], [40.0, 50.0, 60.0, 70.0]],
    [[-3.0, 1.0, 5.0, 9.0], [2.0, 2.5, 3.0, 3.5]],
])
normalized = nn.functional.layer_norm(audit, (4,))
token_means = normalized.mean(dim=-1)
token_vars = normalized.var(dim=-1, unbiased=False)
# [2]
assert token_means.abs().max().item() < 1e-6
assert (token_vars - 1).abs().max().item() < 1e-4
```

PLAN

1. Report the per-token LayerNorm audit.
-

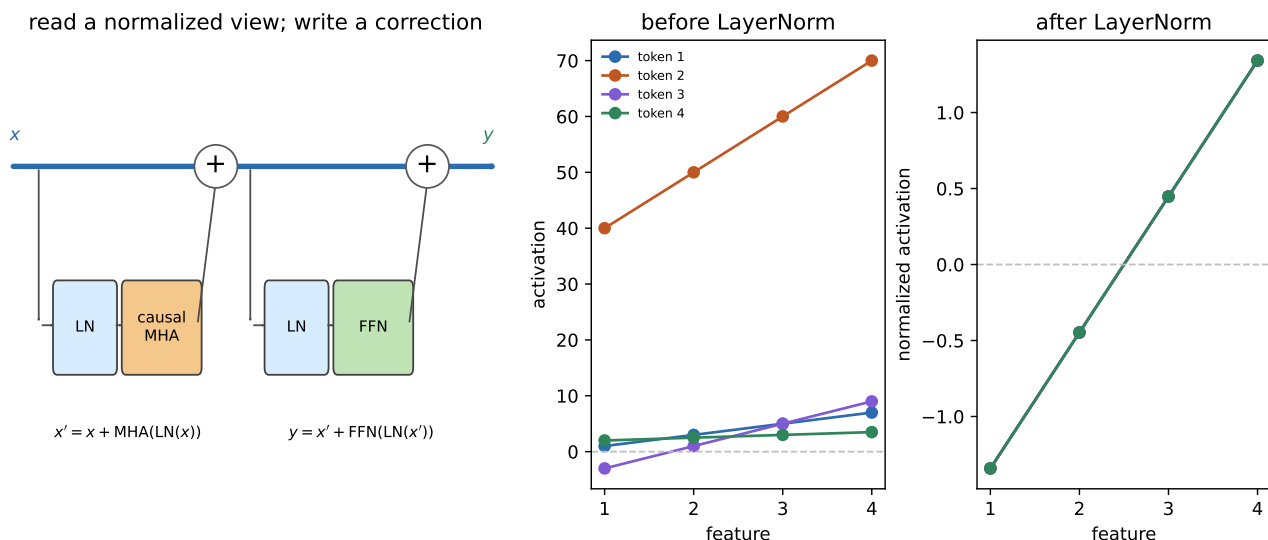


Figure 14.4.: A pre-LayerNorm Transformer block as two read–compute–write stages. The residual stream carries each token directly across both additions while LayerNorm feeds a normalized view to causal multi-head attention and then the position-wise FFN. The audit at right shows what LayerNorm changes: four tokens with different offsets and scales become the same centered, unit-variance profile before the learned affine rescaling.

CODE

```
# [1]
print("maximum absolute token mean:", token_means.abs().max().item())
print("maximum token variance error:", (token_vars - 1).abs().max().item())
```

maximum absolute token mean: 0.0
maximum token variance error: 3.2067298889160156e-05

The original Transformer normalized *after* each residual addition:

$$x \leftarrow \text{LN}(x + \text{MHA}(x)), \quad x \leftarrow \text{LN}(x + \text{FFN}(x)).$$

Our small experiment uses the now-common pre-LayerNorm arrangement:

$$\begin{aligned} x &\leftarrow x + \text{MHA}(\text{LN}(x)), \\ x &\leftarrow x + \text{FFN}(\text{LN}(x)). \end{aligned}$$

Pre-LayerNorm leaves an unobstructed identity route along the residual stream and often improves optimization at initialization. This is an implementation choice, not a universal theorem that post-LayerNorm fails or that pre-LayerNorm eliminates every need for careful optimization.

RMSNorm keeps the scale control but omits mean subtraction:

$$\text{RMSNorm}(\mathbf{x}) = \mathbf{g} \odot \frac{\mathbf{x}}{\sqrt{\frac{1}{d} \sum_{j=1}^d x_j^2 + \epsilon}}.$$

It normalizes each token by its root-mean-square magnitude and retains a learned per-feature scale $\mathbf{g}$. This removes one statistic from the computation; it does not imply universal superiority over LayerNorm. Architecture, optimizer, dtype, and scale still determine the observed trade-off.

14.4.2. The position-wise feedforward network

After tokens communicate through attention, the same two-layer network processes each position independently:

$$\text{FFN}(z) = W_2 \text{ReLU}(W_1 z + b_1) + b_2.$$

Here $W_1 \in \mathbb{R}^{d_{ff} \times d}$, $b_1 \in \mathbb{R}^{d_{ff}}$, $W_2 \in \mathbb{R}^{d \times d_{ff}}$, and $b_2 \in \mathbb{R}^d$.

If $w_{1j}^\top$ is row j of W_1 and v_j is column j of W_2 , the same operation can be written

$$\text{FFN}(z) = \sum_j [w_{1j}^\top z + b_{1j}]_+ v_j + b_2.$$

This form motivates an associative-memory lens. Each first-layer row tests whether the current representation lies in a learned half-space; its positive activation controls how much of the corresponding output vector is written. Empirically, Transformer FFNs can behave like key-value memories. The analogy has limits—the selectors are half-spaces rather than point anchors, the gates do not sum to one, and the operation is not literally kernel regression or a normalized lookup.

The complete block therefore alternates two kinds of computation:

1. causal multi-head attention moves information *between positions*; and
2. the FFN transforms information *within each position*.

Residual additions keep both results in one shared stream—a common workspace that survives the full stack.

14.5. From the original Transformer to this experiment

The 2017 Transformer was an encoder–decoder system for translation. Its encoder stack uses unmasked self-attention to build a representation of the source sentence. Its decoder stack uses causal self-attention over the partial target, then cross-attention whose queries come from the decoder while keys and values come from the encoder. In compact form:

$$\begin{aligned} \text{encoder: } & X_s \rightarrow \text{self-attention} \rightarrow \text{FFN}, \\ \text{decoder: } & X_t \rightarrow \text{causal self-attention} \rightarrow \text{cross-attention to encoder} \rightarrow \text{FFN}. \end{aligned}$$

The book-corpus task predicts the next character from earlier characters. It has no separate source sentence, so the encoder and cross-attention are unnecessary. What remains—a **decoder-only causal Transformer**—has token plus position embeddings, two pre-LayerNorm blocks, a final LayerNorm, and a linear map to vocabulary logits. The logits go directly to cross-entropy. Softmax belongs inside attention and, later, inside sampling—not between the output layer and the loss.

14.6. The book reads itself, again

A fair rematch begins by preserving what can be preserved. Chapter 10 trained on a committed snapshot of Chapters 1–9 with executable code cells and HTML comments removed: 148,594 characters, vocabulary 104, a contiguous 90/10 split, 100-character windows, batch size 64, and 2,501 updates with Adam at learning rate 0.002 and gradient clipping at norm 1. The snapshot keeps later copyedits from moving the benchmark. The held-out metric resets state at each fixed window.

The two architectures cannot be made identical—one has recurrent gates and the other has attention blocks—but their opportunity to see training targets can be. The code below reconstructs Chapter 10’s exact random-window schedule, then gives both Transformer variants the same initial weights and all 16,006,400 target characters in the same order. The held-out tail has been inspected repeatedly across chapters, so treat it as a stable validation benchmark—not a newly sealed test set.

PLAN

1. Implement the tiny Transformer — positions, block, model.
-

CODE

```
# [1]
def sinusoidal_positions(length: int, width: int) -> torch.Tensor:
    position = torch.arange(length, dtype=torch.float32).unsqueeze(1)
    frequency = torch.exp(
        torch.arange(0, width, 2, dtype=torch.float32)
        * (-math.log(10_000.0) / width)
    )
    table = torch.zeros(length, width)
    table[:, 0::2] = torch.sin(position * frequency)
    table[:, 1::2] = torch.cos(position * frequency)
    return table

class TransformerBlock(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.norm1 = nn.LayerNorm(WIDTH)
        self.attention = CausalMultiHeadAttention(WIDTH, HEADS)
        self.norm2 = nn.LayerNorm(WIDTH)
        self.ff = nn.Sequential(
            nn.Linear(WIDTH, FF_WIDTH),
            nn.ReLU(),
            nn.Linear(FF_WIDTH, WIDTH),
        )

    def forward(self, x: torch.Tensor, return_weights: bool = False):
        if return_weights:
            attended, weights = self.attention(self.norm1(x), True)
```

```

        x = x + attended
        return x + self.ff(self.norm2(x)), weights
x = x + self.attention(self.norm1(x))
return x + self.ff(self.norm2(x))

class TinyTransformerLM(nn.Module):
    def __init__(self, vocab: int, positions: bool) -> None:
        super().__init__()
        self.positions = positions
        self.token_embedding = nn.Embedding(vocab, WIDTH)
        nn.init.normal_(
            self.token_embedding.weight,
            mean=0.0,
            std=1 / math.sqrt(WIDTH),
        )
        self.blocks = nn.ModuleList(
            [TransformerBlock() for _ in range(BLOCKS)]
        )
        self.final_norm = nn.LayerNorm(WIDTH)
        self.output = nn.Linear(WIDTH, vocab)
        self.register_buffer(
            "position_table",
            sinusoidal_positions(CONTEXT, WIDTH),
            persistent=False,
        )

    def forward(self, tokens: torch.Tensor, return_weights: bool = False):
        length = tokens.shape[1]
        x = self.token_embedding(tokens) * math.sqrt(WIDTH)
        if self.positions:
            x = x + self.position_table[:length]
        maps = []
        for block in self.blocks:
            if return_weights:
                x, weights = block(x, True)
                maps.append(weights)
            else:
                x = block(x)
        logits = self.output(self.final_norm(x))
        return (logits, maps) if return_weights else logits

```

The paired protocol is worth printing in full — it *is* the experimental design:

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Implement the paired protocol — shared schedule, identical initial tensors.

CODE

```
# Reconstruct the random-number state immediately after Chapter 10 created
# its LSTM, then draw that chapter's complete minibatch-start schedule.
# [1]
torch.manual_seed(SEED)
_ = HistoricalCharLSTM(len(chars))
schedule_generator = torch.Generator()
schedule_generator.set_state(torch.random.get_rng_state())
starts = torch.randint(
    0,
    len(train_data) - CONTEXT - 1,
    (UPDATES, BATCH),
    generator=schedule_generator,
)

# The two Transformer variants begin with exactly the same tensors.
torch.manual_seed(SEED)
base_model = TinyTransformerLM(len(chars), positions=True)
initial_state = {
    name: value.clone() for name, value in base_model.state_dict().items()
}
initial_hash = state_digest(initial_state)
schedule_hash = hashlib.sha256(starts.numpy().tobytes()).hexdigest()

# [2]
print(f"corpus: 9 chapters, {len(text):,} characters, vocab {len(chars)}")
print(
    f"deterministic split: {len(train_data):,} train / "
    f"{len(valid_data):,} held out"
)
print(
    f"Transformer parameters: "
    f"{sum(parameter.numel() for parameter in base_model.parameters()):,}"
)
print(f"first eight window starts: {starts[0, :8].tolist()}")
print(f"initial tensors: {initial_hash[:12]}...")
print(f>window schedule: {schedule_hash[:12]}...")
```

```
corpus: 9 chapters, 148,594 characters, vocab 104
deterministic split: 133,734 train / 14,860 held out
Transformer parameters: 132,488
```



```
first eight window starts: [24368, 94128, 60074, 121547, 118001, 4396, 17011, 8495]
initial tensors: 4d2f7f434cb5...
window schedule: e470a091bd50...
```

The model has 132,488 trainable parameters—736 fewer than Chapter 10’s 133,224. That difference is only 0.55%—close enough to remove model size as an easy explanation. The model width is 84, split into four 21-dimensional heads; each of the two FFNs expands to 168 features. Token embeddings are initialized with coordinate standard deviation $1/\sqrt{d}$ before the displayed $\sqrt{d}$ scaling, so their coordinates and the sinusoidal coordinates begin on comparable scales. `ModuleList` registers the two repeated blocks with the model. The nonpersistent position buffer moves with the model but is neither optimized nor included in the state-dictionary fingerprint; its contents follow deterministically from the displayed formula. There is no dropout, so resetting the trainable/state-dictionary tensors and window schedule makes the positional ablation exact and repeatable.

💡 One seed call was not enough

Calling the same random seed before both training runs would not, by itself, prove that they saw the same windows. Model construction consumes random numbers, and different construction paths can silently shift every later draw. The experiment precomputes the full start-index tensor with an explicit generator and copies one saved initialization into both models. It also prints fingerprints for both artifacts.

Training and evaluation are Chapter 10’s canonical listings, imported. The signature of the imported trainer, for reference:

PLAN

1. Reuse the shared trainer through its explicit model, data, schedule, and optimization interface.
-

CODE

```
# [1]
def fit_next_token(
    model: nn.Module,
    data: torch.Tensor,
    *,
    vocab: int,
    context: int = 100,
    batch: int = 64,
    steps: int = 2501,
    lr: float = 2e-3,
    clip: float = 1.0,
    schedule: list[torch.Tensor] | None = None,
    log_every: int = 500,
    log_decimals: int = 2,
) -> tuple[nn.Module, list[tuple[int, float]]]:
```

14. Self-Attention and the Transformer

... body exactly as Listing 10.1 — this chapter prints only what it changes.

PLAN

1. Chapter 10's trainer, imported — only the deltas printed.

CODE

```

from dlbook.training import fit_next_token      # Listing 10.1, Ch. 10
from dlbook.evaluation import fixed_window_loss # Listing 10.2, Ch. 10

# [1]
def train_transformer(
    positions: bool,
) -> tuple[nn.Module, list[tuple[int, float]]]:
    net = TinyTransformerLM(len(chars), positions=positions)
    net.load_state_dict(initial_state, strict=True) # paired start: same tensors
    assert state_digest(net.state_dict()) == initial_hash
    return fit_next_token(
        net, train_data, vocab=len(chars),
        schedule=starts,                # minibatch order is protocol, not chance
        log_every=250, log_decimals=4,
    )

@torch.no_grad()
def transformer_sample(
    net: nn.Module,
    prompt: str,
    n: int = 300,
    temperature: float = 0.8,
) -> str:
    net.eval()
    generator = torch.Generator().manual_seed(1_406_050)
    tokens = [stoi.get(char, 0) for char in prompt]
    for _ in range(n):
        context = torch.tensor(tokens[-CONTEXT:]).unsqueeze(0)
        probabilities = F.softmax(
            net(context)[0, -1] / temperature, dim=-1
        )
        next_token = torch.multinomial(
            probabilities, 1, generator=generator
        ).item()
        tokens.append(next_token)
    return "".join(chars[index] for index in tokens)

```

The training function is defined once, then called in separate cells so each substantial run stays within the chapter's execution budget.

PLAN

1. Train the positional Transformer.
 2. Report or visualize the measured result.
-

CODE

```
# [1]
position_model, position_curve = train_transformer(positions=True)
position_train_loss = fixed_window_loss(position_model, train_data, vocab=len(chars))
position_valid_loss = fixed_window_loss(position_model, valid_data, vocab=len(chars))
# [2]
print(f"fixed-window train loss: {position_train_loss:.4f}")
print(f"fixed-window held-out loss: {position_valid_loss:.4f}")
```

```
step    0    loss 4.7550
step  250    loss 2.1875
step  500    loss 1.7175
step  750    loss 1.5115
step 1000    loss 1.3890
step 1250    loss 1.3568
step 1500    loss 1.2856
step 1750    loss 1.2261
step 2000    loss 1.1663
step 2250    loss 1.1557
step 2500    loss 1.1101
fixed-window train loss: 1.1157
fixed-window held-out loss: 1.9306
```

PLAN

1. Repeat with position removed and everything else fixed.
 2. Report or visualize the measured result.
-

CODE

```
# [1]
no_position_model, no_position_curve = train_transformer(positions=False)
no_position_train_loss = fixed_window_loss(
    no_position_model, train_data, vocab=len(chars)
)
no_position_valid_loss = fixed_window_loss(
    no_position_model, valid_data, vocab=len(chars)
)
```

```
# [2]
print(f"fixed-window train loss: {no_position_train_loss:.4f}")
print(f"fixed-window held-out loss: {no_position_valid_loss:.4f}")
```

```
step    0    loss 4.7289
step  250    loss 2.4946
step  500    loss 2.3608
step  750    loss 2.2417
step 1000    loss 2.2277
step 1250    loss 2.1510
step 1500    loss 2.0745
step 1750    loss 1.9970
step 2000    loss 1.9607
step 2250    loss 1.9131
step 2500    loss 1.8467
fixed-window train loss: 1.8622
fixed-window held-out loss: 2.3494
```

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Report the positional improvement and LSTM gap.
-

CODE

```
# [1]
lstm_valid_loss = 1.888110429

# [2]
improvement = no_position_valid_loss - position_valid_loss
relative = improvement / no_position_valid_loss
lstm_gap = position_valid_loss - lstm_valid_loss
print(
    f"position improvement: {improvement:.4f} loss "
    f"({relative:.1%} relative)"
)
print(f"positional Transformer minus LSTM: {lstm_gap:.4f} loss")
```

```
position improvement: 0.4188 loss (17.8% relative)
positional Transformer minus LSTM: 0.0425 loss
```

The positional model finishes at 1.1132 on a deterministic pass over the training text and 1.9190 on the held-out tail. Removing position raises those values to 1.8672 and 2.3405. Position lowers held-out loss by 0.4214, or 18.0% relative to the no-position variant.

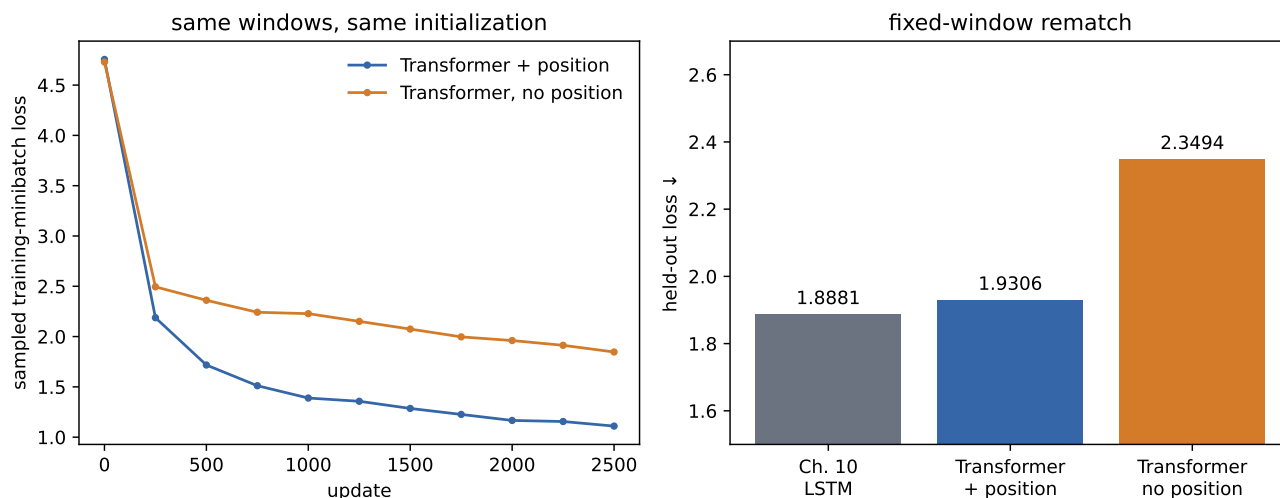


Figure 14.5.: Matched training curves (left) and fixed-window held-out loss (right). Position helps in this seed, while the Chapter 10 LSTM remains the strongest small model.

⚠ What this matched run can—and cannot—show

The causal mask already gives a position some weak structural information: row i sees a prefix of length $i + 1$, and stacked layers can propagate clues from those nested prefixes. That is why the no-position model does better than a purely orderless caricature might suggest—fixed sinusoidal encoding still produces a large improvement in this run.

But Chapter 10’s LSTM remains ahead by 0.0309 held-out loss, or 1.64%. At 100 characters of context and roughly 133,000 parameters, the LSTM wins this regime. This comparison matches corpus, parameter scale, updates, minibatch order, optimizer, clipping, and evaluation. It does not match internal architecture—or promise that either model has been tuned to its own optimum. We did not repeat the comparison across initialization seeds, so the 0.4214 gap is a controlled case study, not an estimate of an average effect.

14.6.1. What did the heads route?

An attention map records routing weights, not reasons, confidence, or a guaranteed explanation of a prediction. Still, inspecting real learned maps can verify the mask and reveal the kinds of patterns a trained model happened to use. The next cell sends the prompt *The gradient* through the positional model and plots the second block’s four heads.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Audit causal masking and row normalization.

CODE

```
# [1]
prompt = "The gradient "
prompt_tokens = torch.tensor([[stoi[char] for char in prompt]])
position_model.eval()
with torch.no_grad():
    _, learned_maps = position_model(prompt_tokens, return_weights=True)
learned = learned_maps[-1][0]

future = torch.triu(
    torch.ones(len(prompt), len(prompt), dtype=torch.bool), diagonal=1
)
assert learned[:, future].max().item() == 0.0
assert torch.allclose(
    learned.sum(dim=-1),
    torch.ones_like(learned.sum(dim=-1)),
    atol=1e-6,
)

# [2]
print("maximum future attention:", learned[:, future].max().item())
print(
    "maximum row-sum error:",
    (learned.sum(dim=-1) - 1).abs().max().item(),
)
```

```
maximum future attention: 0.0
maximum row-sum error: 1.1920928955078125e-07
```

The maps obey the contract exactly: no future key receives weight and every row sums to one. The heads also differ from one another, but naming one “the previous character head” or another “the word head” from this single prompt would outrun the evidence. Attention shows where values were mixed—the rest of the network can transform, cancel, or ignore what was retrieved.

14.6.2. Listen, but do not grade by fluency

Both Transformer variants can now generate by repeatedly truncating to the most recent 100 characters, predicting a distribution for the next one, sampling, and feeding the result back. Unlike Chapter 10’s LSTM, this simple implementation does not carry a recurrent state; it recomputes the current context on every step.

💡 Practice bridge (non-examinable): decoding is another model choice

For next-token logits z_i , **temperature** applies Chapter 2’s softmax dial (Chapter 2):

$$p_i(T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}, \quad T > 0.$$

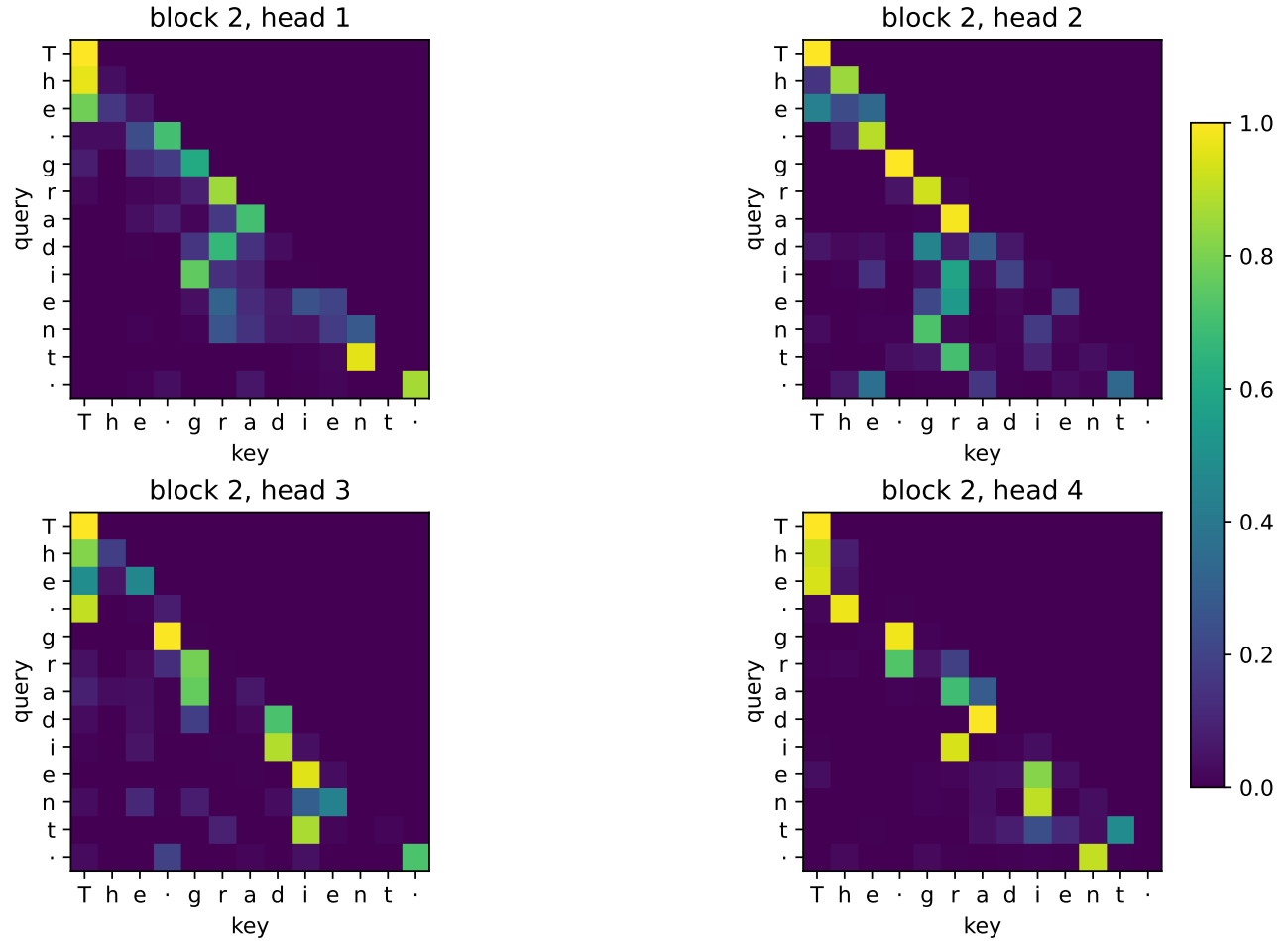


Figure 14.6.: Learned attention weights in the second block for the prompt “The gradient”. The maps are real routing distributions, not explanations or named head roles.

Lower T concentrates probability; higher T flattens it. **Top- k sampling** keeps the k largest logits, sets the rest to negative infinity, then renormalizes. **Nucleus (top- p) sampling** first sorts the model probabilities and keeps the smallest prefix whose cumulative mass reaches p , then renormalizes. The former fixes a candidate count; the latter lets that count adapt to the distribution's shape. Neither changes the trained weights, and neither guarantees a better sample. For evaluation, if L is mean token negative log-likelihood in natural-log units, **perplexity** is $\exp(L)$. It is an exponentiated loss, not a fluency score. Compare it only under the same tokenization, data, and averaging contract.

PLAN

1. Deterministic samples from both Transformer variants.
-

CODE

```
print("WITH POSITION\n")
# [1]
print(transformer_sample(position_model, "The gradient "))
print("\n\nWITHOUT POSITION\n")
print(transformer_sample(no_position_model, "The gradient "))
```

WITH POSITION

The gradient choweven it on and validation for equivariant already of calc
 nuisan one-layer reade-deflane we will have comparable); only the bias boundariest
 ↪ scales and loss long - it training to descens and need gradient chapter. The
 ↪ complex
 pattiallizes the two a rules of this into a biwas a ****multipliest**** w

WITHOUT POSITION

The gradient imodes fing on indevatin to ritwe? Cofiter e. Thrivofforical
 ul ffiftwerdgit) tht dea the the che ift low t Xaysherd unaperie bund n forme.
 The matr s chagule mownte it the of os to derndescate ive. Whis thing are is the comof
 ↪ *****sthe*** is it ist fither \$rongheal 5\$ is ay
 a ***Aneverissist** bow

The positional sample has the book's punctuation, fragments of mathematical vocabulary, and stretches that resemble prose. It still loses its argument. The no-position sample is visibly rougher. Neither qualitative judgment replaces the held-out loss—a sample is one stochastic path, while the loss evaluates every held-out target under the model's full probability distribution.

14.7. What the architecture bought

The rematch is not a referendum on all Transformers—it isolates what this small one bought and what it paid.

Direct content routing. Any earlier visible token can contribute to the next representation in one attention sublayer. The recurrent state no longer has to compress the entire past before the current token can consult it.

Short paths, dense work. The graph distance between visible tokens is short, but the routing matrix is quadratic in context length. Global access is a different tradeoff, not free memory.

Position becomes a modeling choice. Convolution and recurrence bake in local order. Attention asks the designer to supply position—and decide visibility. [Chapter 15](#) will change visibility itself; [Chapter 16](#) will revisit what happens when global routing trades away an image model’s locality bias.

A modular residual stream. Attention moves information, the FFN transforms it, and residual additions let both write into a shared representation. LayerNorm keeps each token’s feature scale manageable along the way.

The causal mask is therefore more than an implementation detail. It says which information a representation is allowed to use. “Visibility is a modeling decision” is the seed this chapter hands forward.

During autoregressive generation, exact softmax attention also makes the past a systems object. A **key/value (KV) cache** retains each layer’s earlier key and value rows so a new query can reuse them instead of recomputing the whole prefix. In the regression language of [Chapter 12](#), the KV cache is the nonparametric estimator’s dataset: one new query still reads a table that grows with t , but the earlier rows do not have to be rebuilt. The toy sampler in [Chapter 14](#) is wasteful precisely because it rebuilds that dataset at every step; [Appendix C](#) separates caching from FlashAttention’s I/O schedule.

The next interlude holds this regression problem fixed and changes its solver. That is where the growing table, finite sufficient state, and delta update can be compared without interrupting the Transformer construction.

The companion volume deliberately does not repeat [Chapters 12–14](#)’s assembly. Its [Beyond This Volume](#) route instead shows how the diagnostic instruments developed there return for kernel geometry, exact I/O-aware algorithms, parallel recurrence, and feature-learning regimes.

i Check yourself

Close the book for one minute and rebuild the Transformer block.

- Why is bare self-attention permutation equivariant?
- Which three paths write into the residual stream?
- What changes when a causal mask restricts visibility?

14.8. Okay, so — the Transformer routes, then computes

1. **Self-attention reuses [Chapter 13](#)’s operator within one sequence.** Q , K , and V all come from X ; one learned comparison rule is shared over every ordered pair. Training positions can be evaluated together, while autoregressive generation remains sequential.

2. **Bare attention is permutation equivariant, not invariant.** $\text{SA}(PX) = P\text{SA}(X)$. Sinusoidal clocks repay Chapter 8's position debt, and their sine/cosine pairs turn relative shifts into rotations.
3. **Multi-head attention is shape discipline.** Split $(B, n, d) \rightarrow (B, h, n, d_h)$, divide scores by $\sqrt{d_h}$, form (B, h, n, n) routing weights, concatenate, and project. At fixed d , the four leading projections remain about $4d^2$ parameters.
4. **The causal mask defines visibility.** Future scores become negative infinity before softmax. The dense operator still performs quadratic work; the mask does not make the forbidden half computationally disappear.
5. **The block alternates routing and computation.** Multi-head attention mixes positions, the FFN transforms each position, and both write corrections into Chapter 9's residual stream. LayerNorm is the same normalization equation over a token's feature axis.
6. **Position wins this seed's controlled ablation.** It lowers held-out loss from 2.3405 to 1.9190. **The LSTM narrowly wins the small-model rematch**, 1.8881 versus 1.9190. Architecture changes inductive bias; it does not guarantee victory in every data and scale regime. ## Sources and further reading {unnumbered}
 - Vaswani et al., *Attention Is All You Need*: multi-head attention, sinusoidal position, and the original post-LayerNorm encoder-decoder.
 - Su et al., *RoFormer: Enhanced Transformer with Rotary Position Embedding*: rotates queries and keys so their positional interaction is expressed through relative offset.
 - Ba, Kiros, and Hinton, *Layer Normalization*: normalization within one example and identical train/evaluation computation.
 - Zhang and Sennrich, *Root Mean Square Layer Normalization*: RMSNorm's scale-only normalization and empirical study.
 - Xiong et al., *On Layer Normalization in the Transformer Architecture*: a qualified comparison of pre- and post-LayerNorm optimization.
 - Geva et al., *Transformer Feed-Forward Layers Are Key-Value Memories*: empirical evidence behind the associative-memory interpretation of FFNs.
 - Jain and Wallace, *Attention Is Not Explanation*: an important limit on reading attention maps as explanations.
 - Holtzman et al., *The Curious Case of Neural Text Degeneration*: nucleus sampling and the distinction between likely continuations and useful generation behavior.

Exercises

1. **(Pencil.)** Starting from $Q' = PQ$, $K' = PK$, and $V' = PV$, prove each step of $\text{SA}(PX) = P\text{SA}(X)$. Then show that mean-pooling the token outputs produces a permutation-*invariant* sequence representation.
2. **(Pencil.)** For width $d = 512$ and $h = 8$, write every tensor shape in multi-head attention for $B = 32$ and $n = 100$. Count the weights in W_Q, W_K, W_V, W_O . Repeat for $h = 16$ and explain which dimensions change while the leading parameter count does not.
3. **(Code.)** Deliberately transpose the causal mask in Figure 14.3, then design an assertion that identifies the first leaked query-key pair. Next set forbidden probabilities to zero *after* softmax and measure the row-sum error.
4. **(Code.)** Replace the fixed sinusoidal table with a learned $\text{Embedding}(100, d)$ while preserving the exact initialization and window schedule protocol. Compare held-out loss. Then extend the sinusoidal buffer by evaluating its formula beyond position 99, and propose an explicit extension

- rule for the learned table before testing either model on a longer context. Which design has a natural extension, and which has to invent one?
5. **(Code.)** Aggregate learned attention by relative offset across the held-out tail rather than naming heads from one prompt. Report a distribution for every head and block. Then perturb the highest-weight key's value vector and measure the logit change. Explain why routing weight and causal influence need not rank tokens identically.

Interlude: Attention as Test-Time Regression

Chapter 14 kept visible evidence in a growing key–value table. This interlude holds the regression question fixed and changes the forward-pass solver. The comparison reveals what each solver retains, what it costs, and which statistical contract it accepts.

One regression, three solvers

The Transformer resolved Chapter 10’s bottleneck by keeping the visible evidence available. Its quadratic routing bill was the price. There is another resolution: keep the *problem* Chapter 12 posed, but change how the forward pass solves it.

That move has become a live architecture program under names such as test-time regression, fast weights, linear attention, DeltaNet, and state-space sequence models. The useful unit is not the model name. It is one objective and the statistical bargain made by its solver.

One objective, four dials

At time t , let learned projections turn token representation $\mathbf{x}_\tau$ into two views,

$$\mathbf{k}_\tau = \mathbf{W}_K \mathbf{x}_\tau, \quad \mathbf{v}_\tau = \mathbf{W}_V \mathbf{x}_\tau.$$

A memory layer can be described as building an online predictor $M_t : \mathbb{R}^{d_k} \rightarrow \mathbb{R}^{d_v}$:

$$M_t = \arg \min_{M \in \mathcal{M}} \frac{1}{2} \sum_{\tau \leq t} w_{t,\tau} \|M(\mathbf{k}_\tau) - \mathbf{v}_\tau\|^2 + \Omega(M), \quad \mathbf{y}_t = M_t(\mathbf{q}_t), \quad \mathbf{q}_t = \mathbf{W}_Q \mathbf{x}_t.$$

This is Wang, Shi, and Fox’s test-time-regression frame, written for vector values. It exposes four dials the book already knows.

The views. W_Q, W_K, W_V are Chapter 13’s learned comparison and payload spaces. The token is not assigned one permanent meaning; it learns what to ask, advertise, and carry.

The history weights. $w_{t,\tau}$ decide how much an older residual matters to the fitting problem. Chapter 10’s forget gates prepared the idea that retention can depend on time and content. A geometric schedule can privilege recent errors; an input-dependent schedule can decide that one token deserves a longer trace.

The model and regularizer. The class $\mathcal{M}$ decides whether the predictor is a local constant, a linear map, or something richer. Ω brings back ridge penalties from Chapter 1 and weight decay from Chapter 4. Regularization controls the fitted map; it is not, by itself, the same operation as forgetting old observations.

The solver. A closed-form local fit, a running sufficient statistic, and one gradient step do different work and preserve different information. This is the new rung: *what if the solver itself were a design choice, or even learned?*

This objective is an umbrella, not an equivalence theorem. Its methods can restrict different function classes, use different weights, and stop at different approximations. Those differences are the lesson.

Solver 1: keep the data and fit at the query

Chapter 12 was more precise than saying “attention minimizes the shared objective.” For a query $\mathbf{q}$, softmax attention solves the local-constant problem

$$\mathbf{c}^*(\mathbf{q}) = \arg \min_{\mathbf{c}} \frac{1}{2} \sum_{\tau \leq t} \kappa(\mathbf{q}, \mathbf{k}_{\tau}) \|\mathbf{c} - \mathbf{v}_{\tau}\|^2.$$

Write $\kappa_{\tau} = \kappa(\mathbf{q}, \mathbf{k}_{\tau})$. Differentiation gives

$$\nabla_{\mathbf{c}} = \sum_{\tau \leq t} \kappa_{\tau} (\mathbf{c} - \mathbf{v}_{\tau}) = \mathbf{0} \quad \Rightarrow \quad \mathbf{c}^* = \frac{\sum_{\tau} \kappa_{\tau} \mathbf{v}_{\tau}}{\sum_{\tau} \kappa_{\tau}}.$$

Choose $\kappa(\mathbf{q}, \mathbf{k}) = \exp(\mathbf{q}^{\top} \mathbf{k} / \sqrt{d_k})$ and the normalized coefficients are exactly a row of scaled dot-product softmax. The kernel supplies the weights; the constant fit supplies the average. An unrestricted M with unit observation weights and no regularizer would instead interpolate compatible training pairs and be undetermined away from them. We will not smuggle the average into a stronger claim.

Here is the numerical stationarity audit. Subtracting the largest score rescales every κ_{τ} by the same positive constant and leaves the minimizer unchanged.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify softmax attention’s local-constant optimum.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
ms_rng = np.random.default_rng(6050)
ms_d, ms_T = 8, 12
ms_K = ms_rng.standard_normal((ms_T, ms_d))
ms_V = ms_rng.standard_normal((ms_T, 3))
ms_q = ms_rng.standard_normal(ms_d)
ms_scores = ms_K @ ms_q / np.sqrt(ms_d)
ms_kappa = np.exp(ms_scores - ms_scores.max())
ms_c = (ms_kappa[:, None] * ms_V).sum(0) / ms_kappa.sum()
ms_attention = (ms_kappa / ms_kappa.sum()) @ ms_V
```

```

ms_gradient = (ms_kappa[:, None] * (ms_c - ms_V)).sum(0)
# [2]
assert np.allclose(ms_c, ms_attention)
# [3]
print(f"max |stationarity gradient|: {np.abs(ms_gradient).max():.3e}")

```

```
max |stationarity gradient|: 3.053e-16
```

This solver retains the key-value rows because a future query may weight every row differently. That is why dense causal attention carries a growing dataset forward.

Solver 2: collapse a factorized kernel into sufficient state

Suppose the kernel factors through a finite feature map $\phi : \mathbb{R}^{d_k} \rightarrow \mathbb{R}^r$:

$$\kappa(\mathbf{q}, \mathbf{k}) = \phi(\mathbf{q})^\top \phi(\mathbf{k}).$$

Substitute that factorization into the weighted average and regroup terms:

$$\begin{aligned}
\mathbf{y}_t(\mathbf{q}) &= \frac{\sum_{\tau \leq t} [\phi(\mathbf{q})^\top \phi(\mathbf{k}_\tau)] \mathbf{v}_\tau}{\sum_{\tau \leq t} \phi(\mathbf{q})^\top \phi(\mathbf{k}_\tau)} \\
&= \frac{\phi(\mathbf{q})^\top \mathbf{S}_t}{\phi(\mathbf{q})^\top \mathbf{z}_t},
\end{aligned}$$

where

$$\mathbf{S}_t = \sum_{\tau \leq t} \phi(\mathbf{k}_\tau) \mathbf{v}_\tau^\top, \quad \mathbf{z}_t = \sum_{\tau \leq t} \phi(\mathbf{k}_\tau).$$

The sufficient state is the pair $(\mathbf{S}_t, \mathbf{z}_t)$, not $\mathbf{S}_t$ alone: the numerator needs accumulated value-weighted features and the denominator needs its normalizer. Each new pair updates that fixed-size state once. Any later query reads the same answer as a full traversal of all pairs, for this factorized kernel.

PLAN

1. Define the reusable `ms_phi` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Compare a factorized-kernel traversal with its running state.
 4. Check the claimed identities, shapes, or invariants.
 5. Report or visualize the measured result.
-

CODE

```

# [1]
def ms_phi(x):
    return np.where(x > 0, x + 1, np.exp(x))

# [2]
ms_S = np.zeros((ms_d, ms_V.shape[1]))
ms_z = np.zeros(ms_d)
# [3]
for ms_key, ms_value in zip(ms_K, ms_V, strict=True):
    ms_S += np.outer(ms_phi(ms_key), ms_value)
    ms_z += ms_phi(ms_key)
ms_streaming = (ms_S.T @ ms_phi(ms_q)) / (ms_z @ ms_phi(ms_q))
ms_weights = ms_phi(ms_K) @ ms_phi(ms_q)
ms_traversal = (ms_weights[:, None] * ms_V).sum(0) / ms_weights.sum()
# [4]
assert np.allclose(ms_streaming, ms_traversal)
# [5]
print("streaming equals traversal:", np.allclose(ms_streaming, ms_traversal))
print(f"max |difference|: {np.abs(ms_streaming-ms_traversal).max():.3e}")

```

```

streaming equals traversal: True
max |difference|: 3.469e-17

```

The equality is exact up to floating-point order: the full dataset traversal has collapsed into two running sums. With r and d_v fixed, storage no longer grows with the prefix. This is the honest content behind “Transformers are RNNs” for causal **linear** attention: the recurrent state returns, dignified, as a sufficient statistic of a regression.

There is a price. Ordinary softmax’s exponential dot-product kernel does not have the finite exact feature map used above. Changing to a finite factorization changes the kernel, or approximates it. Collisions in (S_t, z_t) cannot be undone by revisiting a discarded row. The traversal equality is exact for the chosen factorized kernel, not a proof that finite-state linear attention reproduces exact softmax for arbitrary prefixes.

Solver 3: take one gradient step and get the delta rule

Now restrict the predictor to a scalar linear map $M(\mathbf{q}) = \mathbf{h}^\top \mathbf{q}$ and let the solver take one SGD step on only the newest pair. Its instantaneous loss and gradient are

$$\ell_t(\mathbf{h}) = \frac{1}{2}(\mathbf{h}^\top \mathbf{k}_t - v_t)^2, \quad \nabla_{\mathbf{h}} \ell_t = \mathbf{k}_t(\mathbf{k}_t^\top \mathbf{h} - v_t).$$

Substituting the gradient into the update gives the delta recurrence in three lines:

$$\begin{aligned} \mathbf{h}_t &= \mathbf{h}_{t-1} - \eta_t \mathbf{k}_t(\mathbf{k}_t^\top \mathbf{h}_{t-1} - v_t) \\ &= (\mathbf{I} - \eta_t \mathbf{k}_t \mathbf{k}_t^\top) \mathbf{h}_{t-1} + \eta_t v_t \mathbf{k}_t. \end{aligned}$$

 PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify that one SGD step is the delta recurrence.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
ms_eta = 0.3
ms_h = ms_rng.standard_normal(ms_d)
ms_k = ms_rng.standard_normal(ms_d)
ms_v = ms_rng.standard_normal()
ms_sgd = ms_h - ms_eta * ms_k * (ms_k @ ms_h - ms_v)
ms_delta = ((np.eye(ms_d) - ms_eta * np.outer(ms_k, ms_k)) @ ms_h
            + ms_eta * ms_v * ms_k)

# [2]
assert np.allclose(ms_sgd, ms_delta)

# [3]
print("SGD equals Delta recurrence:", np.allclose(ms_sgd, ms_delta))
print(f"max |difference|: {np.abs(ms_sgd-ms_delta).max():.3e}")
```

```
SGD equals Delta recurrence: True
max |difference|: 4.441e-16
```

For vector values, store $\mathbf{H}_t \in \mathbb{R}^{d_v \times d_k}$ and replace the update by

$$\mathbf{H}_t = \mathbf{H}_{t-1} + \eta_t (\mathbf{v}_t - \mathbf{H}_{t-1} \mathbf{k}_t) \mathbf{k}_t^\top.$$

Each outer product writes the residual left by the current key. DeltaNet turns that classical online least-squares step into a sequence layer. The recurrence also exposes a state-space form: its transition is $\mathbf{A}_t = \mathbf{I} - \eta_t \mathbf{k}_t \mathbf{k}_t^\top$ and its input write is $\eta_t \mathbf{v}_t \mathbf{k}_t$. The state transition is therefore determined by the current token's learned key.

Forgetting needs one more careful distinction. Exponential history weights $w_{t,\tau} = \gamma^{t-\tau}$ define an exponentially weighted least-squares *objective*, but its exact minimizer generally requires a recursive least-squares covariance state. Writing $\gamma \mathbf{H}_{t-1}$ into a one-step delta recurrence is a fading-memory solver choice, not an algebraic consequence of those weights. Gates can make that retention and the step size depend on the current token.

This is the derivation-first connection to the wider SSM family. Mamba makes parts of its state transition, input injection, and readout depend on the token; in the present language, those mechanisms play roles analogous to a learnable retention-and-write schedule. Gated DeltaNet adds learned gating around delta-style updates. That is an interpretive bridge, not a claim that Mamba is literally obtained by differentiating the shared regression objective. Structured SSMs also use different state parameterizations and can have costs below the dense outer-product ledger here.

The price list

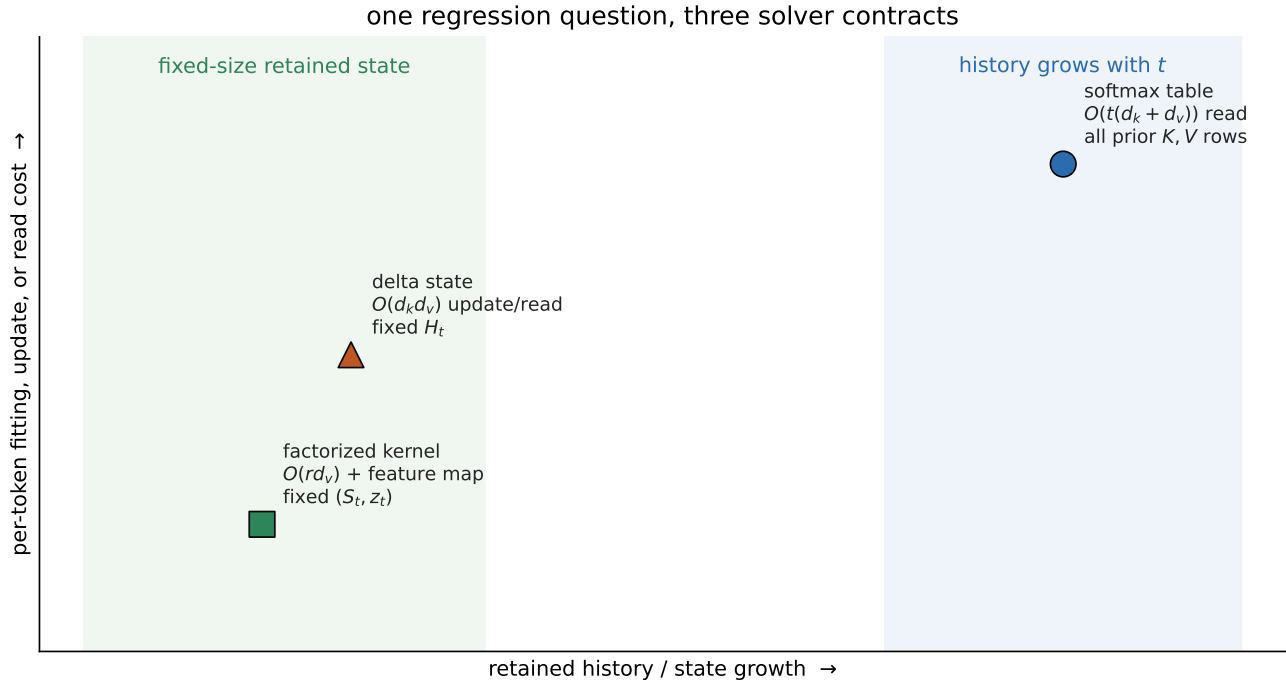


Figure TTR.1: The solver map compares retained state and per-token work. Its positions are qualitative; the formulas and direct labels carry the exact ledger.

The asymptotic labels in Figure TTR.1 name this dense educational setup, not every implementation. Feature width r , head dimensions, low-rank structure, convolutions, and hardware schedules change constants and sometimes orders. Most importantly, “retains every row” does not mean “recalls every association exactly.” Softmax still returns a convex mixture whose quality depends on keys, temperature, and interference.

⚠️ Trap: attention’s memory is a dataset, not a hidden box

Attention does not *have* a separate memory in the recurrent sense. The attention operation is the estimator; during causal decoding, the KV cache is the observed key–value dataset it must retain. A nonparametric query rule keeps growing data so it can form a new local fit. A parametric state instead compresses those data into a fixed-size object. FlashAttention changes how the uncompressed fit is scheduled; it does not turn it into the compressed contract in the second or third row.

Mechanism test: recall under a fixed capacity

Figure TTR.1 makes a prediction we can test without pretending to train a language model. Store N random unit key–value pairs, then query every stored key. Decode a read by the nearest stored value. Exact softmax attention keeps all pairs. A dense delta state has d^2 numbers regardless of N . A selective delta arm receives one extra priority bit and writes ordinary pairs at only one tenth strength, so it can choose what to sacrifice.

The study was designed on seed branches 0 and 1. Those branches fixed $\eta = 0.20$, the 25% priority rate, the 0.10 ordinary write gate, dimensions (8, 16, 32), and loads $N/d \in (0.5, 1, 2, 4, 8)$. Only then was disjoint branch 2 opened for 30 endpoint repetitions. All methods in a trial receive the same keys, values, order, queries, and decoder. The endpoint code below is the frozen protocol; it runs on CPU in under a second on the reference machine.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `ms_unit` and `ms_decode`.
 3. Define the reusable helpers: `ms_trial` and `ms_panel`.
 4. Run the sealed synthetic recall-under-capacity study.
-

CODE

```
# [1]
MS_SEED, MS_ETA, MS_GATE = 6050, 0.20, 0.10
MS_DIMS, MS_LOADS, MS_REPEATS = (8, 16, 32), (0.5, 1, 2, 4, 8), 30

# [2]
def ms_unit(x):
    return x / np.linalg.norm(x, axis=1, keepdims=True)

def ms_decode(prediction, values):
    return np.argmax(prediction @ values.T, axis=1)

# [3]
def ms_trial(rng, repeat, d, load):
    n = max(4, round(load * d))
    keys = ms_unit(rng.standard_normal((n, d)))
    values = ms_unit(rng.standard_normal((n, d)))
    priority = np.zeros(n, dtype=bool)
    priority[rng.permutation(n)[:max(1, round(0.25 * n))]] = True

    scores = 32.0 * (keys @ keys.T)
    scores -= scores.max(axis=1, keepdims=True)
    weights = np.exp(scores); weights /= weights.sum(axis=1, keepdims=True)
    attention = weights @ values

    def delta(write_gate):
        state = np.zeros((d, d))
        for key, value, gate in zip(keys, values, write_gate, strict=True):
            state += MS_ETA * gate * np.outer(value - state @ key, key)
        return keys @ state.T

    plain = delta(np.ones(n))
    selective = delta(np.where(priority, 1.0, MS_GATE))
```

```

truth = np.arange(n)
correct = lambda pred: ms_decode(pred, values) == truth
a_ok, d_ok, s_ok = correct(attention), correct(plain), correct(selective)
ordinary = ~priority
return [repeat, d, n, load, a_ok.mean(), ((attention-values)**2).mean(),
        d_ok.mean(), d_ok[priority].mean(), d_ok[ordinary].mean(),
        s_ok.mean(), s_ok[priority].mean(), s_ok[ordinary].mean()]

def ms_panel(tag, repeats):
    seeds = np.random.SeedSequence([MS_SEED, tag]).spawn(
        repeats * len(MS_DIMS) * len(MS_LOADS))
    rows, index = [], 0
    for repeat in range(repeats):
        for d in MS_DIMS:
            for load in MS_LOADS:
                rows.append(ms_trial(np.random.default_rng(seeds[index]),
                                    repeat, d, load))
                index += 1
    return np.asarray(rows)

# Tags 0 and 1 were development; tag 2 remained sealed until settings were fixed.
ms_development = np.vstack([ms_panel(tag, 5) for tag in (0, 1)])
assert np.all(ms_development[:, 4] == 1.0)
ms_endpoint = ms_panel(2, MS_REPEATS)
assert np.all(ms_endpoint[:, 4] == 1.0)

print("study_seed=6050; development_tags=(0, 1); sealed_endpoint_tag=2")
print("endpoint_trials=30 x 3 dimensions; d=(8, 16, 32); "
      "N/d=(0.5, 1, 2, 4, 8)")
print("fixed: eta=0.20; priority_fraction=0.25; ordinary_write_gate=0.10")
print("load attention Delta gated-all gated-priority gated-ordinary")
# [4]
for load in MS_LOADS:
    block = ms_endpoint[ms_endpoint[:, 3] == load]
    means = block[:, [4, 6, 9, 10, 11]].mean(0)
    print(f"{load:>4.1f} " + " ".join(f"{value:.3f}" for value in means))

hard = ms_endpoint[ms_endpoint[:, 3] == 8]
priority_gain = hard[:, 10] - hard[:, 7]
ordinary_gain = hard[:, 11] - hard[:, 8]
print(f"N/d=8 gated-minus-Delta priority: {priority_gain.mean():+.3f} "
      f"(SD {priority_gain.std(ddof=1):.3f})")
print(f"N/d=8 gated-minus-Delta ordinary: {ordinary_gain.mean():+.3f} "
      f"(SD {ordinary_gain.std(ddof=1):.3f})")
print(f"attention top-1 failures=0/{int(ms_endpoint[:, 2].sum())}")
print(f"mean value MSE={ms_endpoint[:, 5].mean():.2e}; "
      f"max trial MSE={ms_endpoint[:, 5].max():.2e}")

```

```
study_seed=6050; development_tags=(0, 1); sealed_endpoint_tag=2
```

```

endpoint_trials=30 x 3 dimensions; d=(8, 16, 32); N/d=(0.5, 1, 2, 4, 8)
fixed: eta=0.20; priority_fraction=0.25; ordinary_write_gate=0.10
load  attention  Delta  gated-all  gated-priority  gated-ordinary
0.5   1.000   0.988   0.471   1.000   0.294
1.0   1.000   0.910   0.352   0.992   0.139
2.0   1.000   0.657   0.282   0.982   0.048
4.0   1.000   0.338   0.228   0.848   0.022
8.0   1.000   0.134   0.138   0.522   0.010
N/d=8 gated-minus-Delta priority: +0.387 (SD 0.156)
N/d=8 gated-minus-Delta ordinary: -0.124 (SD 0.065)
attention top-1 failures=0/26040
mean value MSE=5.43e-06; max trial MSE=4.55e-04

```

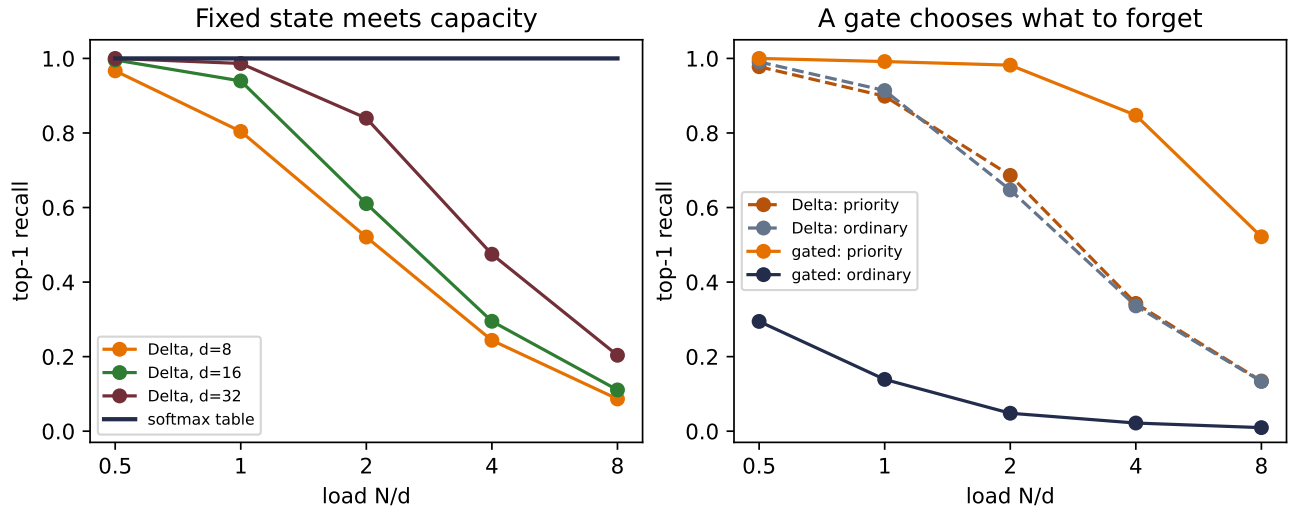


Figure TTR.2: A sealed synthetic mechanism test, not a language-model or state-space benchmark. The uncompressed table keeps perfect top-1 identification in this sharp-softmax construction, while the fixed delta state loses associations as N/d grows. The selective arm receives an unmatched priority bit and trades ordinary recall for priority recall. Keys, values, order, queries, decoder, and state width are matched; stored data, side information, and inference arithmetic are not.

At $N/d = 8$, the gate raises priority recall over the plain delta state by 0.387 on average and lowers ordinary recall by 0.124. That is selective forgetting made visible, not a claim that the gated arm is globally better. Overall gated recall is 0.138 while plain Delta recall is 0.134 at that load. The useful result is the trade: given an extra signal, a fixed state can spend capacity differently.

The fixed-size state could not hold everything in Chapter 10. It still cannot. Now we know the price list on which that failure was one entry. Keeping the dataset buys uncompressed access at growing cost; sufficient statistics buy an exact stream for a restricted kernel; online updates buy a steerable fixed state and accept interference. The RNN state has returned as a statistical choice rather than a design we were supposed to declare dead.

i Check yourself

Close the book for one minute and reconstruct the three memory contracts.

- Which solver retains the original key–value rows?
- Why does a factorized kernel need both S_t and z_t ?
- What information can a fixed delta state overwrite as its load grows?

Okay, so — the solver is part of the architecture

1. **The objective alone does not specify the memory system.** Views, history weights, function class, regularizer, and solver jointly define the result.
2. **Softmax attention keeps a query-dependent dataset.** It preserves every key–value row and pays a growing read cost.
3. **A finite kernel feature map admits sufficient state.** The pair (S_t, z_t) reproduces the chosen factorized-kernel traversal without retaining raw rows.
4. **One SGD step yields a delta update.** Its fixed matrix writes residuals online, which makes capacity and interference explicit.
5. **Selective retention spends a budget.** The sealed recall study shows a gate trading ordinary recall for priority recall; it does not establish a universal architecture ranking.

Sources and further reading

- Katharopoulos et al., *Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention*: the recurrent sufficient-state form for causal linear attention.
- Schlag, Irie, and Schmidhuber, *Linear Transformers Are Secretly Fast Weight Programmers*: the connection among linear attention, fast weights, and the delta update.
- Gu and Dao, *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*: input-dependent state-space parameters; this interlude uses selectivity as an analogy, not an exact regression equivalence.
- Yang et al., *Gated Delta Networks: Improving Mamba2 with Delta Rule*: learned gating around delta-style associative updates.
- Wang, Shi, and Fox, *Test-Time Regression: A Unifying Framework for Designing Sequence Models with Associative Memory*: the shared regression objective and solver-design perspective.

Exercises

1. **(Pencil.)** Starting from the local-constant objective, derive its stationarity equation and normalized solution. Identify exactly where the local-constant restriction enters. Why would an unrestricted function with no regularizer not imply the same average?
2. **(Code.)** Reproduce the factorized-kernel sufficient-state identity with a positive feature map of your choice. Assert that full traversal and the running pair (S_t, z_t) agree for every prefix. Which assertion fails if you omit z_t ?

3. **(Pencil.)** Derive the delta recurrence from one SGD step. Then multiply the old state by γ . Explain why this is a fading-memory solver choice rather than the exact minimizer of exponentially weighted least squares.
4. **(Code.)** Sweep the capacity study over at least five state widths d . Preserve the sealed-seed protocol, plot recall against both N and N/d , and report where the curves align. Do not call top-1 identification exact value recall.
5. **(Audit.)** For each solver in the cost diagram, record state size, update cost, query cost, and information that cannot be reconstructed. Identify what is compute-matched and storage-matched, then design one rematch for a claim you would be willing to make.

15. The BERT Moment: Pretraining as the New Regime

Chapter 9 left a decision rule unfinished. Transfer should pay when **labels are scarce, the target is feature-hungry, and source coverage is useful at a matched scale**. Images supplied a natural source task—learn visual features from many labeled photographs, then adapt them. What should play the same role for language, where task labels are expensive but raw text is everywhere?

Text can write its own questions. Hide *bank* in “the boat reached the bank,” and the surrounding words contain the answer. Hide the same spelling in “the bank approved the loan,” and a different neighborhood supplies a different answer. No person had to label either sense. The answer is already inside the sequence; corruption turns it into a question.

That move depends on the seed from **Chapter 14: visibility is a modeling decision**. A next-token model used a causal triangle because looking right would reveal its target. A representation model can instead let every nonpadding token consult both sides—provided selected targets are treated so copying cannot solve the whole task. This chapter separates those controls, derives **masked language modeling (MLM)**, and shows how pretraining changed the default workflow from “initialize and train” to “pretrain, then adapt.”

15.1. Separate controls, not one mask

Bidirectional Encoder Representations from Transformers—**BERT**—uses a few special tokens. [CLS] is a learned summary placeholder, [SEP] marks a boundary, and [PAD] fills unused batch slots. Consider six positions:

[CLS], the, bank, rose, [SEP], [PAD].

There are four distinct decisions in play.

1. **Attention visibility** decides which key positions each query may use. A causal decoder blocks future keys; a BERT encoder permits all nonpadding keys.
2. **The MLM target selector** chooses which ordinary positions contribute loss.
3. **The corruption branch** replaces some selected tokens by [MASK], replaces some by random tokens, and deliberately leaves some unchanged.
4. **The padding mask** prevents real tokens from routing information from placeholder slots used only to make batch shapes rectangular.

PLAN

1. Construct causal and full visibility beside a separate MLM selector.

CODE

```
# [1]
import copy
import itertools

import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F

tokens = ["[CLS]", "the", "bank", "rose", "[SEP]", "[PAD]"]
nonpadding = np.array([1, 1, 1, 1, 1, 0], dtype=bool)
causal = np.tril(np.ones((6, 6), dtype=int)) * nonpadding[None, :]
full = np.ones((6, 6), dtype=int) * nonpadding[None, :]
selector = np.zeros((1, 6), dtype=int)
selector[0, 2] = 1
```

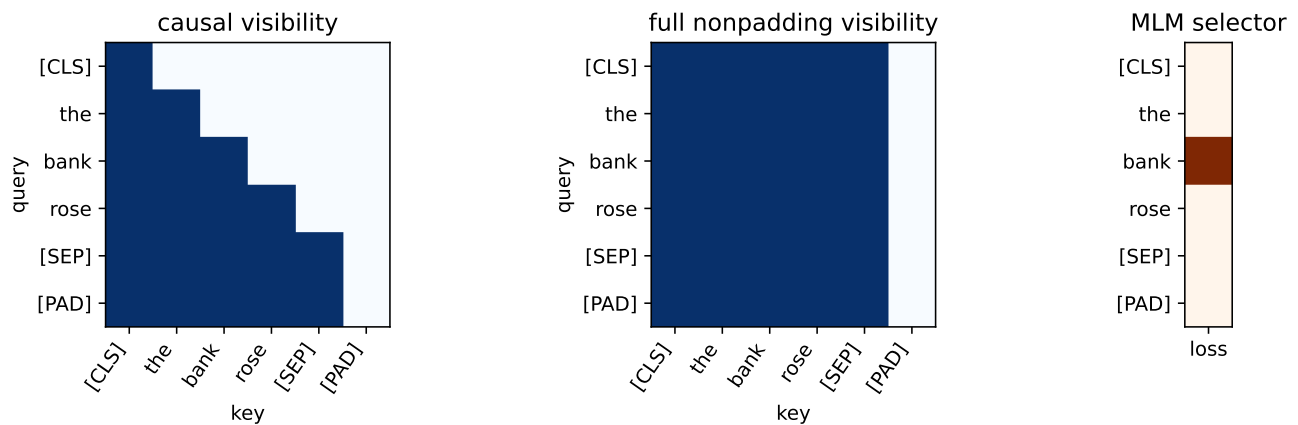


Figure 15.1.: Visibility and supervision are separate controls. A causal decoder sees a lower triangle (left); BERT-style attention sees every nonpadding key (center). The MLM selector marks one selected target position (right).

The center matrix contains no future-blocking triangle. Every nonpadding query can route from every nonpadding key—including itself. That is what *bidirectional* means here: a token's representation can jointly use left and right context in every layer. It does not mean running two separate recurrent networks, and it does not define a left-to-right probability for the whole sentence.

Only padding *keys* are blocked in the drawing. A [PAD] query row can still produce an output, but downstream computation ignores that row; the important guarantee is that real queries cannot retrieve padding as information.

Now suppose *bank* is selected and replaced:

[CLS] the [MASK] rose [SEP].

Every ordinary token may attend to [MASK], because [MASK] is an input token—not an attention prohibition. Only the selector says that position 2 contributes MLM loss. The padding mask separately blocks the final slot as a key. Reusing *mask* for visibility, selection, corruption, and padding is convenient shorthand—and a reliable source of bugs.

The copy trap

If full attention received the original token at *every* selected position, the model could copy answers through the residual stream. BERT’s policy corrupts 90% of selected sites, so copying cannot solve the whole task; the deliberate 10% unchanged branch has a separate purpose below. Conversely, replacing a token by [MASK] does not make it invisible—its neighbors still see the placeholder and one another. Always audit visibility, selection, corruption, and padding separately.

15.2. Text supplies supervision

In ordinary supervised learning, an external label y accompanies input x . Masked language modeling derives both from one unlabeled sequence: the corrupted sequence is the input, and the original selected tokens are the targets. This is *self-supervised learning*—the supervision is constructed from the data rather than supplied by a human annotator.

Self-supervised does not mean target-free

The method still has labels and a loss. What changes is their origin. The clean text supplies a target for every selected position, so a large corpus can create many training examples without a task-specific annotation campaign.

This idea was part of a broader 2018 convergence, not a solitary invention.

Work	Pretraining signal	What moved downstream
ELMo	Forward and backward LSTM language models	Fixed contextual features
ULMFiT	Causal language modeling	The entire adapted language model
GPT	Causal Transformer language modeling	A pretrained Transformer plus task head
BERT	Masked tokens in full context, plus NSP	A deeply bidirectional Transformer encoder plus task head

ELMo demonstrated that a word’s useful vector should depend on its sentence. ULMFiT supplied a robust recipe for adapting a pretrained language model rather than freezing it. GPT paired Transformer pretraining with supervised fine-tuning. BERT’s decisive change was to pretrain every encoder layer with joint left-and-right context. The workflow mattered as much as the architecture: learn broadly from raw text first, then spend scarce labels on adaptation.

15.3. The masked-language-model objective

Let us pin the idea to shapes before writing code. Let a tokenized sequence be

$$x = (x_1, \dots, x_T), \quad x_t \in \{1, \dots, |V|\},$$

where T is sequence length and V is the vocabulary. Choose a random set M of eligible positions, then apply a corruption operator c to obtain $\tilde{x} = c(x, M)$. A bidirectional encoder with parameters θ produces

$$H = f_\theta(\tilde{x}) \in \mathbb{R}^{T \times d}.$$

An output head maps each selected hidden state to vocabulary logits. The MLM loss is

$$\mathcal{L}_{\text{MLM}}(\theta) = -\frac{1}{|M|} \sum_{i \in M} \log p_\theta(x_i \mid \tilde{x}).$$

The condition is $\tilde{x}$, the one jointly corrupted sequence—not a different clean sequence with only x_i removed for every term. With batches, token IDs have shape (B, T) , hidden states (B, T, d) , and dense vocabulary logits $(B, T, |V|)$. Boolean selection gathers $|M|$ rows, so cross-entropy receives selected logits $(|M|, |V|)$ and targets $(|M|,)$. Unselected positions can affect the representations but contribute no direct MLM loss.

15.3.1. Fifteen percent, then 80/10/10

Original BERT selects 15% of eligible tokens. Conditional on selection:

- 80% are replaced by [MASK];
- 10% are replaced by a random vocabulary token;
- 10% remain unchanged.

For 1,000 eligible positions, the expected flow is therefore

$$1000 \longrightarrow 150 \longrightarrow \begin{cases} 120 & \text{[MASK],} \\ 15 & \text{random token,} \\ 15 & \text{unchanged.} \end{cases}$$

All 150 selected positions are scored against their original token. The 120 mask replacements are 12% of all eligible positions, not the complete supervision rate. Random ordinary-looking replacements force the encoder to cross-check a token against its context rather than trusting the visible identity. The

unchanged branch prevents the opposite shortcut—the assumption that every selected ordinary-looking token must be wrong. It also reduces the mismatch between pretraining, where [MASK] exists, and downstream inputs, where it ordinarily does not. An unchanged selected token exposes its answer on purpose; only about 1.5% of all eligible positions follow that branch, and the encoder is not told which normal-looking tokens were selected.

PLAN

1. Compute the expected two-stage policy counts.
 2. Build and audit the five Boolean ledgers on one illustrative sequence.
 3. Report the policy totals.
-

CODE

```
# [1]
eligible_count = 1_000
selected_count = round(0.15 * eligible_count)
branch_counts = np.array([120, 15, 15])

# [2]
ledger_tokens = [
    "[CLS]", "the", "quiet", "bank", "rose", "after",
    "the", "rain", "today", "[SEP]", "[PAD]", "[PAD]",
]
eligible = np.array([False] + [True] * 8 + [False] * 3)
selected = np.array(
    [False, False, True, False, True, False, False, True, True, False, False, False]
)
mask_sites = np.array(
    [False, False, True, False, False, False, False, True, False, False, False,
    ↪ False]
)
random_sites = np.array(
    [False, False, False, False, True, False, False, False, False, False, False,
    ↪ False]
)
unchanged_sites = selected & ~mask_sites & ~random_sites
assert np.all(selected <= eligible)
assert np.array_equal(selected, mask_sites | random_sites | unchanged_sites)
assert not np.any(mask_sites & random_sites)

# [3]
print(f"direct loss targets: {branch_counts.sum()}")
print(f"[MASK] share of all eligible positions: {branch_counts[0] / 1000:.1%}")
```

```
direct loss targets: 150
[MASK] share of all eligible positions: 12.0%
```

one sequence, five Boolean controls

	[CLS]	the	quiet	bank	rose	after	the	rain	today	[SEP]	[PAD]	[PAD]	
eligible	*	T	T	T	T	T	T	T	T	*	*	*	eligibility
selected	*	*	T	*	T	*	*	T	T	*	*	*	supervision
mask_sites	*	*	T	*	*	*	*	T	*	*	*	*	corruption
random_sites	*	*	*	*	T	*	*	*	*	*	*	*	
unchanged_sites	*	*	*	*	*	*	*	*	T	*	*	*	

visibility is separate: every query may attend to every nonpadding key

Figure 15.2.: Five Boolean ledgers keep BERT’s controls separate. Eligibility excludes special and padding tokens; selection identifies direct loss targets; three disjoint corruption rows partition the selected sites into mask, random, and unchanged branches. Attention visibility remains full over nonpadding keys and is not encoded by any of these rows. The illustrated branch allocation is deliberately small; the 15% and 80/10/10 rates are population policies, not proportions to infer from twelve positions.

Original BERT pre-generated several masked versions of its training examples; RoBERTa later redrew masks as examples were presented. Our controlled lab uses the latter *dynamic corruption* choice—the same sentence can ask different questions on later passes. Reproducibility therefore requires a separate random generator. Seeding model initialization while leaving the corruption stream implicit is not an exact experiment.

💡 Audit the selector before the loss

Print the number of eligible and selected positions, verify that special and padding tokens are excluded, and inspect the realized 80/10/10 proportions over many batches. Then assert that the selected logit and target shapes agree. A loss can decrease while a selector silently trains on padding or on the wrong tokens.

15.4. From a Transformer encoder to BERT

The original BERT input at position i adds three learned vectors:

$$z_i = e_{\text{token}(i)} + e_{\text{position}(i)} + e_{\text{segment}(i)}.$$

The token comes from a **WordPiece** vocabulary: frequent character sequences are stored whole, while rarer words can be assembled from reusable subword pieces.

💡 Practice bridge (non-examinable): make the vocabulary learnable

Chapter 10 fixed its vocabulary at individual characters. A full-word vocabulary shortens sequences but gives every rare spelling its own entry. Subword tokenization chooses a middle contract. **Byte Pair Encoding (BPE)** begins with small symbols, counts adjacent pairs in the training corpus, merges the most frequent pair, and repeats until it reaches a chosen vocabulary budget.

This original toy implementation starts from characters plus an end-of-word marker. Corpus frequencies weight the pair counts; lexicographic order breaks exact ties so the result is reproducible.

PLAN

1. Define the reusable `merge_pair` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Learn five subword merges from weighted word counts.
 4. Report or visualize the measured result.
-

CODE

```
from collections import Counter

# [1]
def merge_pair(
    symbols: tuple[str, ...], pair: tuple[str, str]
) -> tuple[str, ...]:
    merged: list[str] = []
    index = 0
    while index < len(symbols):
        if index + 1 < len(symbols) and symbols[index:index + 2] == pair:
            merged.append("".join(pair))
            index += 2
        else:
            merged.append(symbols[index])
            index += 1
    return tuple(merged)

# [2]
corpus = {"low": 5, "lower": 2, "newest": 6, "widest": 3}
words = {tuple(word) + ("</w>",): count for word, count in corpus.items()}
merges: list[tuple[tuple[str, str], int]] = []

# [3]
for _ in range(5):
    counts: Counter[tuple[str, str]] = Counter()
    for symbols, frequency in words.items():
        for pair in zip(symbols, symbols[1:]):
            counts[pair] += frequency
    maximum = max(counts.values())
```

```

best = min(pair for pair, count in counts.items() if count == maximum)
merges.append((best, maximum))
words = {merge_pair(symbols, best): count for symbols, count in words.items()}

# [4]
print("merges:", [( "+" .join(pair), count) for pair, count in merges])
print("lower:", " | ".join(next(word for word in words if "r" in word)))

```

```

merges: [('e+s', 9), ('es+t', 9), ('est+</w>', 9), ('l+o', 7), ('lo+w', 7)]
lower: low | e | r | </w>

```

i Reading the learned pieces (non-examinable)

The first three merges build `est</w>` from nine weighted occurrences; the next two build `low` from seven. The word `lower` is therefore represented as `low | e | r | </w>` even though the whole word was never granted its own token. This is the mechanism, not BERT’s tokenizer: WordPiece uses a different merge-scoring criterion, and practical byte-level tokenizers begin from bytes rather than the toy character alphabet. The shared idea is the important one: the vocabulary becomes a learned compromise between sequence length and reusable coverage.

Learned absolute position embeddings replace Chapter 14’s fixed sinusoidal table. Segment A/B embeddings mark which of two packed text spans supplied a token; they do not block cross-segment attention. `[CLS]` begins the sequence, and `[SEP]` ends each span:

`[CLS] A [SEP] B [SEP]`.

The resulting sequence passes through a stack of full-context Transformer encoder blocks. Original BERT uses the post-LayerNorm order introduced as the historical Transformer in [Chapter 14](#). Its two published sizes were:

Model	Blocks L	Width d	Heads	Parameters
BERT Base	12	768	12	about 110 million
BERT Large	24	1,024	16	about 340 million

Pretraining used BooksCorpus and English Wikipedia—about 3.3 billion words in the paper’s accounting. The scale is historically important—but the reusable idea is the separation of stages:

raw text $\xrightarrow{\text{self-supervised pretraining}}$ general encoder $\xrightarrow{\text{labeled adaptation}}$ task model.

Original BERT combined MLM with **next sentence prediction** (NSP). Half the training pairs used the actual following segment and half used a random segment; the `[CLS]` state predicted which kind it saw—its direct pretraining assignment. MLM alone gives `[CLS]` no token-reconstruction target, although information can still route through it. Later recipes such as RoBERTa showed that strong BERT-style training did not require NSP in their settings. That is a recipe result, not proof that a sentence-relation objective can never help. This chapter studies the MLM half because it supplies the cleanest bridge from visibility to representation learning.

15.4.1. [CLS] is a learned meeting place

[CLS] is an ordinary learned token with an unusual assignment: place it first, allow it to attend through every layer, then give its final hidden state to a sequence-level head. For K classes,

$$\ell = W_{\text{cls}} h_{\text{CLS}} + b, \quad W_{\text{cls}} \in \mathbb{R}^{K \times d}.$$

For token labeling, apply a head to every h_i . For extractive question answering, score candidate start and end positions. The encoder interface stays the same while the small task head changes.

[CLS] is not born with sentence meaning. Pretraining and downstream gradients teach it what information to gather. Chapter 16 will reuse this learned-summary pattern after replacing words by image patches.

15.5. Fine-tuning means the backbone moves

Let θ_0 denote pretrained encoder weights and ϕ a newly initialized task head. Standard BERT fine-tuning solves

$$(\theta^*, \phi^*) = \arg \min_{\theta, \phi} \sum_{(x, y) \in \mathcal{D}_{\text{task}}} \mathcal{L}_{\text{task}}(g_{\phi}(f_{\theta}(x)), y),$$

starting at $\theta = \theta_0$. Gradients update both θ and ϕ . Freezing θ_0 and training only ϕ is a useful *linear probe* or feature probe, but it is not the standard fine-tuning procedure. This distinction matters when the task needs the representation itself to move.

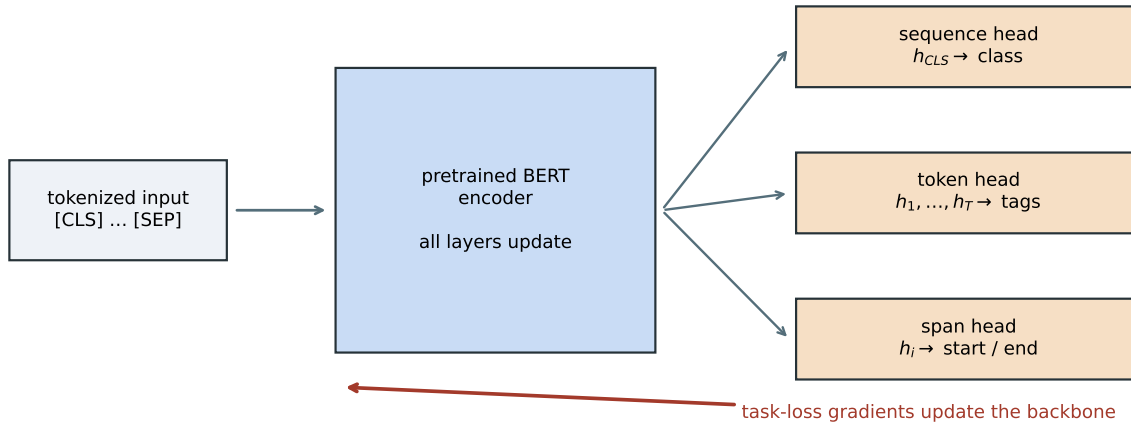


Figure 15.3.: One pretrained encoder supports several task interfaces. In standard fine-tuning, task-loss gradients pass through the new head and continue through the encoder; stopping them at the head would make the run a frozen probe.

Fine-tuning also creates experimental obligations. The scratch and pretrained arms must share labeled examples, head initialization, minibatch order, update count, evaluation, and downstream hyperparameters. The pretrained arm has already received extra unlabeled-data compute, so the comparison estimates the value of pretraining—not a total-compute advantage.

15.6. A controlled transfer lab

Chapter 9’s three gates are easy to repeat and surprisingly hard to test. A first pre-test asked a tiny encoder to distinguish true continuations in this book from distant splices. The five-seed MLM gain was small, faded as labels increased, and lost decisively to a weighted lexical-overlap baseline. We rejected that task as the chapter’s affirmative experiment—the shallow baseline solved more of the problem than the representation story did.

The replacement lab is synthetic by design. It removes spelling, frequency, and downstream-context shortcuts so that source coverage is a controllable variable.

15.6.1. Build a latent regularity

Eighty randomly generated four-letter token strings are divided into two latent families. Family assignment is independent of spelling. Forty *covered* strings—20 per family—each appear eight times in an unlabeled MLM corpus. Forty more—20 per family—live in the input vocabulary but are withheld from MLM input sequences, random replacements, and prediction targets.

Each covered word receives eight source sentences. Six contain cues from its own family; two deliberately contain cues from the other family. One family appears near words such as *rain*, *soil*, *roots*, and *garden*; the other appears near *power*, *metal*, *gears*, and *workshop*. The 25% cue noise gives every covered string a mixture of supporting and opposing contexts—the aggregate family signal is statistical rather than pure.

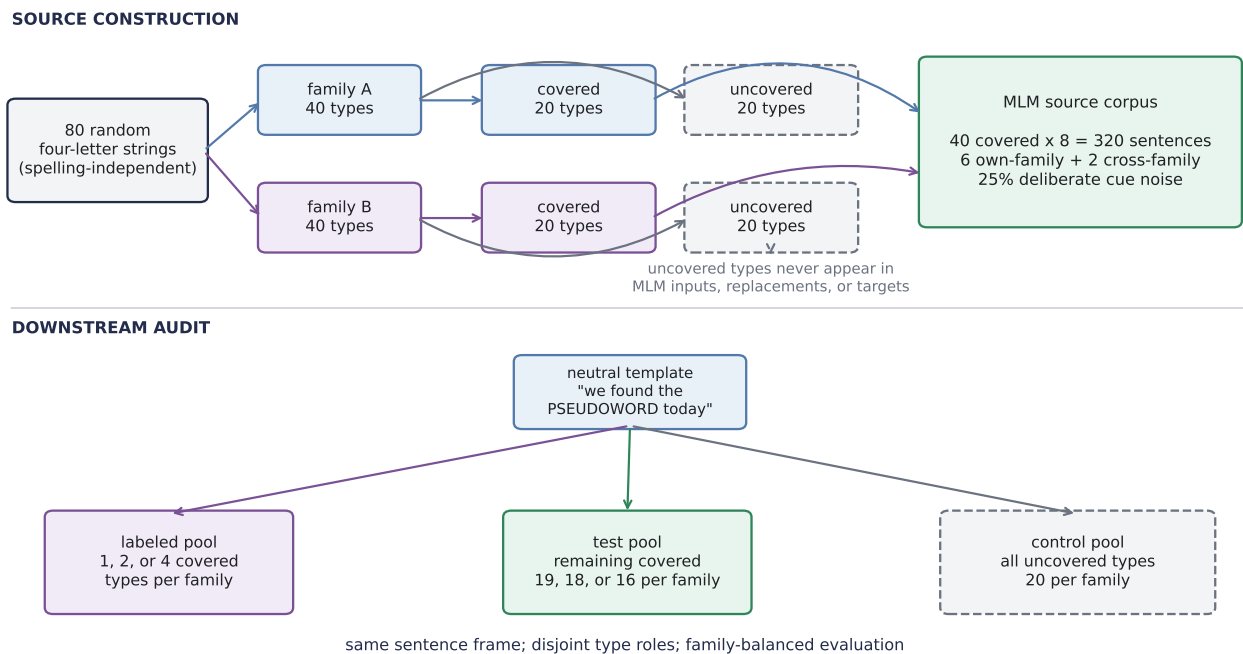


Figure 15.4.: The controlled transfer lab makes source exposure and downstream evaluation separate, auditable choices. Family is independent of spelling. Only the 40 covered types enter MLM inputs, random replacements, or targets; each contributes eight source sentences with six own-family and two cross-family cues. The neutral downstream template then separates labeled covered types, held-out covered test types, and all 40 source-invisible controls.

Read Figure 15.4 from top to bottom. Source exposure is decided before the neutral downstream sentence is formed, so the covered test pool measures transfer to unseen *types* and the uncovered pool audits what vocabulary membership alone can buy.

The downstream input is always the neutral, previously unseen template

“we found the PSEUDOWORD today.”

No family cue remains. A classifier reads the generated token’s hidden state and receives only 1, 2, or 4 labeled word types from each family. Evaluation uses the remaining covered types, never the labeled types. The 40 uncovered controls use the same neutral sentence. Source and target use the same atomic token granularity and similarly short sequences, making “matched scale” explicit rather than assuming it from a shared domain.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `source_sentences` and `encode`.
 3. Construct the controlled source and target distributions.
 4. Check the claimed identities, shapes, or invariants.
 5. Report or visualize the measured result.
-

CODE

```
# [1]
SEED = 6050
WIDTH = 48
HEADS = 4
FF_WIDTH = 96
BLOCKS = 2
MAX_LEN = 10
PRETRAIN_STEPS = 600
BATCH_SIZE = 64
LABEL_COUNTS = (1, 2, 4)
REPLICATE_SEEDS = tuple(range(6050, 6055))

torch.set_num_threads(4)
torch.use_deterministic_algorithms(True)

forms = [
    "".join(letters)
    for letters in itertools.product("bglmprstvz", "aeiou", "dknrst", "aeiou")
]
form_order = torch.randperm(
    len(forms), generator=torch.Generator().manual_seed(SEED)
)[:80]
token_strings = [forms[index] for index in form_order]
```

```

family_words = [token_strings[:40], token_strings[40:]]
covered_words = [words[:20] for words in family_words]
uncovered_words = [words[20:40] for words in family_words]

context_pairs = [
    ("rain", "power"),
    ("soil", "metal"),
    ("roots", "gears"),
    ("water", "fuel"),
    ("garden", "workshop"),
    ("grows", "moves"),
    ("blooms", "turns"),
    ("sunlight", "current"),
]

# [2]
def source_sentences(word: str, family: int) -> list[list[str]]:
    cue = [pair[family] for pair in context_pairs]
    other = [pair[1 - family] for pair in context_pairs]
    return [
        ["after", cue[0], "the", word, cue[5]],
        ["the", word, "needs", cue[3], "near", cue[1]],
        [cue[2], "support", "the", word],
        ["in", "the", cue[4], "the", word, cue[6]],
        ["we", "found", "the", word, "beside", cue[1]],
        [cue[0], "and", cue[7], "help", "the", word],
        ["the", word, "rests", "near", other[4]],
        ["today", "the", word, "follows", other[2]],
    ]

sentences: list[list[str]] = []
# [3]
for family, words in enumerate(covered_words):
    for word in words:
        sentences.extend(source_sentences(word, family))

specials = ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"]
ordinary = sorted(
    {token for sentence in sentences for token in sentence} | set(token_strings)
)
vocabulary = specials + [token for token in ordinary if token not in specials]
stoi = {token: index for index, token in enumerate(vocabulary)}
PAD, UNK, CLS, SEP, MASK = [stoi[token] for token in specials]

def encode(sentence: list[str]) -> torch.Tensor:
    ids = [CLS] + [stoi.get(token, UNK) for token in sentence] + [SEP]
    ids += [PAD] * (MAX_LEN - len(ids))
    return torch.tensor(ids[:MAX_LEN])

```

```

source_ids = torch.stack([encode(sentence) for sentence in sentences])
uncovered_ids = torch.tensor([
    stoi[word] for family in uncovered_words for word in family
])
source_token_ids = torch.unique(source_ids)
random_pool = source_token_ids[
    ~torch.isin(source_token_ids, torch.tensor([PAD, CLS, SEP]))
]

# [4]
assert len(sentences) == 320
assert len(vocabulary) == 115
assert not torch.isin(source_ids, uncovered_ids).any()
# [5]
print(f"source: {len(sentences)} sentences; vocabulary: {len(vocabulary)} tokens")
print(f"covered/uncovered token strings: 40 / {len(uncovered_ids)}")

```

```

source: 320 sentences; vocabulary: 115 tokens
covered/uncovered token strings: 40 / 40

```

This is an atomic word-level vocabulary, not WordPiece. That simplification is intentional—splitting the generated strings into subwords could let spelling leak family identity. The source generator—not natural language—defines the latent regularity, so the experiment can demonstrate a mechanism but cannot estimate real-world BERT performance.

15.6.2. A small BERT-style encoder

The encoder has two post-LayerNorm blocks, width 48, four heads, learned absolute positions, full nonpadding attention, and GELU feedforward layers. Here $\text{GELU}(x) = x\Phi(x)$, with Φ the standard-normal cumulative distribution—a smooth, input-dependent gate used by original BERT. The lab omits segment embeddings, NSP, and dropout. Its 43,920 encoder parameters make the study fast enough to repeat end to end, while preserving the relevant path from bidirectional attention to MLM to fine-tuning.

PLAN

1. Define the reusable helpers: `FullAttention`, `EncoderBlock`, and `TinyBertEncoder`.
 2. Define the reusable helpers: `MaskedLanguageModel` and `corrupt`.
 3. Prepare the inputs and fixed settings for the example.
 4. Define full attention, the encoder, and the MLM head.
-

CODE

```

# [1]
class FullAttention(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.qkv = nn.Linear(WIDTH, 3 * WIDTH)
        self.output = nn.Linear(WIDTH, WIDTH)

    def forward(
        self, x: torch.Tensor, visible: torch.Tensor
    ) -> torch.Tensor:
        batch, length, _ = x.shape
        head_width = WIDTH // HEADS
        qkv = self.qkv(x).view(
            batch, length, 3, HEADS, head_width
        )
        q, k, v = qkv.permute(2, 0, 3, 1, 4)
        scores = q @ k.transpose(-2, -1) / head_width**0.5
        scores = scores.masked_fill(
            ~visible[:, None, None, :], float("-inf")
        )
        weights = F.softmax(scores, dim=-1)
        mixed = (weights @ v).transpose(1, 2).reshape(
            batch, length, WIDTH
        )
        return self.output(mixed)

class EncoderBlock(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.attention = FullAttention()
        self.norm1 = nn.LayerNorm(WIDTH)
        self.ff = nn.Sequential(
            nn.Linear(WIDTH, FF_WIDTH),
            nn.GELU(),
            nn.Linear(FF_WIDTH, WIDTH),
        )
        self.norm2 = nn.LayerNorm(WIDTH)

    def forward(
        self, x: torch.Tensor, visible: torch.Tensor
    ) -> torch.Tensor:
        x = self.norm1(x + self.attention(x, visible))
        return self.norm2(x + self.ff(x))

class TinyBertEncoder(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.token_embedding = nn.Embedding(

```

```

        len(vocabulary), WIDTH, padding_idx=PAD
    )
    self.position_embedding = nn.Embedding(MAX_LEN, WIDTH)
    self.blocks = nn.ModuleList([
        EncoderBlock() for _ in range(BLOCKS)
    ])
    nn.init.normal_(self.token_embedding.weight, std=0.02)
    nn.init.normal_(self.position_embedding.weight, std=0.02)
    with torch.no_grad():
        self.token_embedding.weight[PAD].zero_()

    def forward(self, ids: torch.Tensor) -> torch.Tensor:
        positions = torch.arange(ids.shape[1], device=ids.device)
        hidden = (
            self.token_embedding(ids)
            + self.position_embedding(positions)
        )
        visible = ids != PAD
        for block in self.blocks:
            hidden = block(hidden, visible)
        return hidden

# [2]
class MaskedLanguageModel(nn.Module):
    def __init__(self, encoder: TinyBertEncoder) -> None:
        super().__init__()
        self.encoder = encoder
        self.dense = nn.Linear(WIDTH, WIDTH)
        self.norm = nn.LayerNorm(WIDTH)
        self.decoder = nn.Linear(WIDTH, len(vocabulary))

    def forward(
        self, ids: torch.Tensor, selected: torch.Tensor
    ) -> torch.Tensor:
        hidden = self.encoder(ids)[selected]
        hidden = self.norm(F.gelu(self.dense(hidden)))
        return self.decoder(hidden)

def corrupt(
    ids: torch.Tensor,
    generator: torch.Generator,
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    eligible = (ids != PAD) & (ids != CLS) & (ids != SEP)
    selected = eligible & (
        torch.rand(ids.shape, generator=generator) < 0.15
    )
    if not selected.any():
        rows, columns = torch.where(eligible)
        choice = torch.randint(

```

```

        len(rows), (1,)), generator=generator
    )
    selected[rows[choice], columns[choice]] = True

    draw = torch.rand(ids.shape, generator=generator)
    mask_sites = selected & (draw < 0.8)
    random_sites = selected & (draw >= 0.8) & (draw < 0.9)
    corrupted = ids.clone()
    corrupted[mask_sites] = MASK
    random_choices = torch.randint(
        len(random_pool), ids.shape, generator=generator
    )
    random_ids = random_pool[random_choices]
    corrupted[random_sites] = random_ids[random_sites]

    unchanged_sites = selected & ~mask_sites & ~random_sites
    counts = torch.tensor([
        mask_sites.sum().item(),
        random_sites.sum().item(),
        unchanged_sites.sum().item(),
    ])
    assert not (selected & ~eligible).any()
    assert not torch.isin(corrupted, uncovered_ids).any()
    return corrupted, selected, counts

# [3]
encoder_parameters = sum(
    parameter.numel() for parameter in TinyBertEncoder().parameters()
)
# [4]
print(f"encoder parameters: {encoder_parameters:,}")

```

```
encoder parameters: 43,920
```

The MLM decoder is deliberately *untied* from the input embedding table. The original released BERT code explicitly tied those weights, but tying would update an uncovered input row through the vocabulary softmax even when that token never appeared. The custom control is stricter—random replacements are drawn only from source tokens, and the code later asserts that every uncovered input embedding remains bitwise unchanged through pretraining. Untied decoder rows for the controls still participate as negative softmax classes, but they are discarded downstream and never expose a control token’s context or identity to the encoder input.

15.6.3. Pretrain, then adapt

For each seed from 6050 through 6054, one encoder is saved before training and the same encoder receives 600 MLM updates. The two downstream arms then receive the same labeled word types, identical classifier-head tensors, 160 full-batch updates, and the same learning rates. All encoder weights move in both arms. Each seed therefore repeats initialization, corruption, MLM, label selection, and fine-tuning—not merely the last classifier fit.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable helpers: `pretrain`, `neutral_sentence`, and `WordClassifier`.
3. Define the reusable helpers: `fit_classifier` and `classifier_accuracy`.
4. Pretrain five paired encoders and fine-tune all weights.
5. Report or visualize the measured result.

CODE

```
# [1]
TensorState = dict[str, torch.Tensor]

# [2]
def pretrain(
    seed: int,
) -> tuple[TensorState, TensorState, torch.Tensor, int]:
    torch.manual_seed(seed)
    encoder = TinyBertEncoder()
    initial = copy.deepcopy(encoder.state_dict())
    mlm = MaskedLanguageModel(encoder)
    optimizer = torch.optim.Adam(mlm.parameters(), lr=1e-3)
    index_generator = torch.Generator().manual_seed(seed + 1)
    mask_generator = torch.Generator().manual_seed(seed + 2)
    schedule = torch.randint(
        len(source_ids),
        (PRETRAIN_STEPS, BATCH_SIZE),
        generator=index_generator,
    )
    branch_counts = torch.zeros(3, dtype=torch.long)
    eligible_count = 0

    for indices in schedule:
        original = source_ids[indices]
        eligible_count += (
            (original != PAD) & (original != CLS) & (original != SEP)
        ).sum().item()
        corrupted, selected, counts = corrupt(
            original, mask_generator
        )
        branch_counts += counts
        logits = mlm(corrupted, selected)
        targets = original[selected]
        assert logits.shape == (targets.numel(), len(vocabulary))
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()
        loss.backward()
```



```

nn.utils.clip_grad_norm_(mlm.parameters(), 1.0)
optimizer.step()

pretrained = copy.deepcopy(mlm.encoder.state_dict())
initial_control = initial["token_embedding.weight"][uncovered_ids]
final_control = pretrained["token_embedding.weight"][uncovered_ids]
assert torch.equal(initial_control, final_control)
return initial, pretrained, branch_counts, eligible_count

def neutral_sentence(word: str) -> torch.Tensor:
    return encode(["we", "found", "the", word, "today"])

class WordClassifier(nn.Module):
    def __init__(
        self, encoder_state: TensorState, head_state: TensorState
    ) -> None:
        super().__init__()
        self.encoder = TinyBertEncoder()
        self.encoder.load_state_dict(encoder_state)
        self.head = nn.Linear(WIDTH, 2)
        self.head.load_state_dict(head_state)

    def forward(self, ids: torch.Tensor) -> torch.Tensor:
        # [CLS] we found the WORD today [SEP] -> WORD is position 4.
        return self.head(self.encoder(ids)[: , 4])

# [3]
def fit_classifier(
    encoder_state: TensorState,
    head_state: TensorState,
    family_zero: list[str],
    family_one: list[str],
) -> tuple[WordClassifier, float]:
    inputs = torch.stack([
        neutral_sentence(word)
        for word in family_zero + family_one
    ])
    targets = torch.tensor(
        [0] * len(family_zero) + [1] * len(family_one)
    )
    model = WordClassifier(encoder_state, head_state)
    optimizer = torch.optim.Adam([
        {"params": model.encoder.parameters(), "lr": 2e-4},
        {"params": model.head.parameters(), "lr": 2e-3},
    ])
    for _ in range(160):
        loss = F.cross_entropy(model(inputs), targets)
        optimizer.zero_grad()
        loss.backward()

```

```

        optimizer.step()
    train_accuracy = (
        (model(inputs).argmax(1) == targets).float().mean().item()
    )
    return model, train_accuracy

@torch.no_grad()
def classifier_accuracy(
    model: WordClassifier,
    family_zero: list[str],
    family_one: list[str],
) -> float:
    inputs = torch.stack([
        neutral_sentence(word)
        for word in family_zero + family_one
    ])
    targets = torch.tensor(
        [0] * len(family_zero) + [1] * len(family_one)
    )
    return (
        (model(inputs).argmax(1) == targets).float().mean().item()
    )

paired_results: dict[int, list[tuple[float, float]]] = {
    count: [] for count in LABEL_COUNTS
}
uncovered_results: dict[int, list[tuple[float, float]]] = {
    count: [] for count in LABEL_COUNTS
}
training_results: dict[int, list[tuple[float, float]]] = {
    count: [] for count in LABEL_COUNTS
}
all_branches = torch.zeros(3, dtype=torch.long)
all_eligible = 0
first_states: tuple[TensorState, TensorState] | None = None

# [4]
for replicate_seed in REPLICATE_SEEDS:
    initial_state, pretrained_state, counts, eligible_count = pretrain(
        replicate_seed
    )
    if first_states is None:
        first_states = (initial_state, pretrained_state)
    all_branches += counts
    all_eligible += eligible_count

    label_generator = torch.Generator().manual_seed(replicate_seed)
    order_zero = torch.randperm(
        len(covered_words[0]), generator=label_generator

```

```

).tolist()
order_one = torch.randperm(
    len(covered_words[1]), generator=label_generator
).tolist()
torch.manual_seed(replicate_seed + 10_000)
head_state = copy.deepcopy(nn.Linear(WIDTH, 2).state_dict())

for count in LABEL_COUNTS:
    label_zero = [covered_words[0][i] for i in order_zero[:count]]
    label_one = [covered_words[1][i] for i in order_one[:count]]
    test_zero = [covered_words[0][i] for i in order_zero[count:]]
    test_one = [covered_words[1][i] for i in order_one[count:]]

    scratch, scratch_train = fit_classifier(
        initial_state, head_state, label_zero, label_one
    )
    pretrained, pretrained_train = fit_classifier(
        pretrained_state, head_state, label_zero, label_one
    )
    scratch_score = classifier_accuracy(
        scratch, test_zero, test_one
    )
    pretrained_score = classifier_accuracy(
        pretrained, test_zero, test_one
    )
    scratch_uncovered = classifier_accuracy(
        scratch, uncovered_words[0], uncovered_words[1]
    )
    pretrained_uncovered = classifier_accuracy(
        pretrained, uncovered_words[0], uncovered_words[1]
    )
    paired_results[count].append(
        (scratch_score, pretrained_score)
    )
    uncovered_results[count].append(
        (scratch_uncovered, pretrained_uncovered)
    )
    training_results[count].append(
        (scratch_train, pretrained_train)
    )

branch_shares = all_branches / all_branches.sum()
# [5]
print(
    "selected targets across five runs: "
    f"{all_branches.sum().item():,}"
)
print(f"eligible positions: {all_eligible:,}")
print(

```

```

    "realized selection rate: "
    f"{all_branches.sum().item() / all_eligible:.3%}"
)
print("realized mask/random/unchanged shares:", branch_shares.tolist())
for count in LABEL_COUNTS:
    pairs = np.array(paired_results[count])
    control_pairs = np.array(uncovered_results[count])
    train_pairs = np.array(training_results[count])
    deltas = pairs[:, 1] - pairs[:, 0]
    print(
        f"{count}/family: scratch={pairs[:, 0].mean():.3f}, "
        f"pretrained={pairs[:, 1].mean():.3f}, "
        f"delta={deltas.mean():+.3f}, wins={(deltas > 0).sum()}/5, "
        f"uncovered={control_pairs[:, 0].mean():.3f}/"
        f"{control_pairs[:, 1].mean():.3f}, "
        f"train={train_pairs[:, 0].mean():.3f}/"
        f"{train_pairs[:, 1].mean():.3f}"
    )

```

```

selected targets across five runs: 155,019
eligible positions: 1,032,096
realized selection rate: 15.020%
realized mask/random/unchanged shares: [0.8003599643707275, 0.09977486729621887,
↪ 0.09986517578363419]
1/family: scratch=0.468, pretrained=0.995, delta=+0.526, wins=5/5,
↪ uncovered=0.475/0.425, train=1.000/1.000
2/family: scratch=0.489, pretrained=1.000, delta=+0.511, wins=5/5,
↪ uncovered=0.500/0.440, train=1.000/1.000
4/family: scratch=0.494, pretrained=1.000, delta=+0.506, wins=5/5,
↪ uncovered=0.550/0.430, train=1.000/1.000

```

Across 1,032,096 eligible positions, the five runs select 155,019—15.020%. The realized branches are 80.04% mask, 9.98% random, and 9.99% unchanged. (Estimator reading, per [Chapter 4](#): the MLM loss is Case 1 over exactly the *selected* positions — the corruption policy defines the scored population, the batch mean estimates it without bias, and these counts are that denominator made visible.) Both scratch and pretrained models reach 100% on their tiny labeled training sets in every run; failure to fit the labels therefore does not explain the scratch result.

15.6.4. What transferred?

Remember the question: did pretraining organize covered representations in a way that scarce labels could reuse—or did the classifier merely learn the tiny label set? The task, control, and geometry answer different parts.

PLAN

1. Define the reusable `representation_geometry` helper.
2. Audit paired transfer, source coverage, and representation geometry.

CODE

```
# [1]
@torch.no_grad()
def representation_geometry(
    encoder_state: TensorState,
) -> tuple[float, float]:
    model = TinyBertEncoder()
    model.load_state_dict(encoder_state)
    words = covered_words[0] + covered_words[1]
    vectors = model(torch.stack([
        neutral_sentence(word) for word in words
    ]))[:, 4]
    vectors = vectors - vectors.mean(dim=0, keepdim=True)
    vectors = F.normalize(vectors, dim=1)
    similarities = vectors @ vectors.T
    labels = torch.tensor(
        [0] * len(covered_words[0]) + [1] * len(covered_words[1])
    )
    off_diagonal = ~torch.eye(len(words), dtype=torch.bool)
    within = similarities[
        (labels[:, None] == labels[None, :]) & off_diagonal
    ]
    between = similarities[labels[:, None] != labels[None, :]]
    return within.mean().item(), between.mean().item()

assert first_states is not None
initial_geometry = representation_geometry(first_states[0])
pretrained_geometry = representation_geometry(first_states[1])

budgets = np.array(LABEL_COUNTS)
scratch_means = np.array([
    np.mean(paired_results[count], axis=0)[0]
    for count in LABEL_COUNTS
])
pretrained_means = np.array([
    np.mean(paired_results[count], axis=0)[1]
    for count in LABEL_COUNTS
])
scratch_uncovered_means = np.array([
    np.mean(uncovered_results[count], axis=0)[0]
    for count in LABEL_COUNTS
])
```

```

])
pretrained_uncovered_means = np.array([
    np.mean(uncovered_results[count], axis=0)[1]
    for count in LABEL_COUNTS
])

# [2]
print(f"initial within/between: {initial_geometry}")
print(f"after MLM within/between: {pretrained_geometry}")

```

initial within/between: (-0.03466804325580597, -0.0160280279815197)
after MLM within/between: (0.5322557687759399, -0.5517237186431885)

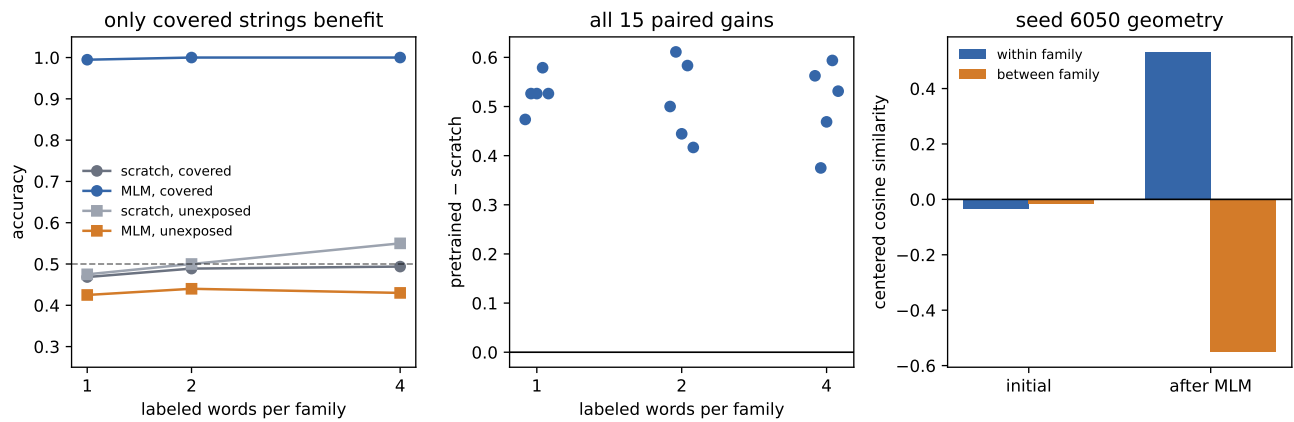


Figure 15.5.: Five paired end-to-end seeds in the synthetic transfer lab. MLM initialization separates covered latent families and improves every covered-type comparison (left and center); both arms stay near chance on input-vocabulary controls withheld from MLM inputs, replacements, and targets (left). Centered cosine similarity is a representation diagnostic, not a task metric (right).

Labels / family	Scratch, covered	MLM, covered	Gain	Scratch, unexposed	MLM, unexposed
1	0.468	0.995	+0.526	0.475	0.425
2	0.489	1.000	+0.511	0.500	0.440
4	0.494	1.000	+0.506	0.550	0.430

MLM initialization wins all 15 paired covered-type comparisons. At two labels per family, for example, mean covered accuracy moves from 0.489 to 1.000. On the 40 unexposed controls, scratch and MLM-initialized means stay around chance at every budget. MLM therefore helps types whose source contexts contain the latent regularity—not arbitrary rows that merely exist in the input vocabulary.

The labeled sets are nested within a seed, but increasing the budget removes those additional types from that budget’s test pool. The within-budget scratch-versus-MLM comparison is exact; movement across budgets is not a controlled learning curve.

The representation diagnostic agrees with the task result. In seed 6050, mean centered cosine similarity starts at -0.0347 within a family and -0.0160 between families—no useful separation. After MLM, it

is 0.5323 within and -0.5517 between. *Centered cosine* first subtracts the mean representation, then unit-normalizes each vector and takes a dot product: positive values point in similar residual directions, negative values in opposing directions. Pretraining has organized neutral-context token states around the latent source regularity, and a few labels recover that organization for unseen covered types.

This closes Chapter 9’s contract at toy scale:

- **Labels are scarce:** the task supplies only 2, 4, or 8 labeled words total.
- **The target is representation-hungry:** spelling, frequency, and task-context cues are neutralized; the label must transfer through the token representation.
- **Source coverage is useful at a matched scale:** source and target use atomic tokens and similarly short sequences; covered strings transfer, while balanced vocabulary residents withheld from MLM inputs, replacements, and targets remain near chance in both arms.

The result is an existence proof for a mechanism. A co-occurrence method could also recover the synthetic families; the experiment does not establish Transformer superiority. The limits are concrete—the data generator builds the regularity, the pretrained arm receives extra compute, and five toy seeds do not estimate natural-language performance. The justified conclusion is narrower: **in this controlled construction, MLM organizes covered token representations around a latent regularity, and a few labels recover it for unseen covered types.**

15.7. Three pretraining families

Pretraining is a regime, not an architecture. The visibility rule and pretraining objective determine what a model can learn—and how it will later be used.

Table 15.1.: Three pretraining families separate visibility, objective, and characteristic use.

Family	Visibility during pretraining	Objective	Characteristic use
BERT-style encoder	Full nonpadding context	Predict selected original tokens after 80/10/10 treatment	Contextual representations and task heads
GPT-style decoder	Causal prefix	Predict the next token	Autoregressive generation and adaptation
T5-style encoder–decoder	Full encoder; causal decoder with cross-attention	Generate removed spans separated by sentinel tokens	Text-to-text conditional generation

The visibility column becomes concrete in Figure 15.6. Each matrix uses queries as rows and keys as columns, exactly as in Figure 15.1.

The separation in Table 15.1 matters during execution too. A GPT-style decoder can train all next-token positions in parallel under teacher forcing because the causal mask exposes only each prefix. Generation is still sequential: the model must emit a token before that token can join the next prefix. Training and inference share a factorization and visibility rule, not an execution schedule.

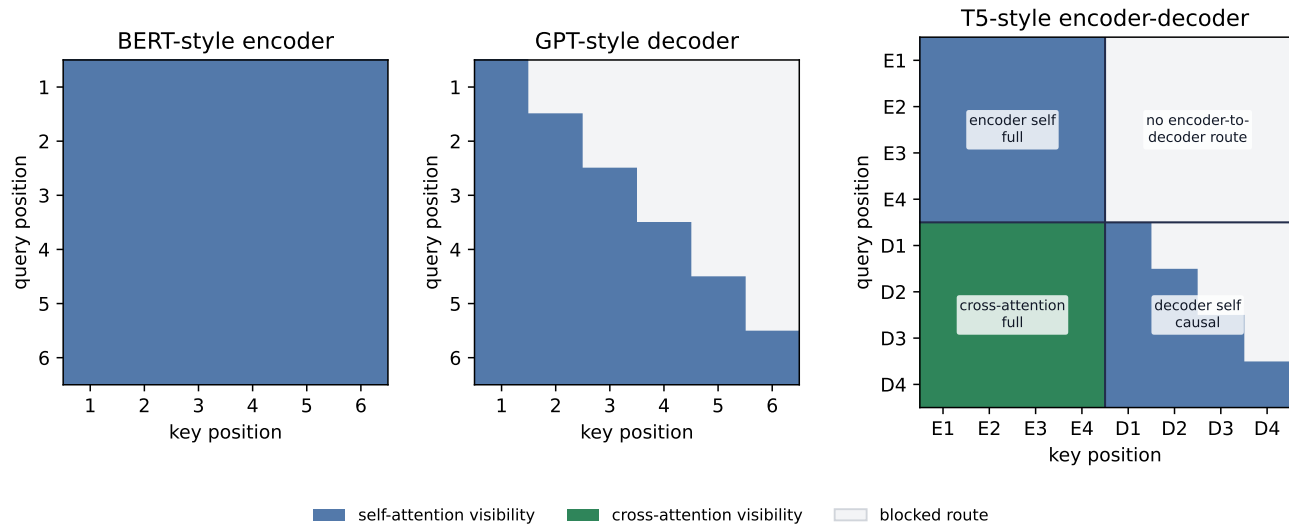


Figure 15.6.: Three pretraining families differ first in what each position may retrieve. A BERT-style encoder has full nonpadding self-attention; a GPT-style decoder has causal self-attention. T5 uses three routes in two stacks: full encoder self-attention, causal decoder self-attention, and decoder-to-encoder cross-attention. The combined T5 block is a visibility ledger for those separate sublayers, not one mask applied by the implementation.

Once those next-token logits exist, Chapter 11’s greedy and beam-search machinery can rank continuations—the architecture changes, but decoding remains a separate decision layer. Chapter 17 will return to how prompts alter the context supplied to that decoder.

T5 turns tasks into text-to-text problems and marks removed spans with learned sentinel tokens. Its encoder produces a *sequence* of memory states under full visibility; its causal decoder cross-attends to that sequence while generating the target. It does not squeeze the input into one fixed bottleneck vector. These families trade representation, generation, and input–output structure differently—they are not a ranking from old to new.

15.8. What changed after 2018

Before the pretrained era, a new labeled dataset often implied a new model trained from a random initialization. After BERT and its contemporaries, the default question changed: which pretrained model, which adaptation, and which data interface fit the task?

Three consequences follow.

Optimization starts elsewhere. The downstream run begins in a parameter region already shaped by a broad source objective. The task is not asking a few labels to invent syntax and semantics from scratch.

Data work moves upstream. Corpus choice, tokenization, corruption, and source coverage become part of model design. Pretraining is curriculum at dataset scale.

One backbone serves many interfaces. A sequence head, token head, or span head can reuse the same encoder. That modularity makes comparison easier—but also makes pretraining assumptions easy to inherit without inspecting them.

The regime introduces new failure modes. Source bias can transfer with useful features. A vocabulary can represent a token without supplying useful exposure. Full fine-tuning can forget source behavior or become too expensive as the backbone grows. Those are not footnotes to transfer; they are the agenda of the pretrained era.

The chapter closes with three forward seeds. A learned summary token can gather a sequence for a downstream head; [Chapter 16](#) will give that same pattern a sequence of image patches. Pretraining is a regime, not an architecture; [Chapter 16](#) will ask what changes when an encoder leaves text for images and scaling takes center stage. Fine-tuning changed every encoder weight here; [Chapter 17](#) will ask what adaptation remains when most, or all, of those weights must stay fixed.

i Check yourself

Close the book for one minute and reconstruct masked-language-model pretraining.

- How are visibility, supervision, corruption, and padding represented separately?
- Why is the 80/10/10 rule conditional on selection?
- What changes between a frozen probe and full fine-tuning?

15.9. Okay, so — pretraining manufactures supervision

1. **Visibility, selection, corruption, and padding are separate.** BERT removes the causal triangle, selects loss targets, then applies the 80/10/10 treatment so copying cannot solve most of them. [MASK] is a token—not an attention ban.
2. **MLM turns raw text into supervision.** Select 15% of eligible positions; conditional on selection, use 80% mask, 10% random, and 10% unchanged. All selected positions predict their original tokens.
3. **BERT is a full-context Transformer encoder.** WordPiece tokens, learned token/position/segment embeddings, post-LayerNorm blocks, MLM, and historical NSP define the original recipe.
4. **Fine-tuning updates the encoder.** Training only a new head over frozen features is a probe. Standard BERT adaptation moves the pretrained backbone and task head together.
5. **The controlled lab closes the transfer loop at toy scale.** Across five full seeds, MLM initialization wins every covered-type comparison under 2–8 total labels; input-vocabulary controls withheld from MLM inputs, replacements, and targets remain near chance in both arms.
6. **Pretraining changed the unit of work.** The field moved from one random model per task toward reusable backbones and specialized adaptation. Corpus, objective, visibility, and adaptation now belong to one design decision.

Sources and further reading

- [Peters et al., *Deep Contextualized Word Representations*](#): ELMo and fixed contextual features from forward and backward language models.
- [Howard and Ruder, *Universal Language Model Fine-tuning for Text Classification*](#): ULMFiT’s full-model adaptation recipe.
- [Radford et al., *Improving Language Understanding by Generative Pre-Training*](#): causal Transformer pretraining followed by supervised adaptation.

- Sennrich, Haddow, and Birch, *Neural Machine Translation of Rare Words with Subword Units*: the frequency-driven BPE adaptation for subword vocabularies.
- Devlin et al., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*: the original BERT architecture, MLM/NSP objectives, and fine-tuning results.
- Google Research, *BERT pretraining implementation*: the released implementation documents the tied input/output embedding weights.
- Liu et al., *RoBERTa: A Robustly Optimized BERT Pretraining Approach*: a later study of BERT training choices, including removal of NSP.
- Raffel et al., *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*: the T5 text-to-text framework and span-corruption recipe.
- Salazar et al., *Masked Language Model Scoring* (2020): token-by-token masking as a pseudo-log-likelihood sentence score, with the distinction between a score and a normalized left-to-right likelihood.

Exercises

1. **(Pencil.)** For [CLS] the wug is calm [SEP] [PAD] [PAD], suppose *wug* is selected and replaced by [MASK]. Draw Chapter 14’s causal visibility matrix and BERT’s full nonpadding visibility matrix, using queries as rows and keys as columns. Separately mark the one position that contributes MLM loss. Why may neighboring tokens attend to [MASK], and why would full visibility leak information in a next-token generator?
2. **(Pencil.)** A batch contains 2,000 eligible ordinary tokens. Under BERT’s policy, how many positions are expected to be selected, replaced by [MASK], replaced by a random token, and left unchanged? How many contribute direct MLM loss? Explain why 12% is not the supervision rate.
3. **(Code.)** For token IDs (B, T) , width d , and vocabulary size $|V|$, add assertions for hidden states (B, T, d) , dense logits $(B, T, |V|)$, selected logits $(|M|, |V|)$, and selected targets $(|M|,)$. Assert that [PAD], [CLS], and [SEP] are never selected. Show numerically that selected-only cross-entropy matches dense cross-entropy when unselected targets use `ignore_index`.
4. **(Code.)** Starting from the same pretrained checkpoint and split, compare a frozen encoder probe with full fine-tuning in the generated-token lab. Match head tensors, labeled types, updates, and seeds. Plot all five paired results rather than only their means. Which regime wins here, and what broader claim would outrun the experiment?
5. **(Audit.)** Reproduce paired gains for covered and control strings at all three budgets. Map the evidence to Chapter 9’s gates. Explain the unchanged-row assertion, name two unsupported conclusions, and propose one predeclared coverage test.
6. **(Pencil.)** Define the pseudo-log-likelihood $PLL(x) = \sum_{i=1}^T \log p(x_i | x_{\setminus i})$ and explain why ordinary masked scoring costs T forward passes. Show why the collection of conditionals need not define one normalized joint distribution. **(Code.)** Score minimal sentence pairs from the controlled lab by masking each ordinary token once. Train a small causal model on that same corpus and vocabulary with Chapter 10’s trainer, then compare rankings under one declared length convention. Assert that special and padding tokens contribute neither PLL score nor mask position. Report PLL as a *score*, not a normalized sentence likelihood.

16. Vision Transformers and Scaling Laws

Chapter 14 promised that **global routing trades away locality bias**. Chapter 15 added two pieces: a learned summary token can gather a sequence for a downstream head, and **pretraining is a regime, not an architecture**. Let us collect those promises in one image. If a Transformer can read a sentence of words, what must change before it can read a photograph?

Surprisingly little. Cut the image into patches, project each patch to a vector, and let the Transformer encoder read those vectors as tokens. That minimalist transplant is the **Vision Transformer (ViT)**. Its simplicity—and the freedom it creates—is also the source of the interesting question. A convolutional network arrives knowing that nearby pixels belong together and that one local detector should slide across the image. ViT arrives with a much weaker image-specific prescription. Which learner wins therefore depends not just on architecture, but on data, training, and scale.

That word—*scale*—will change jobs halfway through the chapter. First it will mean making an image model deeper, wider, or higher-resolution. Then it will mean fitting empirical laws that connect loss to parameters, tokens, and training compute. The bridge between the two meanings is resource allocation: what should we enlarge, and what must grow with it?

16.1. Let the image become a sequence

Start with the arithmetic used by the original ViT. A color image is 224×224 pixels, and each square patch is 16×16 . There are

$$\frac{224}{16} = 14 \quad \Rightarrow \quad 14^2 = 196$$

patches. Each patch contains $16 \times 16 \times 3 = 768$ raw channel values. So one image has become a sequence of 196 vectors, each initially 768-dimensional—already the same width used by the base Transformer in that paper.

In general, let a batch be

$$\mathbf{X} \in \mathbb{R}^{B \times C \times H \times W},$$

where B is batch size, C channels, and H, W are spatial dimensions. For a square patch width P that divides both H and W , the patch count is

$$N = \frac{H}{P} \frac{W}{P}, \tag{16.1}$$

and flattening each patch gives

$$\mathbf{X}_{\text{patch}} \in \mathbb{R}^{B \times N \times (CP^2)}.$$

We can see this on the fixed Fashion-MNIST subset first opened in [Chapter 6](#). The 28×28 grayscale images divide into $7 \times 7 = 49$ patches when $P = 4$.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Load imports, the fixed Chapter 6 split, and image shapes.
-

CODE

```
import hashlib
import math

import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F

# [1]
torch.set_num_threads(6)
torch.manual_seed(6050)

development = torch.load("../data/fashion-train.pt")
holdout = torch.load("../data/fashion-test.pt")
X_dev = development["X"].float().unsqueeze(1) / 255.0 # (1200, 1, 28, 28)
y_dev, classes = development["y"], development["classes"]
X_test = holdout["X"].float().unsqueeze(1) / 255.0 # (600, 1, 28, 28)
y_test = holdout["y"]

split = torch.randperm(len(X_dev), generator=torch.Generator().manual_seed(6050))
fit_idx, val_idx = split[:1000], split[1000:]
X_fit, y_fit = X_dev[fit_idx], y_dev[fit_idx] # (1000, 1, 28, 28)
X_val, y_val = X_dev[val_idx], y_dev[val_idx] # (200, 1, 28, 28)

# [2]
print("fit / validation / benchmark:", len(X_fit), len(X_val), len(X_test))
```

```
fit / validation / benchmark: 1000 200 600
```

A learned matrix

$$\mathbf{E} \in \mathbb{R}^{CP^2 \times d}$$

projects every raw patch to model width d . The operation has a useful exact identity. Unfold the image, multiply each flattened patch by $\mathbf{E}$, and add a bias—or reshape the columns of $\mathbf{E}$ into d convolutional kernels with kernel size P and stride P . The two computations are the same linear map.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Patchify and verify unfold-plus-linear equals one strided convolution.
-

CODE

```
# [1]
P, d = 4, 32
sample = X_fit[:1]
generator = torch.Generator().manual_seed(6050)
E_patch = torch.randn(P * P, d, generator=generator)
b_patch = torch.randn(d, generator=generator)

raw_patches = F.unfold(sample, kernel_size=P, stride=P).transpose(1, 2)
tokens_linear = raw_patches @ E_patch + b_patch
conv_kernel = E_patch.T.reshape(d, 1, P, P)
tokens_conv = F.conv2d(
    sample, conv_kernel, bias=b_patch, stride=P
).flatten(2).transpose(1, 2)
max_error = (tokens_linear - tokens_conv).abs().max().item()

assert raw_patches.shape == (1, 49, 16)
assert tokens_linear.shape == (1, 49, 32)
assert max_error < 2e-6
# [2]
print("raw patches:", tuple(raw_patches.shape))
print("projected tokens:", tuple(tokens_linear.shape))
print(f"max |linear - convolution|: {max_error:.2e}")
```

```
raw patches: (1, 49, 16)
projected tokens: (1, 49, 32)
max |linear - convolution|: 7.15e-07
```

This identity is more than a coding shortcut. The patch projection is a learned, shared local operator—a thin residue of image structure at ViT’s front door. What changes is what happens next: instead of stacking local feature maps, we hand the patch sequence to global self-attention.

 Patchify once, audit twice

Check that P divides both spatial dimensions, assert the patch and token shapes, and compare unfold-plus-linear with the equivalent convolution to numerical roundoff. Patchification silently changes sequence length, feature width, and the attention bill; one shape mistake propagates

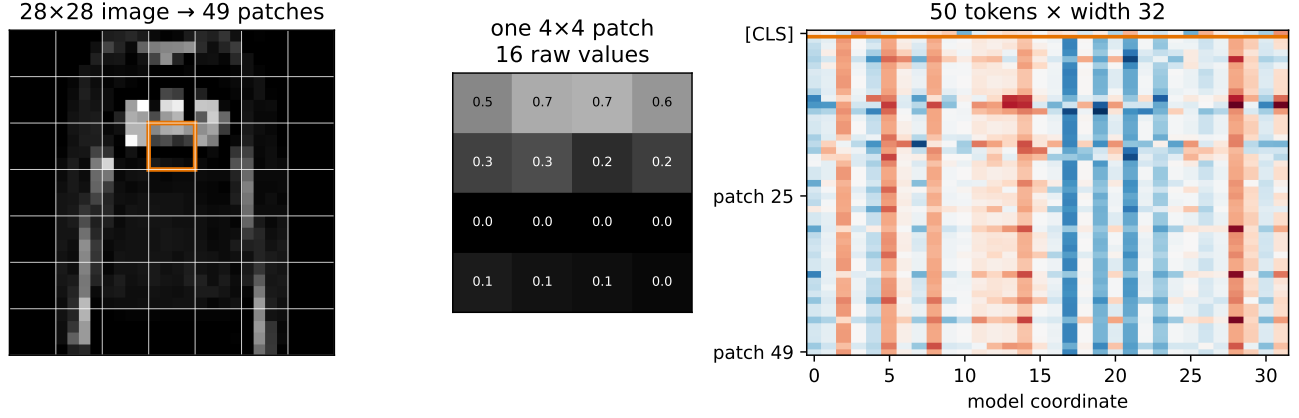


Figure 16.1.: An image becomes a sequence. A 28×28 garment is divided into 49 nonoverlapping 4×4 patches (left); one patch contains 16 raw values (center); a shared linear projection maps every patch to width 32, and a separate learned [CLS] row is prepended (right). The displayed projection is random arithmetic, not a trained representation.

through the entire model.

16.2. A learned meeting place for patches

The projected patches still need two additions. First, prepend a learned vector $\mathbf{x}_{\text{CLS}} \in \mathbb{R}^d$. Second, add one learned position vector to every sequence slot. For image b , ViT’s complete encoder input is

$$\mathbf{Z}_0^{(b)} = [\mathbf{x}_{\text{CLS}}; \mathbf{x}_p^1 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times d}, \quad (16.2)$$

where $\mathbf{x}_p^i \in \mathbb{R}^{CP^2}$ is flattened patch i , $\mathbf{E} \in \mathbb{R}^{CP^2 \times d}$ is the shared patch projection, and $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times d}$ contains learned absolute position vectors.

[CLS] is not a summary by birth. As in [Chapter 15](#), it is a **learned meeting place** whose downstream loss teaches it what to gather. Every encoder layer lets that row query the patch rows; after the final layer, a linear head reads its d coordinates and produces class logits. The original ViT appendix found that class-token and global-average-pooling heads worked similarly after tuning their learning rates. [CLS] is a useful interface—not proof of a superior pooling rule.

Position is essential for a different reason. Without positional information, self-attention is **permutation-equivariant**: permuting patch rows produces the same permutation of output rows. A symmetric pooling rule can consequently become invariant to patch order, unable to distinguish a sleeve above a shoe from a shoe above a sleeve. Learned positions break that symmetry. In the original ablation, removing position hurt, while learned 1-D, learned 2-D, and relative alternatives performed similarly in that setting. The result supports *supplying geometry*; it does not establish one universal positional scheme.

ViT then reuses Chapter 14’s pre-LayerNorm encoder block. With $\mathbf{Z}_{\ell-1} \in \mathbb{R}^{B \times (N+1) \times d}$,

$$\tilde{\mathbf{Z}}_\ell = \mathbf{Z}_{\ell-1} + \text{MSA}(\text{LN}(\mathbf{Z}_{\ell-1})),$$

$$\mathbf{Z}_\ell = \tilde{\mathbf{Z}}_\ell + \text{MLP}\left(\text{LN}(\tilde{\mathbf{Z}}_\ell)\right).$$

The residual stream, LayerNorm axis, multi-head routing, and position-wise MLP are unchanged. This is an encoder-only classifier, so there is no causal mask and no decoder. Pre-LayerNorm gives gradients a cleaner residual path; it does not make optimization automatic. The original ViT training recipe still used learning-rate warmup.

16.3. Inductive bias strikes again

Chapter 6 called constraints a form of knowledge. **Inductive bias strikes again, this time as a trade.** Convolution spends flexibility to buy locality and translation equivariance; ViT gives back much of that image-specific prior and must purchase useful geometry with data, optimization, or a new architectural constraint.

Convolutional network	Vision Transformer
Local, spatially shared kernels	Content-dependent routes between all patch tokens
Translation equivariance built into feature maps	Geometry supplied mainly by patches and position vectors
Global sight accumulated through depth	Every patch reachable in one attention layer
Strong image-specific prior	More freedom, and greater dependence on the training regime

“Less image-specific bias”—not “no bias”—is the precise phrase. Patchification is local, the same projection is shared over the grid, positions identify slots, and the usual optimizer and architecture choices contribute biases of their own. ViT did not remove assumptions. It changed which assumptions were paid for in code and which had to be learned from examples.

That freedom has a computational price. Ignoring [CLS] for the cleanest count, changing a 224×224 image from $P = 32$ to $P = 16$ changes the patch count from 49 to 196: four times as many tokens. Each attention head’s score matrix therefore grows from $49^2 = 2,401$ to $196^2 = 38,416$ entries—sixteen times as many. At 1024×1024 with $P = 16$, there are 4,096 patches—and the score matrix has about 437 times as many entries as the 224×224 case.

Do not shorten this to “the whole Transformer is quadratic.” Attention mixing costs $O(N^2d)$, while the projections and feedforward layers contribute terms like $O(Nd^2)$. Which term dominates depends on both token count and width. The quadratic term becomes especially painful when resolution rises or patches shrink.

16.4. A Fashion rematch, not a referendum

The original ViT paper warns us that regime matters. Before looking at its large pretraining experiments, let us deliberately choose the opposite regime: 1,000 Fashion-MNIST fitting images, no pretraining, and models small enough to inspect. Predict the ordering before the run — which architecture wins clean validation at this scale, and which degrades less under shift?

The ViT uses 4×4 patches, width 32, four heads, two pre-LayerNorm blocks, and feedforward width 64. Its 49 patch tokens plus [CLS] make a 50-token sequence. The convolutional baseline uses four 3×3 layers, two max pools, and global average pooling. Their parameter counts differ by about 3%. Both use the same initialization family, optimizer, update count, seed, and explicit minibatch order.

The implementations below are derived independently from the chapter equations. There is no imported ViT package hiding extra training machinery.

PLAN

1. Define the reusable helpers: `MultiHeadSelfAttention`, `EncoderBlock`, and `TinyViT`.
 2. Define the reusable helpers: `TinyCNN`, `_xavier_init`, and `count_parameters`.
 3. Independently define the tiny ViT and parameter-matched CNN.
-

CODE

```
# [1]
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, width: int, heads: int) -> None:
        super().__init__()
        assert width % heads == 0
        self.heads = heads
        self.head_width = width // heads
        self.qkv = nn.Linear(width, 3 * width)
        self.proj = nn.Linear(width, width)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        batch, tokens, width = x.shape
        qkv = self.qkv(x).reshape(
            batch, tokens, 3, self.heads, self.head_width
        ).permute(2, 0, 3, 1, 4)
        q, k, v = qkv.unbind(0)
        scores = (q @ k.transpose(-2, -1)) / math.sqrt(self.head_width)
        mixed = scores.softmax(dim=-1) @ v
        mixed = mixed.transpose(1, 2).reshape(batch, tokens, width)
        return self.proj(mixed)

class EncoderBlock(nn.Module):
    def __init__(self, width: int, heads: int, ff_width: int) -> None:
        super().__init__()
        self.norm1 = nn.LayerNorm(width)
        self.attn = MultiHeadSelfAttention(width, heads)
        self.norm2 = nn.LayerNorm(width)
        self.ffn = nn.Sequential(
            nn.Linear(width, ff_width), nn.GELU(), nn.Linear(ff_width, width)
        )
```



```

def forward(self, x: torch.Tensor) -> torch.Tensor:
    x = x + self.attn(self.norm1(x))
    return x + self.ffn(self.norm2(x))

class TinyViT(nn.Module):
    def __init__(self, width: int = 32, patch: int = 4, blocks: int = 2) -> None:
        super().__init__()
        self.patch_embed = nn.Conv2d(1, width, kernel_size=patch, stride=patch)
        patch_tokens = (28 // patch) ** 2
        self.cls_token = nn.Parameter(torch.zeros(1, 1, width))
        self.position = nn.Parameter(torch.zeros(1, patch_tokens + 1, width))
        self.blocks = nn.Sequential(*(
            EncoderBlock(width, heads=4, ff_width=2 * width)
            for _ in range(blocks)
        ))
        self.norm = nn.LayerNorm(width)
        self.head = nn.Linear(width, 10)
        self.apply(_xavier_init)
        nn.init.normal_(self.cls_token, std=0.02)
        nn.init.normal_(self.position, std=0.02)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        patches = self.patch_embed(x).flatten(2).transpose(1, 2)
        cls = self.cls_token.expand(len(x), -1, -1)
        tokens = torch.cat((cls, patches), dim=1) + self.position
        encoded = self.blocks(tokens)
        return self.head(self.norm(encoded[:, 0]))

# [2]
class TinyCNN(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 8, 3, padding=1), nn.GELU(), nn.MaxPool2d(2),
            nn.Conv2d(8, 16, 3, padding=1), nn.GELU(), nn.MaxPool2d(2),
            nn.Conv2d(16, 32, 3, padding=1), nn.GELU(),
            nn.Conv2d(32, 48, 3, padding=1), nn.GELU(),
        )
        self.head = nn.Sequential(
            nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(48, 10)
        )
        self.apply(_xavier_init)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.features(x))

```

```
def _xavier_init(module: nn.Module) -> None:
    if isinstance(module, (nn.Linear, nn.Conv2d)):
        nn.init.xavier_uniform_(module.weight)
        if module.bias is not None:
            nn.init.zeros_(module.bias)

def count_parameters(model: nn.Module) -> int:
    return sum(parameter.numel() for parameter in model.parameters())

# [3]
print("CNN parameters:", f"{count_parameters(TinyCNN()):,}")
print("ViT parameters:", f"{count_parameters(TinyViT()):,}")
```

```
CNN parameters: 20,250
ViT parameters: 19,658
```

For one 28×28 image, a simple multiply-accumulate ledger gives about 1.19 million for the CNN and 1.16 million for the ViT, a 1.8% difference. That ledger counts convolution, linear, and attention matrix products; it omits normalization, activations, softmax, pooling, memory traffic, and implementation effects. It is a useful dot-product proxy—not measured FLOPs, speed, or energy. Appendix C supplies the missing byte ledger and Roofline model, including why neither an operation count nor a dtype alone predicts elapsed time.

The training protocol uses AdamW for 120 epochs, a cosine learning-rate schedule, and batches of 100. For each seed, one explicit list of permutations is handed to both architectures. Resetting the seed before each model makes initialization reproducible; sharing the permutation list makes minibatch order exactly equal. The schedule hash makes that equality inspectable rather than aspirational.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `minibatch_schedule`, `schedule_hash`, and `train_with_schedule`.
 3. Define the reusable helpers: `accuracy`, `shift_right`, and `run_paired_seed`.
 4. Define the explicit paired schedule, training loop, and shift audit.
-

CODE

```
# [1]
EPOCHS = 120
SEEDS = list(range(6050, 6055))

# [2]
def minibatch_schedule(seed: int) -> list[torch.Tensor]:
    generator = torch.Generator().manual_seed(seed + 10_000)
```

```

return [torch.randperm(len(X_fit), generator=generator) for _ in range(EPOCHS)]

def schedule_hash(permutations: list[torch.Tensor]) -> str:
    raw = torch.cat(permutations).numpy().tobytes()
    return hashlib.sha256(raw).hexdigest()[:12]

def train_with_schedule(
    model_fn, permutations: list[torch.Tensor], seed: int
) -> nn.Module:
    torch.manual_seed(seed)
    model = model_fn()
    optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=1e-2)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=EPOCHS, eta_min=3e-5
    )
    model.train()
    for permutation in permutations:
        for start in range(0, len(X_fit), 100):
            idx = permutation[start:start + 100]
            loss = F.cross_entropy(model(X_fit[idx]), y_fit[idx])
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            scheduler.step()
    return model

# [3]
@torch.no_grad()
def accuracy(model: nn.Module, x: torch.Tensor, y: torch.Tensor) -> float:
    model.eval()
    return (model(x).argmax(1) == y).float().mean().item()

def shift_right(x: torch.Tensor, pixels: int) -> torch.Tensor:
    if pixels == 0:
        return x.clone()
    shifted = torch.zeros_like(x)
    shifted[..., pixels:] = x[..., :-pixels]
    return shifted

def run_paired_seed(seed: int) -> list[dict]:
    permutations = minibatch_schedule(seed)
    rows = []
    print(f"seed {seed}, schedule {schedule_hash(permutations)}")
    for name, model_fn in (("CNN", TinyCNN), ("ViT", TinyViT)):

```

```

model = train_with_schedule(model_fn, permutations, seed)
row = {
    "seed": seed,
    "model": name,
    "train": accuracy(model, X_fit, y_fit),
    "validation": [
        accuracy(model, shift_right(X_val, px), y_val) for px in range(5)
    ],
    "benchmark": accuracy(model, X_test, y_test),
}
rows.append(row)
print(
    f" {name}: train {row['train']:.3f}, "
    f"validation {[round(value, 3) for value in row['validation']]}, "
    f"benchmark {row['benchmark']:.3f}"
)
return rows

vit_macs = (
    7 * 7 * 32 * 16
    + 2 * (
        50 * 32 * (3 * 32) + 2 * 4 * 50 * 50 * 8
        + 50 * 32 * 32 + 2 * 50 * 32 * 64
    )
    + 32 * 10
)
cnn_macs = (
    28 * 28 * 8 * 1 * 3 * 3
    + 14 * 14 * 16 * 8 * 3 * 3
    + 7 * 7 * 32 * 16 * 3 * 3
    + 7 * 7 * 48 * 32 * 3 * 3
    + 48 * 10
)
# [4]
print(f"dot-product/MAC proxy - CNN: {cnn_macs:,}; ViT: {vit_macs:,}")

```

dot-product/MAC proxy - CNN: 1,185,888; ViT: 1,164,608

The five paired runs use one cell per seed so that each training cell stays within the chapter's execution budget. The split is only operational—all five rows enter one analysis.

PLAN

1. Paired Fashion run for seed 6050.
-

CODE

```
# [1]
fashion_results = []
fashion_results.extend(run_paired_seed(6050))
```

```
seed 6050, schedule c42c1c2b4baf
  CNN: train 0.886, validation [0.73, 0.69, 0.68, 0.685, 0.56], benchmark 0.780
  ViT: train 1.000, validation [0.69, 0.61, 0.48, 0.37, 0.225], benchmark 0.720
```

PLAN

1. Paired Fashion run for seed 6051.
-

CODE

```
# [1]
fashion_results.extend(run_paired_seed(6051))
```

```
seed 6051, schedule f0c73c85e0e3
  CNN: train 0.854, validation [0.755, 0.7, 0.695, 0.705, 0.61], benchmark 0.768
  ViT: train 1.000, validation [0.705, 0.575, 0.445, 0.29, 0.21], benchmark 0.735
```

PLAN

1. Paired Fashion run for seed 6052.
-

CODE

```
# [1]
fashion_results.extend(run_paired_seed(6052))
```

```
seed 6052, schedule 20ec620b3e2e
  CNN: train 0.877, validation [0.745, 0.715, 0.685, 0.695, 0.665], benchmark 0.780
  ViT: train 1.000, validation [0.72, 0.635, 0.455, 0.33, 0.23], benchmark 0.743
```

PLAN

1. Paired Fashion run for seed 6053.
-

CODE

```
# [1]
fashion_results.extend(run_paired_seed(6053))

seed 6053, schedule c93bd2a03038
  CNN: train 0.883, validation [0.745, 0.675, 0.625, 0.56, 0.48], benchmark 0.790
  ViT: train 1.000, validation [0.69, 0.615, 0.41, 0.395, 0.23], benchmark 0.723
```

PLAN

1. Paired Fashion run for seed 6054.
-

CODE

```
# [1]
fashion_results.extend(run_paired_seed(6054))

seed 6054, schedule 87a781e7efa8
  CNN: train 0.894, validation [0.72, 0.715, 0.69, 0.64, 0.63], benchmark 0.778
  ViT: train 1.000, validation [0.7, 0.585, 0.495, 0.395, 0.365], benchmark 0.745
```

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Report the paired clean and shifted endpoints.
-

CODE

```
# [1]
by_model = {
  name: [row for row in fashion_results if row["model"] == name]
  for name in ("CNN", "ViT")
}

for name in by_model:
  assert [row["seed"] for row in by_model[name]] == SEEDS
```

```

colors = {"CNN": "#232D4B", "ViT": "#E57200"}
# [2]
for name in ("CNN", "ViT"):
    rows = by_model[name]
    mean_train = np.mean([row["train"] for row in rows])
    mean_val = np.mean([row["validation"][0] for row in rows])
    mean_test = np.mean([row["benchmark"] for row in rows])
    mean_shift4 = np.mean([row["validation"][4] for row in rows])
    print(
        f"{name}: train {mean_train:.1%}; clean validation {mean_val:.1%}; "
        f"benchmark {mean_test:.1%}; validation at 4px {mean_shift4:.1%}"
    )

```

CNN: train 87.9%; clean validation 73.9%; benchmark 77.9%; validation at 4px 58.9%
 ViT: train 100.0%; clean validation 70.1%; benchmark 73.3%; validation at 4px 25.2%

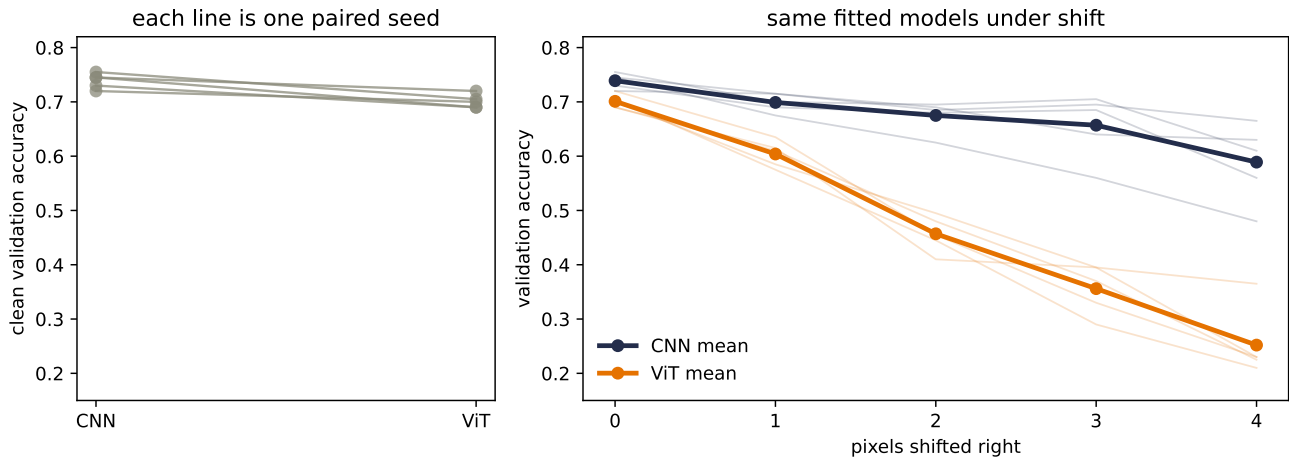


Figure 16.2.: Five parameter- and schedule-matched runs on the fixed Chapter 6 split. Every line at left is one paired clean-validation comparison. At right, thin lines are individual seeds and thick lines are means under zero-padded right shifts; the operation also clips pixels at the right boundary. The CNN’s flatter curve is consistent with its local sharing and pooling, not a causal isolation of any one component.

The result is sharper than a one-number leaderboard—it separates fitting from generalization. The ViT interpolates the fitting set in every run—100.0% mean training accuracy—but averages 70.1% on clean validation images. The CNN fits only 87.9% of the training labels under the shared recipe yet averages 73.9% on validation, winning all five paired comparisons. On the already-opened 600-image benchmark, the same means are 73.3% and 77.9%.

Under a four-pixel shift, the validation means separate much more: 25.2% for ViT and 58.9% for CNN. The zero-filled shift also clips garment content—a reminder that the right edge is lost—so neither curve measures pure translation equivariance. Still, the contrast is consistent across the five fitted pairs with what Chapters 7–9 built into the CNN: local sharing, pooling, and a position-discarding global-average head.

⚠ Trap: a rematch is a regime, not a referendum

These runs match the data split, parameter scale, optimizer, update count, minibatch order, and seeds. Their dot-product ledgers are close as well. They do **not** match architecture-specific tuning, every operation, wall-clock cost, or pretraining. The five runs vary initialization and order while holding one split fixed, so they are not population uncertainty estimates. The 600-image benchmark was opened in earlier chapters and is descriptive, not a fresh final test. This experiment tells us what happened here; it does not rank CNNs and ViTs in general.

16.5. Scale changes the comparison

Now harvest Chapter 15’s second promise: **pretraining is a regime, not an architecture**. ViT keeps the encoder pattern but changes the tokens, source data, and task. In the original work, the decisive source task was supervised image classification—not BERT-style masked prediction—and the amount of source data changed the comparison.

The authors pretrained matched families on random subsets of JFT containing 9, 30, and 90 million images, plus the full 300-million-image dataset. These are four measured regimes—not points on a smooth curve we are free to invent. ViT-B/32 was slightly faster than the compared ResNet-50, performed much worse on the 9-million subset, and performed better at 90 million images and above. The analogous ViT-L/16 versus larger ResNet comparison followed the same regime change.

That does not prove “attention scales forever” or that data erases every useful prior. It demonstrates the narrower—and more valuable—point: the architecture ranking can reverse when the training regime changes. Our 1,000-example scratch run and the paper’s large-source-pretraining run answer different questions.

16.5.1. Three places to buy back useful bias

Pure global attention is not the only possible endpoint. Later designs put useful image structure back in at different places.

Where the prior enters	Example	What changes
Input	Hybrid CNN–ViT	A convolutional stem creates local feature tokens before global routing
Routing	Swin Transformer	Shifted local windows and a hierarchy constrain attention and control resolution cost
Training signal	DeiT	Strong augmentation and a convolutional teacher improve data efficiency without changing the student’s basic patch encoder

DeiT makes a particularly human point. A teacher provides another target, but it does not replace the ground-truth label. Students can become better than the instructor: they listen, see other evidence, and

decide for themselves. In DeiT’s hard-distillation version, a distillation token learns from the teacher’s predicted class while [CLS] learns from the true class; the two heads are combined at inference. The teacher changes supervision, not the identity of the student.

ConvNeXt prevents a lazy conclusion in the opposite direction. Its study first modernized the training recipe, then progressively changed a ResNet toward design choices associated with contemporary Transformers. The training recipe alone raised the reported ResNet-50 ImageNet-1K accuracy from 76.1% to 78.8% before the architectural sequence was complete. Architecture, augmentation, optimization, and data interact—“CNN versus Transformer” is rarely a comparison between two isolated operators.

16.6. The word “scaling” changes jobs

We have now used *scaling* in two different ways. EfficientNet asks how to enlarge one architecture family under a compute multiplier. Kaplan and Chinchilla ask how measured loss changes as parameters, data, and training compute grow. One is a design rule; the other is an empirical forecast.

16.6.1. Compound scaling: three knobs, one budget

Suppose a baseline convolutional network has depth d_0 , channel width w_0 , and input resolution r_0 . Scaling just one knob creates a bottleneck elsewhere—a deeper network still sees the same pixels, a wider network still has the same receptive-field schedule, and higher resolution asks an unchanged feature extractor to digest more detail. EfficientNet couples the knobs with one coefficient ϕ :

$$d_\phi = d_0 \alpha^\phi, \quad w_\phi = w_0 \beta^\phi, \quad r_\phi = r_0 \gamma^\phi. \quad (16.3)$$

For ordinary dense convolutions, the leading operation count scales roughly as

$$\text{FLOPs} \propto d w^2 r^2.$$

Therefore one step in ϕ multiplies the proxy by

$$\alpha \beta^2 \gamma^2 \approx 2. \quad (16.4)$$

EfficientNet’s grid search on its baseline found approximately $\alpha = 1.2$, $\beta = 1.1$, and $\gamma = 1.15$. Their product is 1.92027—not exactly two, but close enough to serve as the intended budget step.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Audit the EfficientNet compound-scaling constraint.
-

CODE

```
# [1]
alpha, beta, gamma = 1.2, 1.1, 1.15
phis = np.arange(5)
depth_multiplier = alpha ** phis
width_multiplier = beta ** phis
resolution_multiplier = gamma ** phis
compute_multiplier = (alpha * beta**2 * gamma**2) ** phis

# [2]
print(f"alpha * beta^2 * gamma^2 = {alpha * beta**2 * gamma**2:.5f}")
```

alpha * beta^2 * gamma^2 = 1.92027

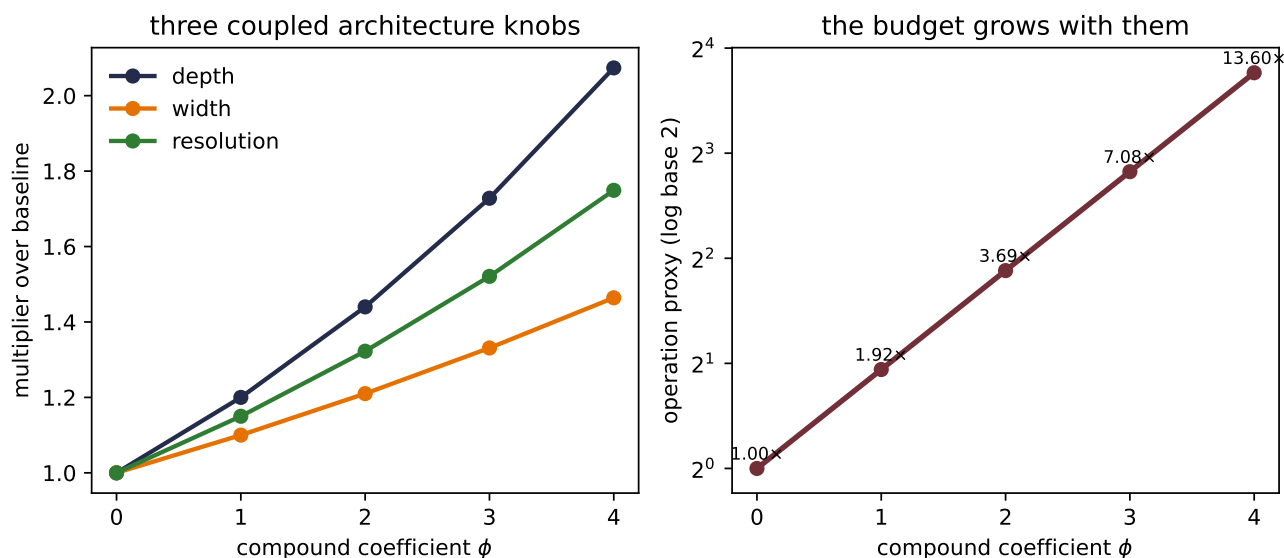


Figure 16.3.: EfficientNet’s published compound multipliers. Each step in the compound coefficient ϕ grows depth, width, and resolution modestly; their approximate convolutional-operation proxy grows by 1.92027 per step. These are normalized design multipliers, not measured accuracy or runtime.

The constants are an empirical design choice from a particular search and model family, not a law that every network must obey. The durable lesson is the budget discipline—when several resources jointly limit performance, scaling one while freezing the others can spend compute badly.

16.7. A fitted power law, not a promise

Kaplan and colleagues asked a different question about decoder-only language models. Holding other resources out of the way, how does test cross-entropy L change with non-embedding parameters N , dataset tokens D , or compute-optimally allocated training compute $C_{\min}$? Across their measured regime, the answer was well described by power laws:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076,$$

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D}, \quad \alpha_D \approx 0.095,$$

$$L(C_{\min}) = \left(\frac{C_c}{C_{\min}}\right)^{\alpha_C}, \quad \alpha_C \approx 0.050.$$

Here N_c, D_c, C_c are fitted scale constants. Their values depend on the data, tokenizer, vocabulary, loss definition, and compute units—the small exponents are the more portable description of diminishing returns. These are conditional fits. The parameter curve assumes data is not the bottleneck, and the data curve assumes model capacity is not the bottleneck.

A power law becomes a straight line after taking logs:

$$\log L = \log K - \alpha \log x.$$

That makes extrapolation visually tempting: fit a line—extend it—and read a future loss. But a straight line on transformed axes does not repeal the conditions that produced it.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Turn the published Kaplan exponents into auditable multipliers.

CODE

```
# [1]
scales = np.logspace(0, 3, 200)
kaplan_exponents = {
    "parameters, $\alpha=0.076$": (0.076, "#232D4B"),
    "tokens, $\alpha=0.095$": (0.095, "#E57200"),
    "optimal compute, $\alpha=0.050$": (0.050, "#2E7D32"),
}

# [2]
for label, (exponent, _) in kaplan_exponents.items():
    doubling_factor = 2 ** (-exponent)
    print(
        f"{label}: doubling leaves {doubling_factor:.4f} "
        f"({100 * (1 - doubling_factor):.1f}% reduction)"
    )
```

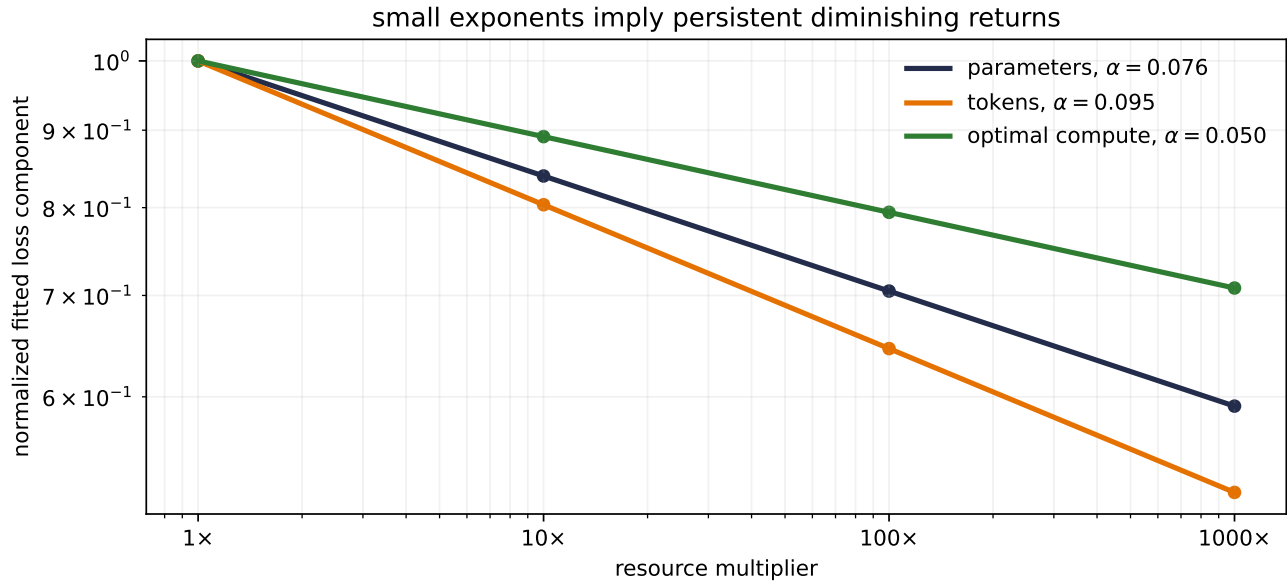


Figure 16.4.: Implications of Kaplan et al.’s published exponents, normalized to one at the left endpoint. The markers and lines are calculated from the fitted laws—not raw training runs from this book. Even a thousand-fold resource increase leaves a substantial fraction of the fitted component, which is the arithmetic of diminishing returns.

parameters, $\alpha=0.076$: doubling leaves 0.9487 (5.1% reduction)
tokens, $\alpha=0.095$: doubling leaves 0.9363 (6.4% reduction)
optimal compute, $\alpha=0.050$: doubling leaves 0.9659 (3.4% reduction)

The doubling arithmetic is now pinned. Twice as many parameters multiplies the fitted parameter-limited loss by $2^{-0.076} = 0.949$, a 5.1% reduction. Doubling nonredundant data gives $2^{-0.095} = 0.936$, a 6.4% reduction. Doubling optimally allocated compute gives $2^{-0.050} = 0.966$, a 3.4% reduction. Comparing those percentages does **not** say that one parameter and one token have comparable cost or value—the units, constants, and constraints differ.

i A scaling law is a fitted conditional

The measured range, data distribution, objective, architecture family, optimizer, and nonbinding-resource assumptions are part of the result. An extrapolation is a forecast under continued conditions, not a guarantee. Kaplan’s basic one-variable fits above do not contain an explicit additive 1.69 floor; that constant belongs to the later Chinchilla joint fit.

Kaplan’s compute-optimal analysis suggested spending additional compute mostly on parameters:

$$N_{\text{opt}} \propto C^{0.73}, \quad D_{\text{opt}} \propto C^{0.27}.$$

That recommendation was influential—and it was later revised by a study designed to measure the allocation more directly.

16.8. Compute-optimal means balanced

Hoffmann and colleagues trained more than 400 language models while varying model size and token count. Their Chinchilla study estimated the compute-optimal frontier three ways.

Fitting approach	N_{opt} exponent	D_{opt} exponent
Minimum over training curves	0.50	0.50
IsoFLOP profiles	0.49	0.51
Parametric loss model	0.46	0.54

The exact answers differ—especially under long extrapolation—but all three move much more of the growing budget toward data than Kaplan’s 0.73/0.27 allocation. That agreement is the central result—not one immortal ratio.

The third approach fits a joint loss surface:

$$\widehat{L}(N, D) = E + \frac{A}{N^\alpha} + \frac{B}{D^\beta}, \quad (16.5)$$

with published constants

$$E = 1.69, \quad A = 406.4, \quad B = 410.7, \quad \alpha = 0.34, \quad \beta = 0.28.$$

E is the fitted irreducible term for that distribution and objective; A/N^α is the model-limited contribution; and B/D^β is the data-limited contribution. This decomposition is empirical. It is not a theorem that every language model has exactly three such terms.

For a dense Transformer, the study uses the approximation

$$C \approx 6ND, \quad (16.6)$$

where C is training floating-point operations, N is non-embedding parameters, and D is processed training tokens. The factor six approximates a forward pass plus a backward pass. It omits context-dependent and embedding details, so it is a planning equation rather than a profiler.

Fixing C forces $D = C/(6N)$. Too large a model leaves too few tokens; too many tokens leave too little model. Substituting the constraint into Equation 16.5 and differentiating gives

$$G = \left(\frac{\alpha A}{\beta B} \right)^{1/(\alpha+\beta)},$$

$$N_{\text{opt}} = G \left(\frac{C}{6} \right)^{\beta/(\alpha+\beta)}, \quad D_{\text{opt}} = G^{-1} \left(\frac{C}{6} \right)^{\alpha/(\alpha+\beta)}. \quad (16.7)$$

The rounded constants imply exponents 0.452 and 0.548, close to the reported parametric-frontier fit of 0.46 and 0.54. At the reference budget $C = 5.76 \times 10^{23}$, this particular fitted surface reaches its minimum near 32.2 billion parameters and 2.98 trillion tokens. The other two methods place the model

farther right, in roughly the 40–70 billion range. The paper chose 70 billion parameters and 1.4 trillion tokens for Chinchilla, at the upper end of that range, partly for data and computational efficiency.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Define the reusable `fitted_chinchilla_loss` helper.
3. Report the joint-fit optimum beside the 20-token heuristic.

CODE

```
# [1]
E_inf, A_fit, B_fit = 1.69, 406.4, 410.7
alpha_fit, beta_fit = 0.34, 0.28
C_budget = 5.76e23

G_fit = (
    alpha_fit * A_fit / (beta_fit * B_fit)
) ** (1 / (alpha_fit + beta_fit))
N_joint = G_fit * (C_budget / 6) ** (
    beta_fit / (alpha_fit + beta_fit)
)
D_joint = (C_budget / 6) / N_joint

N_twenty = math.sqrt(C_budget / 120)
D_twenty = 20 * N_twenty

N_grid = np.logspace(np.log10(4e9), np.log10(5e11), 400)
D_grid = C_budget / (6 * N_grid)
loss_grid = (
    E_inf + A_fit / N_grid**alpha_fit + B_fit / D_grid**beta_fit
)

# [2]
def fitted_chinchilla_loss(parameters: float, tokens: float) -> float:
    return (
        E_inf
        + A_fit / parameters**alpha_fit
        + B_fit / tokens**beta_fit
    )

points = [
    (N_joint, D_joint, "joint-fit minimum", "#2E7D32"),
    (N_twenty, D_twenty, "20-token heuristic", "#E57200"),
    (280e9, C_budget / (6 * 280e9), "Gopher-sized model", "#722F37"),
]

# [3]
print(
    f"joint-fit minimum: N={N_joint / 1e9:.2f}B, "
```

```

    f"D={D_joint / 1e12:.3f}T, D/N={D_joint / N_joint:.1f}"
)
print(
    f"20-token heuristic: N={N_twenty / 1e9:.2f}B, "
    f"D={D_twenty / 1e12:.3f}T, C={6 * N_twenty * D_twenty:.2e}"
)

```

joint-fit minimum: N=32.19B, D=2.982T, D/N=92.6
 20-token heuristic: N=69.28B, D=1.386T, C=5.76e+23

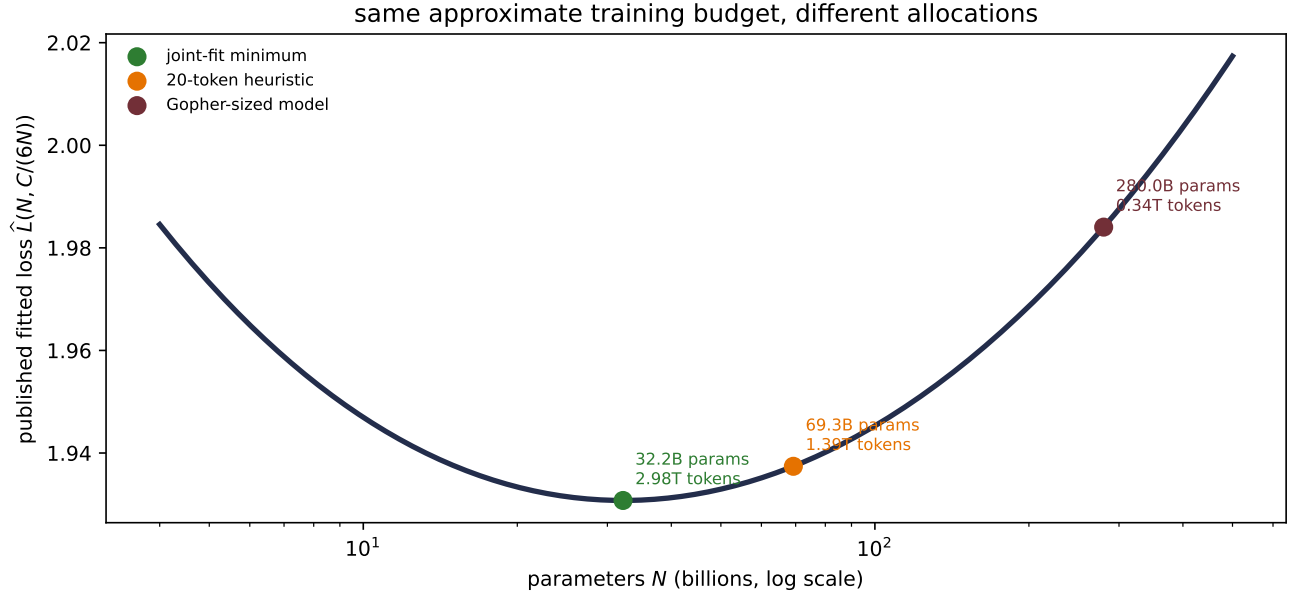


Figure 16.5.: One reference iso-compute curve under the published Chinchilla parametric loss fit. The rounded joint fit reaches its mathematical minimum near 32.2B parameters; the separate 20-tokens-per-parameter heuristic lands near 69.3B/1.386T, close to the 70B/1.4T Chinchilla design. A Gopher-sized 280B allocation leaves far fewer tokens. The curve is a fitted surface evaluation, not a new training experiment.

This distinction matters. The memorable **20 tokens per parameter** comes from the near-balanced frontier estimated by the first approaches: their table gives about 20.2 billion tokens for 1 billion parameters, 205.1 billion for 10 billion, and 1.5 trillion for 67 billion. If we impose the rounded heuristic $D \approx 20N$ at the 5.76×10^{23} budget, Equation 16.6 gives

$$N \approx \sqrt{\frac{C}{120}} = 69.3 \text{ billion}, \quad D \approx 1.386 \text{ trillion}.$$

That arithmetic is close to the actual Chinchilla design, but it is **not** the minimum of the rounded joint fit drawn above. Different fitting methods produced a broad range, and the authors made an engineering choice within it. Hiding that difference would make a heuristic look more exact than the evidence.

The historical comparison makes the allocation tangible. Gopher used about 280 billion parameters and 300 billion training tokens. Chinchilla used 70 billion parameters and 1.4 trillion tokens: four times fewer parameters and about 4.7 times more tokens, at approximately the same reported training budget.

The smaller, longer-trained model achieved lower loss and outperformed the larger model across the paper’s broad evaluation suite. More parameters had not been the only way to spend the budget—and, in that regime, they had not been the best way.

 Twenty is not a constant of nature

The 20-to-1 ratio is an approximate rule from this study, model family, data, and frontier. The fitted methods do not agree exactly, and the ratio changes weakly with compute even inside the parametric model. Data quality, repetition, tokenization, objective, architecture, and downstream use can all move the useful allocation. Use the rule to challenge an obviously undertrained model, not to end the analysis.

16.9. Where the forecast stops

Scaling laws are powerful precisely because they compress many expensive runs into a small predictive object. That compression has boundaries.

Loss is not a capability map. A smoother aggregate cross-entropy curve does not tell us that every downstream skill appears smoothly, or exactly when a particular behavior becomes reliable.

Tokens are not interchangeable. A trillion repeated, low-quality, contaminated, or mismatched tokens need not buy what a trillion useful, diverse tokens buy. D counts processed tokens; it does not measure novelty or truth.

The law is local to a recipe and range. Change the architecture, optimizer, data mixture, context length, or evaluation distribution and the fitted constants can move. Extrapolation beyond the measured range adds uncertainty even when the line looks immaculate on log–log axes.

Training-optimal is not serving-optimal. Chinchilla allocates a training budget; it does not make the resulting backbone cheap to store, run, or adapt. [Chapter 17](#) will harvest that mismatch: what can we change when moving every pretrained weight is no longer affordable?

This closes Part IV’s route from fixed lookup to global learned routing. We began with a memory bank, learned how to score it, let every token query every other token, changed the visibility to pretrain representations, and finally let images enter as tokens. The operator was only half the story. The data and budget that train it are part of the model too.

 Check yourself

Close the book for one minute and retrieve the chapter’s three scales.

- For a 28×28 image with 4×4 patches, how many patch tokens appear before [CLS]?
- Why does a matched toy CNN–ViT rematch describe a regime rather than declare a winner?
- Under fixed training compute, why must parameter count and token count be chosen together?

16.10. Okay, so — patches become tokens, but regime still matters

1. **An image can become a token sequence.** Split it into $P \times P$ patches, flatten to width CP^2 , and share a learned projection. Unfold-plus-linear is exactly a convolution with kernel and stride

P after reshaping its weights.

2. **ViT is an encoder transplant, not assumption-free learning.** A learned [CLS] meeting place, learned positions, pre-LayerNorm blocks, and a linear head reuse the Transformer pattern while retaining a small set of image and optimization biases.
3. **Inductive bias is a trade.** The five-seed Fashion rematch favors the CNN on clean validation and much more strongly under shift, but it is a tiny scratch regime—not an architecture referendum. Large supervised pretraining changed the comparison in the original ViT study.
4. **Architecture and recipe cannot be cleanly collapsed into a brand name.** Hybrid stems, windowed routing, distillation, and ConvNeXt put priors in different places. Data, optimization, and operator design interact.
5. **Scaling has two meanings here.** EfficientNet’s compound rule coordinates depth, width, and resolution; empirical scaling laws forecast loss as parameters, tokens, and compute change. Both are resource-allocation tools, but they answer different questions.
6. **Compute-optimal training balances model and data.** Chinchilla revised the parameter-heavy Kaplan allocation, while also showing why a memorable 20-token heuristic must remain a heuristic. A fitted law carries its regime, assumptions, and uncertainty with it.

Sources and further reading

- Dosovitskiy et al., *An Image Is Worth 16×16 Words: Transformers for Image Recognition at Scale*: the original ViT architecture, position and pooling ablations, and data-scale comparison.
- Tan and Le, *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*: compound depth, width, and resolution scaling.
- Touvron et al., *Training Data-Efficient Image Transformers & Distillation Through Attention*: DeiT’s augmentation and teacher-guided training recipe.
- Liu et al., *Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows*: local shifted windows, hierarchy, and resolution-aware attention cost.
- Liu et al., *A ConvNet for the 2020s*: the ConvNeXt modernization study and architecture–recipe interaction.
- Kaplan et al., *Scaling Laws for Neural Language Models*: the parameter, data, and compute power laws and the earlier compute allocation.
- Hoffmann et al., *Training Compute-Optimal Large Language Models*: the three Chinchilla frontier estimates, joint loss fit, and Gopher comparison.
- Guo et al., *On Calibration of Modern Neural Networks*: establishes single-parameter temperature scaling as a strong post-hoc calibration baseline and motivates Exercise 6’s held-out fit.

Exercises

1. **(Pencil.)** A batch contains 12 RGB images at 384×384 . With $P = 16$ and model width $d = 768$, give the shapes of the raw batch, flattened patch tensor, projected patches, and sequence after [CLS]. Repeat the token and per-head score-entry counts for $P = 8$. By what factor did halving P change each count?
2. **(Code.)** Generalize the patch-equivalence cell to $C = 3$ and a rectangular image whose height and width are each divisible by P . State the required unfold ordering, reshape a matrix $\mathbf{E} \in \mathbb{R}^{CP^2 \times d}$ into convolution weights, and assert equality to numerical tolerance. Explain why overlap appears when stride is smaller than P .

3. **(Audit.)** Reproduce all five paired Fashion results and compute the paired clean-validation gaps, clean-to-four-pixel drops, and schedule hashes. Then change exactly one factor—add translated training examples—while keeping the split, seeds, initialization rule, model sizes, and minibatch schedules auditable. What question does the new comparison answer, and what still prevents a general CNN-versus-ViT claim?
4. **(Pencil.)** Using EfficientNet’s published constants, compute the depth, width, resolution, and operation-proxy multipliers at $\phi = 3$. Then use Kaplan’s exponents to compute the fitted component remaining after a 10-fold increase in parameters, data, and compute. Why can you not compare those three numbers as benefits per dollar?
5. **(Pencil.)** Starting from Equation 16.5 and Equation 16.6, derive Equation 16.7. Evaluate both the joint-fit optimum and the separate $D = 20N$ heuristic at a budget of 1.2×10^{22} FLOPs. Explain why disagreement between the answers is evidence to report rather than an algebra mistake, and name two deployment constraints neither calculation optimizes.
6. **(Code.)** Freeze one pinned Fashion CNN checkpoint from this chapter and fit one positive scalar temperature T on validation logits by minimizing NLL. With seed 6050 and fixed bin edges, draw validation and sealed-test reliability diagrams before and after scaling; report NLL, ECE, accuracy, and the fraction of class rankings that changed. Verify numerically that dividing every class logit by the same positive T leaves argmax predictions unchanged. State what calibration improved and what it did not. This is post-hoc calibration of a frozen model, not Chapter 20’s training-time γ . *Named wrong answer:* “better ECE means a more accurate classifier” — this post-hoc dial changes confidence, not the predicted class.

Part V.

The Pretrained Era

17. Adapting Pretrained Models: Prompting, PEFT, Quantization

Chapter 9 left us a transfer rule: importing a representation is most promising when **labels are scarce, the target is feature-hungry, and the source supplies useful coverage at a matched scale**. Chapter 15 showed the conventional next step—fine-tuning changes every encoder weight. Chapter 16 then planted a second warning: **training-optimal is not serving-optimal**. Chinchilla’s allocation can spend a pretraining budget well while leaving us with a backbone that is expensive to copy, update, or store once per downstream task.

Let us put those promises on one ledger. A pretrained model creates three different bills:

1. the **backbone storage**—a base shared across tasks, or a full task-specific copy when sharing is abandoned;
2. the **incremental task state**—the prompt, adapter, or unmerged delta that makes one task different from another; and
3. the **transient work**—context tokens, activations, gradients, optimizer state, and caches needed while adapting or running the model.

These bills are related—but they are not interchangeable. Freezing a billion weights removes their gradients—it does not make the billion weights disappear. Packing each weight into fewer bits reduces a storage lower bound—it does not prove that a particular kernel will run faster. Adding demonstrations changes no weight—it still lengthens the sequence the model must process. A merged LoRA or fully tuned checkpoint can collapse base and delta into one artifact—but then each task carries a full backbone-sized copy instead of sharing the original.

The useful organizing questions are therefore not simply “How many parameters?” They are **where does task-specific information live, and which object does the method change?** Hard prompts and retrieved documents place task information in context. Parameter-efficient fine-tuning (PEFT) places it in a restricted set of learned values. Quantization need not add task information at all—it changes the numerical representation of a base or adapter. QLoRA composes the last two—it is not a fourth kind of objective.

17.1. Prompting: move evidence into context

A decoder in [Chapter 14](#) already defines a conditional distribution over the next token. Prompting changes the prefix on which that fixed distribution is conditioned. For an instruction I , demonstrations (x_i, y_i) , and a query x_* —a discrete prompted prediction can be written

$$\hat{y} = \arg \max_y p_{\theta_0}(y \mid I, (x_1, y_1), \dots, (x_k, y_k), x_*), \quad \theta_{\text{after}} = \theta_{\text{before}} = \theta_0. \quad (17.1)$$

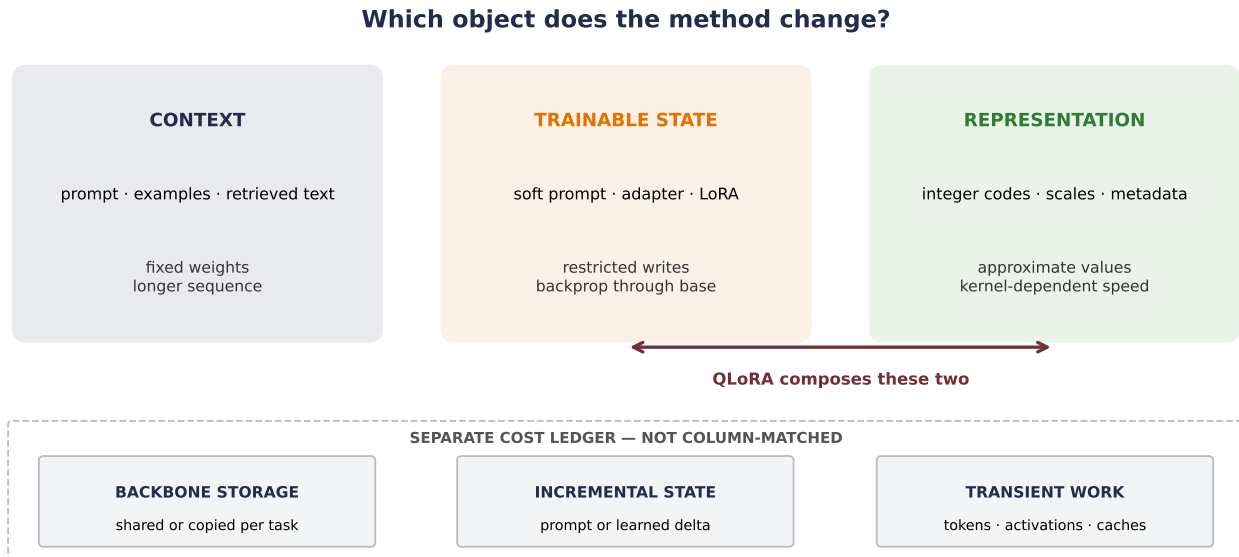


Figure 17.1.: Three levers act on three different objects. Prompting changes the context presented to a fixed model; PEFT permits a small learned state to receive gradient blame; quantization changes how fixed values are represented. QLoRA combines a quantized frozen base with a trainable low-rank branch. The separate cost ledger below is deliberately not column-matched: every lever can affect more than one bill, and none removes all backbone storage or transient work.

With no demonstrations, this is **zero-shot** prompting. With a few, it is **few-shot prompting**; when the model uses those examples to infer the task inside its forward computation, we call the behavior **in-context learning (ICL)**. The word *learning* names a behavioral change with context—not a checkpoint update.

Brown and colleagues found that few-shot gains often grew with model scale in GPT-3—but that is an empirical result over particular models and tasks. It does not establish universal parameter thresholds, nor does it settle the mechanism by which a Transformer uses demonstrations. Attention makes earlier examples visible and routable; saying it literally “performs analogy” would outrun the evidence. Prompt order, label words, formatting, and example choice can all matter. Controlled ablations have found that demonstration inputs, the label space and its mapping, formatting, and ordering can contribute differently across tasks. Those results constrain simple stories about copying examples, but they do not settle one universal causal mechanism for ICL.

Instructions that ask for intermediate steps can improve some multi-step tasks. Those visible steps are generated tokens, however—not a guaranteed faithful window into the model’s internal computation. The same logits and decoding cautions from [Chapter 11](#) still apply.

i Prompting is not prompt tuning

A hard prompt is a sequence of ordinary token IDs chosen by a person or program. It receives no gradient. **Prompt tuning**, introduced later in this chapter, learns continuous vectors by backpropagation and is therefore a PEFT method. Similar names do not mean the same training procedure.

Zero trainable parameters is not zero cost—demonstrations consume context length. They increase

prefill work, the initial forward pass that processes the supplied prefix, and occupy entries in the **key/value (KV) cache**, the stored attention keys and values reused while autoregressively generating later tokens. They also expose task data at inference time. Prompting trades a new checkpoint for a longer, more carefully constructed input.

17.1.1. A frozen-context mechanism test

Can a fixed network genuinely change its answer because examples in the prompt reveal a task? We can test that mechanism without claiming to reproduce natural-language ICL.

Each synthetic episode secretly permutes four group symbols onto four label symbols. A demonstration reveals one group–label pair; demo groups are distinct. A two-shot prompt might read [BOS] **group C, label A; group A, label D**; [QUERY] **group A**, whose target is **label D**. The query asks for another group’s label. A tiny causal Transformer is meta-trained across many such episodes, cycling through one to four demonstrations. During evaluation, it receives a new random mapping and **no parameter update**.

The information ceiling is known before training. With $k < 4$ demonstrations, the query is already covered with probability $k/4$. Otherwise, its label is uniform among the $4 - k$ unused labels. Therefore

$$\Pr(\text{correct} \mid k) = \begin{cases} \frac{k}{4} + \left(1 - \frac{k}{4}\right) \frac{1}{4 - k} = \frac{k + 1}{4}, & 0 \leq k \leq 3, \\ 1, & k = 4. \end{cases} \quad (17.2)$$

At $k = 3$, even an unseen query group is solvable by elimination—a distinction that will tell us whether the model merely copies a seen pair or uses the whole mapping.

The code uses `nn.TransformerEncoderLayer` as a compact container for the same pre-LayerNorm attention–residual–FFN structural pattern built from first principles in [Chapter 14](#). Its GELU choice differs from that chapter’s teaching block; the wrapper does not hide the causal or padding masks, embeddings, or classifier. Token IDs have shape $[B, 11]$, hidden states $[B, 11, 48]$, and the query logits $[B, 4]$.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `sample_episode_bank`, `render_episodes`, and `ContextLearner`.
 3. Define the reusable helpers: `evaluate_context` and `run_context_seed`.
 4. Meta-train a tiny causal Transformer, then audit frozen-weight in-context adaptation over five seeds.
-

CODE

```
# [1]
PAD, BOS, QUERY = 0, 1, 2
GROUP0, LABEL0 = 3, 7
VOCAB, SEQUENCE_LENGTH, N_CLASSES = 11, 11, 4
```

```

# [2]
def sample_episode_bank(
    batch_size: int, generator: torch.Generator
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    mapping = torch.rand(batch_size, N_CLASSES, generator=generator).argsort(dim=1)
    demo_groups = torch.rand(
        batch_size, N_CLASSES, generator=generator
    ).argsort(dim=1)
    query_groups = torch.randint(
        N_CLASSES, (batch_size,), generator=generator
    )
    return mapping, demo_groups, query_groups

def render_episodes(
    bank: tuple[torch.Tensor, torch.Tensor, torch.Tensor], k: int
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    mapping, demo_groups, query_groups = bank
    tokens = torch.full((len(mapping), SEQUENCE_LENGTH), PAD, dtype=torch.long)
    tokens[:, 0] = BOS
    for slot in range(k):
        group = demo_groups[:, slot]
        label = mapping.gather(1, group[:, None]).squeeze(1)
        tokens[:, 1 + 2 * slot] = GROUPO + group
        tokens[:, 2 + 2 * slot] = LABELO + label
    tokens[:, 9] = QUERY
    tokens[:, 10] = GROUPO + query_groups
    targets = mapping.gather(1, query_groups[:, None]).squeeze(1)
    covered = (
        (demo_groups[:, :k] == query_groups[:, None]).any(dim=1)
        if k else torch.zeros(len(mapping), dtype=torch.bool)
    )
    return tokens, targets, covered

class ContextLearner(nn.Module):
    def __init__(self, width: int = 48, heads: int = 4, layers: int = 2) -> None:
        super().__init__()
        self.token = nn.Embedding(VOCAB, width, padding_idx=PAD)
        self.position = nn.Embedding(SEQUENCE_LENGTH, width)
        layer = nn.TransformerEncoderLayer(
            d_model=width, nhead=heads, dim_feedforward=2 * width,
            dropout=0.0, batch_first=True, norm_first=True, activation="gelu",
        )
        self.encoder = nn.TransformerEncoder(
            layer, layers, norm=nn.LayerNorm(width), enable_nested_tensor=False
        )
        self.head = nn.Linear(width, N_CLASSES)

```



```

        self.register_buffer(
            "causal",
            torch.triu(torch.ones(SEQUENCE_LENGTH, SEQUENCE_LENGTH,
→ dtype=torch.bool),
                        diagonal=1),
        )

    def forward(self, tokens: torch.Tensor) -> torch.Tensor:
        positions = torch.arange(SEQUENCE_LENGTH, device=tokens.device)
        hidden = self.token(tokens) + self.position(positions)[None]
        hidden = self.encoder(
            hidden, mask=self.causal, src_key_padding_mask=tokens.eq(PAD)
        )
        return self.head(hidden[:, -1])

# [3]
@torch.no_grad()
def evaluate_context(
    model: nn.Module, seed: int, n: int = 8192
) -> tuple[list[dict], bool]:
    model.eval()
    before = [parameter.detach().clone() for parameter in model.parameters()]
    bank = sample_episode_bank(n, torch.Generator().manual_seed(seed + 10_000))
    rows = []
    for k in range(5):
        tokens, targets, covered = render_episodes(bank, k)
        correct = model(tokens).argmax(1).eq(targets)
        rows.append({
            "k": k,
            "accuracy": correct.float().mean().item(),
            "covered": correct[covered].float().mean().item()
                if covered.any() else float("nan"),
            "uncovered": correct[~covered].float().mean().item()
                if (~covered).any() else float("nan"),
        })
    unchanged = all(
        torch.equal(old, new) for old, new in zip(before, model.parameters())
    )
    return rows, unchanged

def run_context_seed(seed: int, steps: int = 1200) -> dict:
    torch.manual_seed(seed)
    generator = torch.Generator().manual_seed(seed)
    model = ContextLearner()
    optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=1e-2)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=steps, eta_min=3e-5
    )

```

```

)
model.train()
for step in range(steps):
    k = 1 + step % 4
    bank = sample_episode_bank(256, generator)
    tokens, targets, _ = render_episodes(bank, k)
    loss = F.cross_entropy(model(tokens), targets)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()
    scheduler.step()
rows, unchanged = evaluate_context(model, seed)
return {
    "seed": seed,
    "rows": rows,
    "unchanged": unchanged,
    "parameters": sum(parameter.numel() for parameter in model.parameters()),
}

CONTEXT_SEEDS = list(range(6050, 6055))
context_runs = [run_context_seed(seed) for seed in CONTEXT_SEEDS]
context_accuracy = np.array([
    [row["accuracy"] for row in run["rows"]] for run in context_runs
])
context_ceiling = np.array([0.25, 0.50, 0.75, 1.00, 1.00])

print("model parameters:", f'{context_runs[0]["parameters"]:,}')
print("k mean accuracy seed sd information ceiling")
# [4]
for k in range(5):
    print(
        f'{k} {context_accuracy[:, k].mean():.3f} '
        f'{context_accuracy[:, k].std(ddof=1):.3f} '
        f'{context_ceiling[k]:.2f}'
    )
print("all evaluation passes left weights unchanged:",
      all(run["unchanged"] for run in context_runs))

```

```

model parameters: 39,268
k mean accuracy seed sd information ceiling
0 0.252 0.003 0.25
1 0.500 0.005 0.50
2 0.749 0.002 0.75
3 1.000 0.000 1.00
4 1.000 0.000 1.00
all evaluation passes left weights unchanged: True

```

The 39,268-parameter network lands at mean accuracies 0.252, 0.500, 0.749, 1.000, 1.000 for $k = 0, 1, 2, 3, 4$,

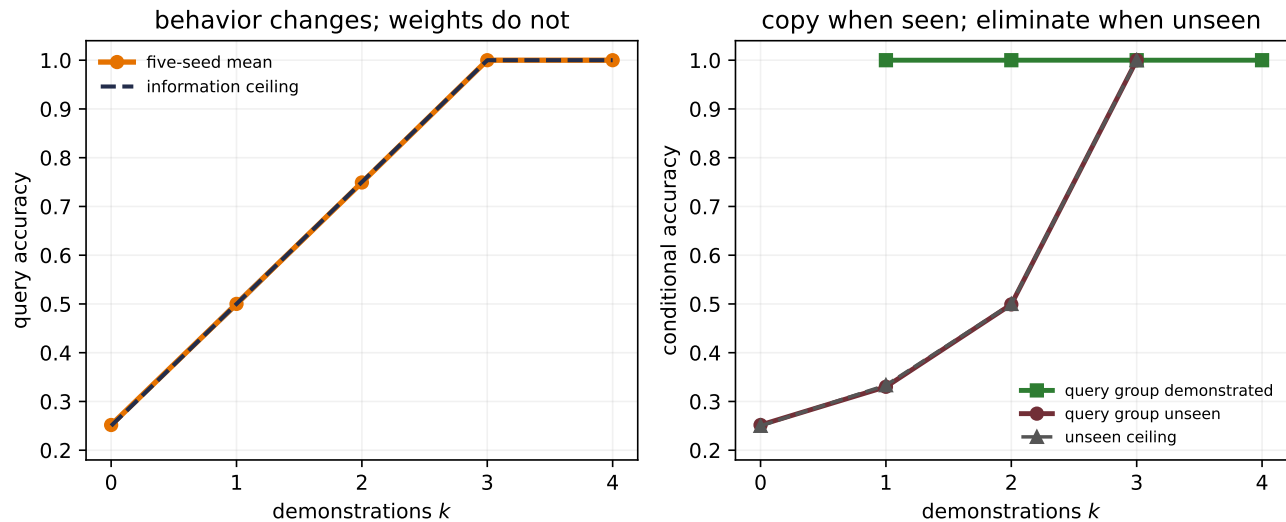


Figure 17.2.: A frozen model changes behavior as demonstrations reveal an episode’s hidden label mapping. Across five independently trained seeds, overall accuracy (left) closely tracks the analytical information ceiling. Conditional accuracy (right) is perfect when the query group appeared; for unseen groups it follows the shrinking set of unused labels and reaches 1 at three demonstrations by elimination. Evaluation uses one nested episode bank per seed, and every parameter is bitwise unchanged across all five context lengths.

respectively. The analytic ceilings are 0.25, 0.50, 0.75, 1, 1. More revealingly, unseen-query accuracy rises from 0.252 to 0.330, 0.499, and 1.000. The model is not only copying a matching demonstration; at $k = 3$ it identifies the remaining label.

⚠ An existence proof in a designed regime

The model was meta-trained on exactly this family of random permutation tasks. The study demonstrates that a frozen causal Transformer can use prompt evidence in this controlled distribution. It does not estimate natural-language ICL quality, explain the mechanism inside a large language model, or show that more examples always help. Across runs, initialization, meta-training episodes, and the finite 8,192-episode evaluation bank all change. The seed SD therefore measures combined run-to-run repeatability—not task or domain diversity, and not optimization alone.

17.1.2. Retrieval is another context lever

A prompt can contain information selected at query time rather than written by hand. **Retrieval-augmented generation (RAG)** first searches an external corpus, then places retrieved passages beside the instruction and query before generation. The generator’s weights may remain fixed—the task-specific state lives partly in the index and partly in the retrieved context.

This reconnects to [Chapter 12](#). A retriever computes query–document similarities, selects neighbors, and hands their content to a downstream weighted computation. Modern systems can learn the retriever, the generator, or both, but the basic context-augmentation move is distinct from PEFT.

Retrieval moves rather than abolishes failure—a missed document cannot support the answer; a highly ranked irrelevant passage can distract it; retrieved text can contain malicious instructions; and a plausible

citation does not prove that the generated claim follows from the cited evidence. RAG is attractive when facts change faster than checkpoints or when provenance matters, but it is a pipeline whose retrieval and generation stages must be evaluated separately.

17.2. What if the prompt itself were learnable?

Hard prompts spend human effort and context tokens. The chapter’s familiar refrain now applies: **what if the prompt itself were learnable?** Let $\mathbf{P} \in \mathbb{R}^{m \times d}$ be m continuous vectors with the same width as token embeddings. For an n -token input, define $\text{Embed}(x) \in \mathbb{R}^{n \times d}$. Prompt tuning prepends the learned rows,

$$\mathbf{H}^{(0)} = [\mathbf{P}; \text{Embed}(x)], \quad \min_{\mathbf{P}} - \sum_i \log p_{\theta_0, \mathbf{P}}(y_i | x_i), \quad (17.3)$$

while freezing θ_0 . Before any task head, the learned state contains exactly md values. Those values are not words—and need not be nearest to any vocabulary embedding. They are virtual tokens optimized by gradient descent, and they still occupy m virtual sequence positions during use.

Prefix tuning moves a related idea deeper. At every Transformer layer ℓ , let the ordinary keys and values be $\mathbf{K}_\ell, \mathbf{V}_\ell \in \mathbb{R}^{n \times d}$ and the learned prefixes be $\mathbf{P}_\ell^K, \mathbf{P}_\ell^V \in \mathbb{R}^{m \times d}$. The prefix rows extend the ordinary keys and values:

$$\mathbf{K}'_\ell = [\mathbf{P}_\ell^K; \mathbf{K}_\ell], \quad \mathbf{V}'_\ell = [\mathbf{P}_\ell^V; \mathbf{V}_\ell].$$

A direct implementation with m prefix slots, width d , and L layers stores $2Lmd$ values, although practical parameterizations can differ. Li and Liang used an auxiliary network during optimization—then retained only the learned prefix for inference. Lester and colleagues found that prompt tuning approached full tuning as T5 scale grew in their SuperGLUE study; that result is not a universal size threshold.

Two other PEFT families complete the map. **Adapters** insert a small bottleneck inside each frozen Transformer block. For bottleneck width d_b , let $\mathbf{W}_{\text{down}} \in \mathbb{R}^{d_b \times d}$, $\mathbf{b}_{\text{down}} \in \mathbb{R}^{d_b}$, $\mathbf{W}_{\text{up}} \in \mathbb{R}^{d \times d_b}$, and $\mathbf{b}_{\text{up}} \in \mathbb{R}^d$. One common residual form is

$$\text{Adapter}(\mathbf{h}) = \mathbf{h} + \mathbf{W}_{\text{up}} \sigma(\mathbf{W}_{\text{down}} \mathbf{h} + \mathbf{b}_{\text{down}}) + \mathbf{b}_{\text{up}},$$

This module adds $2dd_b + d + d_b$ parameters. **BitFit** takes the selective-update idea to an extreme—freeze pretrained weight matrices and update their existing bias terms. The original BitFit classification experiments also trained the task-specific output layer; “bias-only” describes the restriction inside the pretrained backbone. Neither method promises the same accuracy on every task. They are different restrictions on where gradient blame may write.

17.3. LoRA: a low-rank correction beside a frozen path

Consider one frozen linear map

$$\mathbf{W}_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}, \quad \mathbf{h} = \mathbf{W}_0 \mathbf{x}.$$

Full fine-tuning learns an unrestricted correction $\Delta \mathbf{W}$ with $d_{\text{out}} d_{\text{in}}$ values. **Low-Rank Adaptation (LoRA)** instead constrains the correction to a product of two thin matrices:

$$\Delta \mathbf{W} = \frac{\alpha}{r} \mathbf{B} \mathbf{A}, \quad \mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}, \quad \mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}. \quad (17.4)$$

The forward pass is

$$\mathbf{h} = \mathbf{W}_0 \mathbf{x} + \frac{\alpha}{r} \mathbf{B} (\mathbf{A} \mathbf{x}). \quad (17.5)$$

$\mathbf{A}$ compresses to r coordinates and $\mathbf{B}$ expands back to the output width. Because $\text{rank}(\mathbf{B} \mathbf{A}) \leq r$, task updates can use at most r independent linear directions in that matrix. The merged matrix is not therefore rank r —the constraint applies only to the correction.

The trainable count is

$$P_{\text{LoRA}} = r(d_{\text{in}} + d_{\text{out}}). \quad (17.6)$$

For one 4096×4096 map and $r = 16$, full tuning exposes 16,777,216 values; LoRA exposes $16(4096 + 4096) = 131,072$, or 0.78125%. A model-level percentage requires summing Equation 17.6 over the matrices actually targeted. The original LoRA experiments emphasized query and value projections—later recipes often target different or more numerous matrices. “Rank 16” alone does not specify an adapter’s size.

💡 Course-lab bridge (non-examinable): read PEFT through the algebra

Module 11’s supplied Gemma/PEFT lab expresses Equation 17.4 through a package wrapper. Treat the wrapper’s names as a translation layer, not as new mathematics:

- `r` is the inner dimension of A and B ;
- `lora_alpha / r` represents the factor α/r ;
- `target_modules` chooses which frozen maps play the role of W_0 ;
- the adapter checkpoint carries the learned factors, while the base checkpoint supplies the frozen path.

If the lab targets “all linear” modules, do not infer the adapter size from that phrase. Inspect the selected module names and shapes, sum Equation 17.6 over exactly that set, and print the parameters that still have `requires_grad=True`. Then apply the four-part freeze audit below: the base stays unchanged, only permitted factors receive gradients, and merged and unmerged outputs agree within a declared numerical tolerance. Package calls and model identifiers can evolve; these algebraic checks are the stable contract.

When the lab also stores the base on a quantization grid, the result is QLoRA, not a different LoRA equation. Keep the frozen base’s storage and compute dtypes separate from the factors’

trainable dtype, as Appendix C requires. The QLoRA section later in this chapter makes that composition explicit.

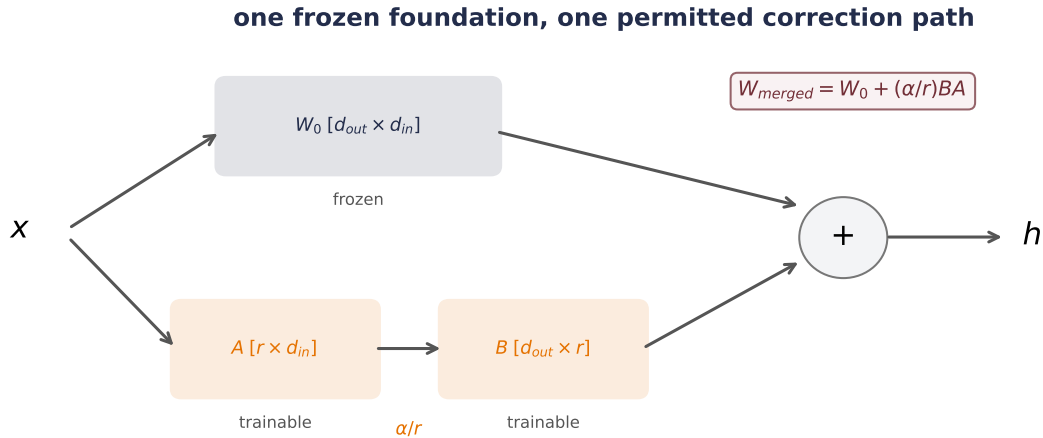


Figure 17.3.: LoRA adds a trainable low-rank branch beside a frozen linear map. Gradient blame flows through both computational paths, but only A and B receive updates. Initializing A randomly and B to zero begins with an exact zero correction while leaving a nonzero first gradient for B ; after training, the correction can be merged into a compatible base weight.

The usual initialization samples A and sets $B = 0$. The initial correction is then exactly zero—yet B can receive a gradient because Ax is generally nonzero. Setting both factors to zero would block both first gradients. The scale α/r separates a chosen adapter strength from rank, but its best value remains a tuning decision.

Once trained, a single compatible adapter can be merged:

$$W_{merged} = W_0 + \frac{\alpha}{r}BA. \quad (17.7)$$

Merged and unmerged arithmetic should agree to numerical tolerance. Only the merged form removes the extra branch at inference. Dynamic adapter switching, keeping many adapters active, or combining LoRA with a quantized base changes that systems story; a quantized merge generally requires dequantization and requantization.

💡 A four-part freeze audit

Count trainable values; assert the base tensor is bitwise unchanged; inspect that only permitted parameters receive gradients; and compare merged with unmerged outputs. A small checkpoint is not evidence that the intended tensors were frozen. The assertions belong beside the implementation, not in a comment after training.

17.3.1. Make the rank bottleneck fail on purpose

Low rank is a hypothesis about the update—not a free compression theorem. We can make its boundary visible by planting a target correction with exactly six independent diagonal directions. The frozen base

is a 32×32 map. Full tuning may move all 1,024 entries; LoRA may use ranks 1, 2, 4, 6, or 8. Inputs are fixed Gaussian vectors, targets are noiseless, and five LoRA initializations see the same problem and optimizer schedule.

Write the six planted magnitudes as $\sigma_1 \geq \dots \geq \sigma_6$. Their values are 1.20, 0.90, 0.65, 0.40, 0.22, 0.10. A rank- r correction below six must leave some directions behind. This design deliberately favors the LoRA assumption—that is what makes the capacity claim falsifiable before we run it.

Because the six diagonal basis directions are mutually orthogonal, their squared correction energy is $\sum_{j=1}^6 \sigma_j^2$. A rank- r correction can retain at most r independent directions; spending them on the r largest magnitudes leaves the concrete relative floor

$$\epsilon_r = \sqrt{\frac{\sum_{j=r+1}^6 \sigma_j^2}{\sum_{j=1}^6 \sigma_j^2}}, \quad r < 6, \quad (17.8)$$

and zero for $r \geq 6$. The plotted “capacity floor” is this calculation—not a result estimated from the optimizer.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `make_rank_problem`, `FullUpdate`, and `LoRAUpdate`.
 3. Define the reusable helpers: `train_linear_update` and `relative_update_error`.
 4. Compare frozen, full, and low-rank updates against a planted rank-six correction.
 5. Check the claimed identities, shapes, or invariants.
 6. Report or visualize the measured result.
-

CODE

```
# [1]
D_IN = D_OUT = 32
TARGET_RANK = 6
DIRECTION_MAGNITUDES = torch.tensor([1.20, 0.90, 0.65, 0.40, 0.22, 0.10])

# [2]
def make_rank_problem(
    seed: int = 1700,
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor,
          torch.Tensor, torch.Tensor]:
    generator = torch.Generator().manual_seed(seed)
    base = torch.randn(D_OUT, D_IN, generator=generator) / math.sqrt(D_IN)
    correction = torch.zeros(D_OUT, D_IN)
    diagonal = torch.arange(TARGET_RANK)
    correction[diagonal, diagonal] = DIRECTION_MAGNITUDES
    x_train = torch.randn(1024, D_IN, generator=generator)
    x_validation = torch.randn(4096, D_IN, generator=generator)
```

```

y_train = F.linear(x_train, base + correction)
y_validation = F.linear(x_validation, base + correction)
return base, correction, x_train, y_train, x_validation, y_validation

class FullUpdate(nn.Module):
    def __init__(self, base: torch.Tensor) -> None:
        super().__init__()
        self.weight = nn.Parameter(base.clone())

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return F.linear(x, self.weight)

class LoRAUpdate(nn.Module):
    def __init__(
        self, base: torch.Tensor, rank: int, generator: torch.Generator
    ) -> None:
        super().__init__()
        self.register_buffer("weight0", base.clone())
        self.A = nn.Parameter(torch.randn(rank, D_IN, generator=generator) * 0.02)
        self.B = nn.Parameter(torch.zeros(D_OUT, rank))
        self.rank = rank
        self.alpha = rank

    @property
    def scaling(self) -> float:
        return self.alpha / self.rank

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        low_rank = F.linear(F.linear(x, self.A), self.B)
        return F.linear(x, self.weight0) + self.scaling * low_rank

    def merged_weight(self) -> torch.Tensor:
        return self.weight0 + self.scaling * (self.B @ self.A)

# [3]
def train_linear_update(
    model: nn.Module, x: torch.Tensor, y: torch.Tensor, steps: int = 1200
) -> None:
    optimizer = torch.optim.Adam(model.parameters(), lr=0.03)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=steps, eta_min=3e-4
    )
    for _ in range(steps):
        loss = F.mse_loss(model(x), y)
        optimizer.zero_grad(set_to_none=True)
        loss.backward()

```



```

optimizer.step()
scheduler.step()

def relative_update_error(
    estimate: torch.Tensor, target: torch.Tensor
) -> float:
    return (torch.linalg.norm(estimate - target) / torch.linalg.norm(target)).item()

base, target_delta, x_train, y_train, x_val, y_val = make_rank_problem()
target_energy = torch.sum(DIRECTION_MAGNITUDES ** 2)

full_model = FullUpdate(base)
# [4]
train_linear_update(full_model, x_train, y_train)
with torch.no_grad():
    full_mse = F.mse_loss(full_model(x_val), y_val).item()
    full_relative_error = relative_update_error(
        full_model.weight - base, target_delta
    )
    frozen_mse = F.mse_loss(F.linear(x_val, base), y_val).item()

LORA_RANKS = [1, 2, 4, 6, 8]
lora_runs = []
# [5]
for seed in range(6050, 6055):
    for rank in LORA_RANKS:
        model = LoRAUpdate(
            base, rank, torch.Generator().manual_seed(seed + rank)
        )
        frozen_before = model.weight0.clone()
        train_linear_update(model, x_train, y_train)
        with torch.no_grad():
            merged = model.merged_weight()
            merge_error = (
                model(x_val[:64]) - F.linear(x_val[:64], merged)
            ).abs().max().item()
            assert torch.equal(frozen_before, model.weight0)
            assert merge_error < 2e-6
            learned_delta = merged - base
            lora_runs.append({
                "seed": seed,
                "rank": rank,
                "trainable": rank * (D_IN + D_OUT),
                "validation_mse": F.mse_loss(model(x_val), y_val).item(),
                "relative_error": relative_update_error(learned_delta, target_delta),
                "merge_error": merge_error,
            })

```

```
# [6]
print(f"frozen validation MSE: {frozen_mse:.6f}")
print(
    f"full update: MSE={full_mse:.2e}, relative delta error="
    f"{full_relative_error:.2e} (1,024 trainable)"
)
print("rank trainable mean validation MSE mean relative update error")
for rank in LORA_RANKS:
    selected = [row for row in lora_runs if row["rank"] == rank]
    print(
        f"{rank:>4} {selected[0]['trainable']:>9,} "
        f"{np.mean([row['validation_mse'] for row in selected]):>19.8f} "
        f"{np.mean([row['relative_error'] for row in selected]):>26.6f}"
    )
print("largest merged/unmerged difference:",
      f"{max(row['merge_error'] for row in lora_runs):.2e}")
```

```
frozen validation MSE: 0.091155
full update: MSE=6.39e-15, relative delta error=2.39e-07 (1,024 trainable)
rank trainable mean validation MSE mean relative update error
1          64          0.04535140          0.709896
2         128          0.02021863          0.471562
4         256          0.00188776          0.142274
6         384          0.00000000          0.000000
8         512          0.00000000          0.000000
largest merged/unmerged difference: 1.43e-06
```

The failure is orderly. Rank 1 leaves relative correction error 0.710; rank 2 leaves 0.472; rank 4 leaves 0.142. Their mean validation MSEs are 0.04535, 0.02022, and 0.001888. Rank 6 and rank 8 recover the target to below 10^{-7} relative correction error. The unrestricted update reaches numerical-zero validation MSE and 2.39×10^{-7} relative correction error. Every frozen base is bitwise intact, and the largest merged–unmerged discrepancy is below 2×10^{-6} .

In this toy 32×32 layer, rank 6 still trains 384 values—37.5% of a full matrix. The point is inspectability, not an impressive compression ratio. Real LoRA savings depend on wide matrices, small ranks, and which layers are targeted. Likewise, the experiment was built from a low-rank shift—a task whose useful correction is not well captured at the chosen rank will retain error no matter how fashionable the adapter is.

17.4. Quantization: put weights on a smaller grid

PEFT restricts which values may change. **Quantization** asks how precisely values must be represented. If a tensor contains N values stored at b bits each, its ideal packed payload is

$$\text{payload} = \frac{Nb}{8} \text{ bytes.} \quad (17.9)$$



Figure 17.4.: LoRA’s rank is a real capacity constraint in a problem constructed to expose it. Five factor initializations converge to the same rank-limited relative update errors (left), closely tracking the analytic best floors from the planted diagonal magnitudes. Validation error (right) falls sharply but remains nonzero below the planted rank six; rank six, rank eight, and the unrestricted full update reach numerical zero. Counts label trainable adapter values in this deliberately small layer.

One billion FP16 weights therefore require 2 GB in decimal units; an ideal packed 4-bit payload requires 0.5 GB. Those are weight-payload lower bounds—not complete checkpoints or working-memory forecasts. Scales, zero-points, unquantized tensors, packing alignment, activations, and caches remain.

This is where a tensor’s *dtype* becomes more than a software label. Appendix B will gather dtype, shape, stride, and physical layout in one place; Appendix C connects FP16, BF16, rounding, and mixed precision to the finite representations first encountered in Chapter 5. Here we need one narrower fact—fewer nominal bits change both representable values and storage arithmetic.

Start with signed **symmetric uniform quantization** of a real-valued tensor $\mathbf{W}$. For b bits, let

$$Q = 2^{b-1} - 1, \quad s = \frac{\max_{ij} |W_{ij}|}{Q}.$$

Quantize and reconstruct by

$$q_{ij} = \text{clip}(\text{round}(W_{ij}/s), -Q, Q), \quad \widehat{W}_{ij} = sq_{ij}. \quad (17.10)$$

s is the distance between adjacent real-valued grid points. This symmetric choice uses codes $-Q, \dots, Q$: $2^b - 1$ levels, leaving the most-negative two’s-complement code unused so the endpoints remain balanced around zero. With the maximum used for calibration, values are not clipped beyond the selected range; nearest rounding gives $|W_{ij} - \widehat{W}_{ij}| \leq s/2$. One extreme value increases s for everything sharing that scale, so small values may all round to zero.

If every value is zero, choose any positive sentinel scale—say $s = 1$ —and store zero codes. A constant affine range likewise needs a separately defined convention; the range formula below otherwise divides by zero.

An **affine** grid also stores an integer zero-point z :

$$q = \text{clip}(\text{round}(w/s) + z, q_{\min}, q_{\max}), \quad \widehat{w} = s(q - z). \quad (17.11)$$

For a nonconstant calibrated interval $[w_{\min}, w_{\max}]$, a standard choice is

$$s = \frac{w_{\max} - w_{\min}}{q_{\max} - q_{\min}}, \quad z = \text{clip}\left(\text{round}\left(q_{\min} - \frac{w_{\min}}{s}\right), q_{\min}, q_{\max}\right).$$

The zero-point lives in integer-code coordinates. Mixing it with a real-valued offset—subtracting z before dividing by s , then adding z after dequantization—is a units error.

17.4.1. One scale can erase a quiet channel

Scale granularity changes the trade—**per-tensor** quantization shares one scale across an entire matrix. **Per-row** quantization gives each output row its own scale; for a linear layer this is often called per-output-channel quantization. More scales cost metadata, but a large row no longer sets the grid spacing for a small row.

The next audit creates a 64×256 weight matrix whose row ranges span 0.01 to 10. It compares 8-bit and 4-bit symmetric grids, either global or per-row, on 4,096 fixed Gaussian inputs. The code reports relative weight error, relative layer-output error, the distribution of row errors, and ideal packed payload plus FP32 scale metadata. It reports no runtime—the simple Python arithmetic is not a quantized production kernel.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `make_quantization_problem`, `symmetric_quantize`, and `run_quantization_audit`.
 3. Compare per-tensor and per-row symmetric quantization, including output error and metadata.
-

CODE

```
# [1]
QUANT_D_IN, QUANT_D_OUT = 256, 64

# [2]
def make_quantization_problem(
    seed: int = 1701,
) -> tuple[torch.Tensor, torch.Tensor]:
    generator = torch.Generator().manual_seed(seed)
    directions = torch.randn(QUANT_D_OUT, QUANT_D_IN, generator=generator)
    directions = directions / directions.abs().amax(dim=1, keepdim=True)
    row_ranges = torch.logspace(-2, 1, QUANT_D_OUT).unsqueeze(1)
```

```

weights = directions * row_ranges
inputs = torch.randn(4096, QUANT_D_IN, generator=generator)
return weights, inputs

def symmetric_quantize(
    weights: torch.Tensor, bits: int, per_row: bool
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    qmax = 2 ** (bits - 1) - 1
    scale = (
        weights.abs().amax(dim=1, keepdim=True) / qmax
        if per_row else weights.abs().amax() / qmax
    )
    scale = torch.where(scale == 0, torch.ones_like(scale), scale)
    codes = torch.clamp(torch.round(weights / scale), -qmax, qmax).to(torch.int8)
    reconstructed = codes.float() * scale
    return codes, reconstructed, scale

@torch.no_grad()
def run_quantization_audit() -> list[dict]:
    weights, inputs = make_quantization_problem()
    reference = inputs @ weights.T
    rows = []
    for bits in (8, 4):
        for per_row in (False, True):
            codes, reconstructed, scale = symmetric_quantize(
                weights, bits, per_row
            )
            row_errors = (
                torch.linalg.norm(reconstructed - weights, dim=1)
                / torch.linalg.norm(weights, dim=1)
            )
            payload_bytes = codes.numel() * bits / 8
            metadata_bytes = scale.numel() * 4
            rows.append({
                "bits": bits,
                "granularity": "per-row" if per_row else "per-tensor",
                "weight_error": (
                    torch.linalg.norm(reconstructed - weights)
                    / torch.linalg.norm(weights)
                ).item(),
                "output_error": (
                    torch.linalg.norm(inputs @ reconstructed.T - reference)
                    / torch.linalg.norm(reference)
                ).item(),
                "row_errors": row_errors.numpy(),
                "row_ranges": weights.abs().amax(dim=1).numpy(),
                "payload_bytes": int(payload_bytes),
            })

```

```

        "metadata_bytes": int(metadata_bytes),
        "total_bytes": int(payload_bytes + metadata_bytes),
    })
    return rows

quant_rows = run_quantization_audit()
print("bits  scheme          w err   out err  med row  max row  payload  meta  total")
# [3]
for row in quant_rows:
    print(
        f"{row['bits']:>4}  {row['granularity']:<10}  "
        f"{row['weight_error']:.4f}      {row['output_error']:.4f}      "
        f"{np.median(row['row_errors']):.4f}      {row['row_errors'].max():.4f}  "
        f"{row['payload_bytes']:>7,}  {row['metadata_bytes']:>8,}  "
        f"{row['total_bytes']:>5,}"
    )

```

bits	scheme	w err	out err	med row	max row	payload	meta	total
8	per-tensor	0.0213	0.0213	0.2213	1.0000	16,384		4
	↪ 16,388							
8	per-row	0.0069	0.0069	0.0067	0.0096	16,384		256
	↪ 16,640							
4	per-tensor	0.2688	0.2695	1.0000	1.0000	8,192		4
	↪ 8,196							
4	per-row	0.1228	0.1223	0.1225	0.1772	8,192		256
	↪ 8,448							

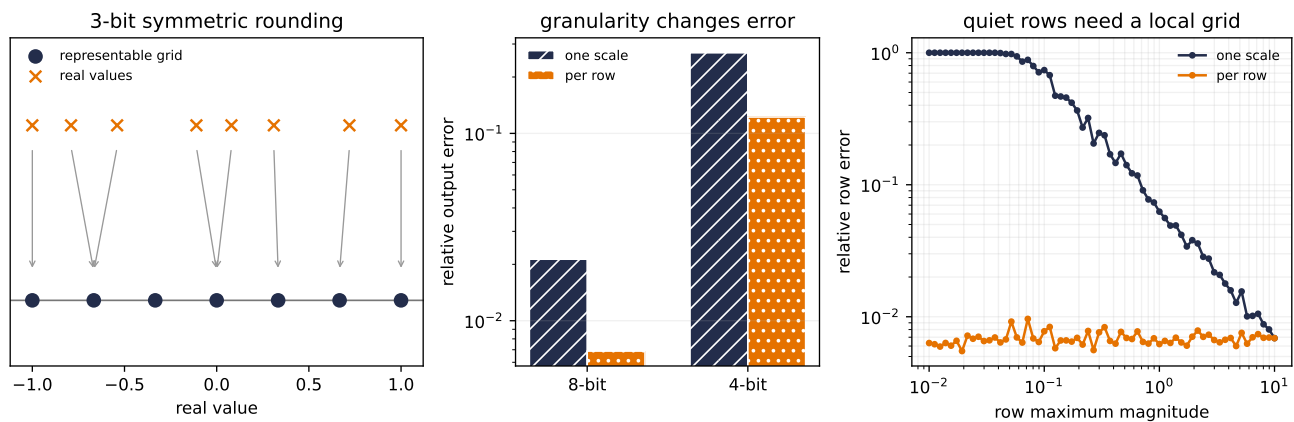


Figure 17.5.: Quantization is a grid plus a granularity decision. A 3-bit symmetric example (left) rounds real values to seven levels. In the unequal-row audit, per-row scales reduce relative layer-output error at both bit widths (center). At 8 bits, the global scale entirely erases the smallest rows—relative error 1—while per-row error stays near 0.7% across ranges (right). The improvement costs 256 rather than 4 bytes of FP32 scale metadata.

The global 8-bit grid has only 2.13% relative output error overall, which sounds reassuring until we inspect rows—the median relative row error is 22.13%, and the quietest rows are reconstructed as all

zeros. Per-row scales lower overall output error to 0.689% and maximum row error below 0.964%. At 4 bits, per-row scales improve output error from 26.95% to 12.23%, but do not make the approximation exact.

The packed 4-bit codes require 8,192 bytes. One FP32 scale brings the global total to 8,196 bytes; 64 row scales bring it to 8,448. The live `torch.int8` tensor in this teaching cell occupies one byte per code—it is **not** physically packed into nibbles. Equation 17.9 is the payload a packing format could achieve.

⚠ A smaller checkpoint is not automatically faster

Bit count does not determine latency by itself. Real speed depends on supported kernels, hardware, packing, dequantization, memory bandwidth, batch shape, and which tensors remain in higher precision. Dettmers and colleagues even reported cases where `LLM.int8()` overhead made smaller models slower than FP16. Measure the target system; do not turn a file-size ratio into a runtime claim. The Roofline and measurement contracts in Appendix C make that distinction explicit.

17.4.2. What a quantization workflow protects

Post-training quantization (PTQ) chooses a representation after the base model has trained. It may use calibration inputs, but it does not run ordinary end-to-end training again. Our uniform audit used only weight ranges. More sophisticated PTQ methods protect behavior measured on calibration data. For a layer $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and n calibration activations $\mathbf{X} \in \mathbb{R}^{d_{\text{in}} \times n}$, GPTQ frames a layerwise weight-only problem as

$$\widehat{\mathbf{W}} = \arg \min_{\widetilde{\mathbf{W}} \in \mathcal{Q}} \|\mathbf{W}\mathbf{X} - \widetilde{\mathbf{W}}\mathbf{X}\|_F^2, \quad (17.12)$$

where $\widetilde{\mathbf{W}}$ has the same shape as $\mathbf{W}$ and $\mathcal{Q}$ is the chosen quantized set. The objective protects layer outputs—not ordinary weight MSE and not the original task loss. AWQ uses activation evidence to choose channel scaling that protects salient weights. These methods make the calibration distribution part of the result.

Quantization-aware training (QAT) instead inserts fake quantize–dequantize operations into the forward pass while retaining higher-precision shadow weights for optimization. Because rounding is nondifferentiable, training uses a surrogate gradient such as a straight-through estimator. QAT is not integer-only training; conversion to integer execution remains a separate, kernel-dependent step.

Weight-only and weight–activation quantization also answer different bottlenecks. A weight-only method leaves activations and the autoregressive key/value cache at their chosen compute precision. `LLM.int8()`, by contrast, quantizes both matrix inputs and weights while routing a small activation-outlier subspace through FP16 in its studied regime. “8-bit method” is not a complete algorithm description.

17.4.3. QLoRA: compose a frozen grid with a trainable delta

QLoRA places the two major levers side by side. Store a pretrained base in blockwise 4-bit **NormalFloat (NF4)**, dequantize blocks to BF16 for matrix multiplication, and train higher-precision LoRA factors:

$$\mathbf{h} = \text{Dequant}_{\text{NF4}}(\mathbf{W}_q)\mathbf{x} + \frac{\alpha}{r}\mathbf{B}(\mathbf{A}\mathbf{x}). \quad (17.13)$$

The base codes $\mathbf{W}_q$ remain frozen. Gradient blame passes through the dequantized computation to the LoRA path, but ordinary updates do not rewrite the 4-bit base. QLoRA is therefore not QAT of the backbone, and “4-bit storage” does not mean every multiply or accumulator is 4-bit.

NF4 uses a nonuniform codebook with denser resolution where a standardized normal distribution places more probability mass, plus an exact zero. Its information argument applies to block-normalized, approximately zero-centered normal weights; it is not a universal optimal grid for arbitrary tensors.

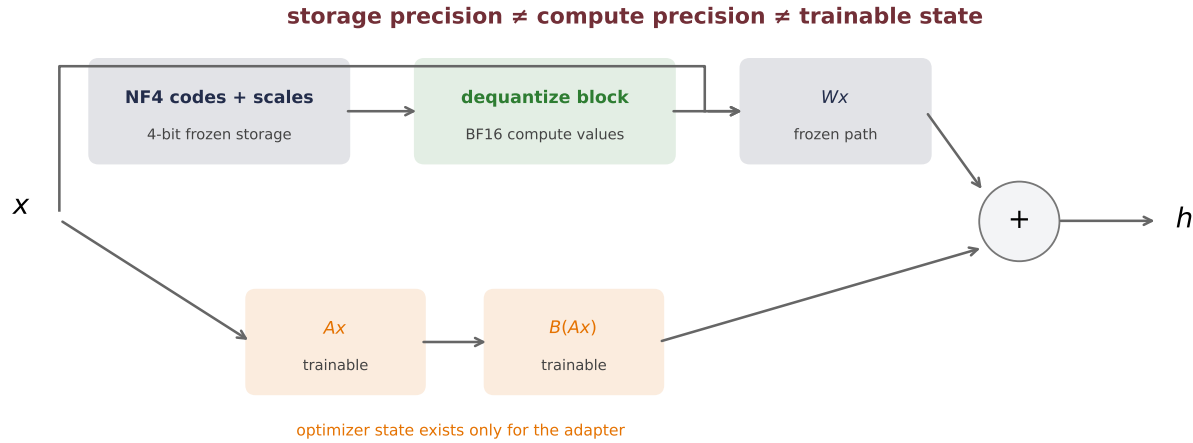


Figure 17.6.: QLoRA composes two axes without conflating them. A frozen blockwise NF4 base is dequantized to the compute dtype for the main matrix multiplication. Higher-precision A and B factors form a trainable low-rank branch, and only that branch has optimizer state. The sum defines the adapted layer; storage dtype, compute dtype, and trainable dtype are separate choices.

QLoRA also uses **double quantization**—it quantizes its quantization constants. With the paper’s illustrative group sizes, FP32 constants for each 64-weight block cost $32/64 = 0.5$ bits per weight. Storing those constants at 8 bits and adding one FP32 second-level constant per 256 first-level constants costs

$$\frac{8}{64} + \frac{32}{64 \cdot 256} = 0.126953 \quad \text{bits per weight,}$$

a saving of 0.373047 bits per weight. That number belongs to those block sizes; it is not a universal number of gigabytes. Paged optimizers address transient memory spikes in the paper’s implementation, while the LoRA factors still carry their own gradients and optimizer state.

i QLoRA is a composition, not a new objective

NF4 decides how the frozen foundation is stored; LoRA decides where task gradients may write. The supervised, preference, or other loss still decides what behavior is rewarded. A parameter-efficient route can optimize a poor objective just as efficiently as a good one.

17.5. Choose by the bill you need to reduce

Method	Task state and permitted writes	Main consequence
Hard prompt / ICL	State in ordinary context; no gradients during use	No learned task checkpoint; longer context
RAG	State in index and retrieved context; writes depend on design	Fresher evidence; retrieval failure surface
Prompt tuning	State in continuous input vectors; write prompt vectors, plus any deliberately unfrozen head	Small learned state; virtual positions remain
Prefix / adapters	State in per-layer prefixes or bottlenecks; write added modules	Small checkpoint; extra inference path
LoRA	State in A , B ; write factors, plus any deliberately unfrozen head	Conditionally mergeable; rank bottleneck
Full fine-tuning	State in a changed backbone; write all permitted parameters	Maximum update freedom; one full task copy
PTQ	No task state—representation only; no gradients in ordinary PTQ	Smaller weight representation; approximation error
QLoRA	Task state in LoRA factors; codes/scales describe the frozen base; write factors plus any deliberately unfrozen head	Smaller base representation during PEFT; mixed dtypes

The instructor’s practical instinct remains useful: when a carefully evaluated prompt or retrieval pipeline solves the task, avoid fine-tuning merely because it is available. Treat that as a search heuristic—not a law. A stable format change may fit a small adapter; genuinely new knowledge may belong in retrieval or further training; a severe source-coverage mismatch may defeat all of them.

That last point harvests Chapter 9’s full promise. PEFT changes the cost **after** we choose transfer; it does not repeal the coverage condition. A frozen foundation cannot guarantee a reliable representation absent from source coverage, and a low-rank update does not guarantee that scarce target labels can build the missing feature. Quantization preserves or damages an existing computation—it does not add coverage.

Trainable parameters are not total memory

During PEFT, the frozen base still occupies memory and backpropagation still stores or recomputes activations through it. During generation, context and key/value caches can dominate different regimes. Optimizer dtype and implementation change the bill. Report trainable state, base storage, transient training memory, context length, and measured runtime separately. One percentage cannot stand in for all five.

Parameter efficiency also does not guarantee immunity to forgetting or harmful behavior. Restricting an update constrains which coordinates or directions are available—it does not bound the update’s magnitude or certify what the movement preserves. Evaluation must include the target task, retained capabilities that matter, calibration or distribution shifts, and the actual deployment representation.

17.6. The next question: what deserves the update?

Where the update lives is not what the update optimizes. Prompt vectors, low-rank matrices, and full fine-tuning decide where task gradients may write. Quantization decides how values are stored. None decides which answer is helpful, harmless, honest, or preferred.

Chapter 18 will keep the adaptation machinery and change the question from *which parameters move?* to *which objective should move them?* Instruction tuning, preference learning, and alignment methods are claims about supervision and objectives—not automatic consequences of choosing LoRA or a full checkpoint.

i Check yourself

Close the book for one minute and separate the adaptation bills.

- Where can task-specific state live when the backbone is frozen?
- What does LoRA’s rank constrain, and what does quantization change instead?
- Why can a method reduce stored bits without reducing every inference cost?

17.7. Okay, so — adaptation has three separate bills

1. **Adaptation has three bills.** Backbone storage, incremental task state, and transient work must be reported separately. Zero trainable weights and fewer stored bits do not mean zero inference or training cost.
2. **Prompting changes context, not weights.** The five-seed permutation study closely tracks the analytical information ceilings while every evaluation weight stays bitwise fixed. RAG is another context lever, with a separate retrieval failure surface.
3. **PEFT restricts where gradient blame may write.** Prompt tuning, prefixes, adapters, BitFit, and LoRA impose different parameterizations; they are not interchangeable labels for “small fine-tuning.”
4. **LoRA learns a low-rank correction.** Its trainable count is $r(d_{\text{in}} + d_{\text{out}})$ per targeted matrix, and its rank is a real capacity limit. The planted rank-six experiment fails exactly where it should.
5. **Quantization is a grid, a granularity, and a workflow.** Per-row scales rescue quiet rows in the controlled audit, but consume metadata. Payload size does not predict runtime without kernels and hardware.
6. **QLoRA composes representation and adaptation.** A frozen NF4 base is dequantized for compute while higher-precision LoRA factors train. Storage dtype, compute dtype, and trainable dtype are distinct.
7. **Transfer coverage still governs the choice.** None of these methods creates source knowledge for free. The next chapter will ask what the permitted update should optimize.

Sources and further reading

- [Brown et al., *Language Models are Few-Shot Learners*](#): GPT-3’s zero-, one-, and few-shot evaluation and its model-scale dependence.

- Min et al., *Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?*: controlled ablations of demonstration inputs, labels, and formatting that qualify simple imitation accounts.
- Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*: learned retrieval combined with a pretrained generator.
- Houlsby et al., *Parameter-Efficient Transfer Learning for NLP*: bottleneck adapters inserted into a frozen Transformer.
- Li and Liang, *Prefix-Tuning: Optimizing Continuous Prompts for Generation*: trainable per-layer prefixes for generation.
- Lester et al., *The Power of Scale for Parameter-Efficient Prompt Tuning*: input-level soft prompts and the scoped T5 scale study.
- Ben-Zaken et al., *BitFit: Simple Parameter-Efficient Fine-Tuning for Transformer-Based Masked Language-Models*: bias-term updates plus the task-specific classifier in the original experiments.
- Hu et al., *LoRA: Low-Rank Adaptation of Large Language Models*: the low-rank update parameterization, initialization, and merge.
- Wei et al., *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*: scoped gains from prompting with intermediate reasoning steps.
- Turpin et al., *Language Models Don't Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting*: controlled evidence that visible explanations can omit causal influences on predictions.
- Greshake et al., *Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection*: the attack surface created when untrusted retrieved data can act as instructions.
- Jacob et al., *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*: affine quantization and the fake-quantization training pattern.
- Dettmers et al., *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*: mixed-precision decomposition for activation outliers—and measured overhead caveats.
- Frantar et al., *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*: calibration-output-aware, layerwise weight-only PTQ.
- Lin et al., *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*: activation evidence for protecting salient weights.
- Dettmers et al., *QLoRA: Efficient Finetuning of Quantized LLMs*: NF4, double quantization, paged optimizers, and LoRA over a frozen quantized base.

Exercises

1. **(Pencil.)** A model has 24 matrices of shape 4096×4096 . LoRA targets each with rank 16. Compute full and adapter value counts, the adapter fraction, and FP16 adapter bytes. Then compute the ideal 4-bit base payload. Add one FP32 scale per block of 64 weights and explain which deployment costs remain absent.
2. **(Code.)** Reproduce the frozen-context experiment, then randomize demonstration order, repeat one group with the same label, and finally insert a contradictory label for a repeated group. Predeclare which interventions preserve Equation 17.2 and which make it inapplicable. Report all seeds and verify bitwise parameter identity before interpreting a difference.
3. **(Pencil.)** (a) Prove $\text{rank}(\mathbf{BA}) \leq r$. **(Code.)** (b) Extend the rank-capacity audit with a planted rank-10 correction and ranks 2, 6, 10, 14. Audit gradients after the first backward pass for (a) random $\mathbf{A}$ and zero $\mathbf{B}$, and (b) both factors zero. Verify merged–unmerged equality.

4. **(Pencil.)** (a) Quantize $[-1.2, -0.3, 0.0, 0.5, 1.7]$ onto a signed 4-bit symmetric grid, showing Q , s , q , and $\hat{w}$. Then construct an affine 8-bit grid for a nonnegative interval. **(Code.)** (b) On the chapter's unequal-row matrix, compare calibration and held-out output error after inserting one new outlier row.
5. **(Audit.)** For each of three cases—fresh facts with citations, a stable output format, and a target domain absent from the source data—choose a first experiment among prompting/RAG, PEFT, full tuning, and quantization. State where task state lives, which bill it reduces, what it cannot repair, and one retained behavior you would evaluate. Finally explain why none of those choices answers the Chapter 18 question of which behavior deserves reward.

18. Alignment and RL Fine-Tuning

Chapter 17 ended with a distinction worth keeping in view—**where the update lives is not what the update optimizes**. Full fine-tuning, LoRA, and a soft prompt choose different places for gradient blame to write. None of them says what should count as a good answer.

This chapter changes that second choice. Pretraining asks for the next corpus token. Instruction tuning asks a model to imitate demonstrated responses. Preference learning asks which response wins a comparison. Online policy optimization asks a changing model to produce responses that score well under a reward signal. These supervision interfaces can often be paired with different update locations:

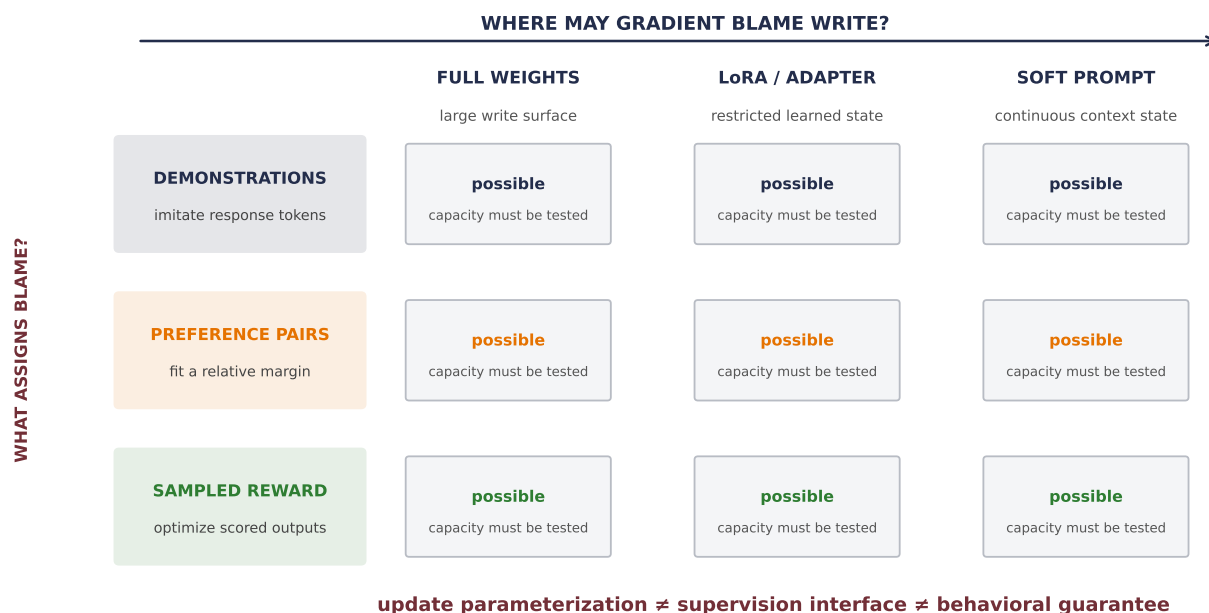


Figure 18.1.: The objective and the update location are separate design axes. Demonstration loss, a preference margin, or sampled reward can assign gradient blame while full weights, a low-rank branch, or continuous prompt vectors determine where that blame may write. The grid marks possible combinations, not a claim that they have equal capacity or cost.

The word **alignment** is sometimes used as though it named one algorithm or one finished property—here it will mean something narrower and checkable: changing a model’s behavior toward a stated objective, for a stated prompt distribution and a stated population of evaluators. Helpful, honest, and harmless are useful goals from the intended contract. They are not certificates that training can stamp onto a model.

The central problem is therefore a measurement problem. We need to specify whose judgment is collected, under which rubric, over which candidate responses, and what happens when optimization reaches beyond that coverage. Only then does it make sense to compare instruction tuning, reward-model pipelines, and direct preference optimization.

18.1. Same model, different supervision

Recall the causal language model from [Chapter 14](#). Pretraining scores every corpus token that the model is asked to predict. In a prompt–response record, however, the prompt usually supplies the condition—the response supplies the supervised target. Let

$$z = (x_1, \dots, x_P, y_1, \dots, y_T)$$

be a serialized prompt x followed by response y . Define $m_t = 1$ when token z_t belongs to the response (including its end token) and $m_t = 0$ for prompt or padding positions. A response-masked supervised fine-tuning objective is

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(x,y) \sim \mathcal{D}_{\text{demo}}} \left[\frac{1}{\sum_t m_t} \sum_{t=2}^{P+T} m_t \log \pi_{\theta}(z_t \mid z_{<t}) \right]. \quad (18.1)$$

The denominator makes the displayed version a mean per scored response token. Some implementations sum instead—the choice changes gradient scale and must be stated. This is the book’s founding instance of the audit habit from [Chapter 4](#): Case 1 within the mask, with the denominator itself part of the protocol. The prompt is still essential because it conditions every response probability. It simply does not receive target loss in this version.

The tensor contract is explicit: `logits` has shape (B, L, V) , token IDs and the mask have shape (B, L) , and the returned sequence score has shape $(B,)$. The helper below audits that excluded prompt and padding targets cannot change that score.

PLAN

1. Define the reusable `completion_log_probability` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Audit completion-only sequence log probabilities.
-

CODE

```
# [1]
def completion_log_probability(
    logits: Tensor, token_ids: Tensor, response_mask: Tensor
) -> Tensor:
    """Sum next-token log probabilities only at response positions."""
    if logits.shape[:2] != token_ids.shape or token_ids.shape != response_mask.shape:
        raise ValueError("logits, token IDs, and mask must share batch and time")
    next_token_logps = F.log_softmax(logits[:, :-1], dim=-1).gather(
        dim=-1, index=token_ids[:, 1:].unsqueeze(-1)
    ).squeeze(-1)
    return (next_token_logps * response_mask[:, 1:]).sum(dim=1)
```

```

# [2]
token_ids = torch.tensor([
    [1, 2, 3, 4, 5, 6, 0],
    [1, 2, 7, 8, 9, 6, 0],
])
response_mask = torch.tensor([
    [False, False, False, True, True, True, False],
    [False, False, True, True, True, True, False],
])
toy_logits = torch.randn(2, 7, 11)
original_logps = completion_log_probability(toy_logits, token_ids, response_mask)

perturbed_logits = toy_logits.clone()
inactive_predictors = ~response_mask[:, 1:]
perturbed_logits[:, :-1][inactive_predictors] = torch.linspace(-5, 5, 11)
perturbed_logps = completion_log_probability(
    perturbed_logits, token_ids, response_mask
)

# [3]
print("scored response tokens:", response_mask[:, 1:].sum(dim=1).tolist())
print(
    "largest change after perturbing prompt/padding predictions:",
    f"{{(perturbed_logps - original_logps).abs().max().item():.2e}}",
)

```

```

scored response tokens: [3, 4]
largest change after perturbing prompt/padding predictions: 0.00e+00

```

Instruction tuning is supervised fine-tuning on one or more tasks expressed through natural-language instructions. It changes the organization of the data—not the optimizer. FLAN, for example, studied whether instruction tuning across many tasks improved transfer to held-out tasks. That is an empirical result about its models and scale—not a theorem that every instruction dataset generalizes.

A useful shorthand says SFT teaches format while preferences teach quality, a distinction that is too sharp. A carefully written demonstration can teach facts, style, safety behavior, and substantive quality. Equation 18.1 says what SFT actually guarantees: it rewards probability assigned to demonstrated response tokens. It does not directly say why one unobserved response should be preferred to another.

Comparisons offer a different interface. For one prompt, a reviewer can sometimes choose between two candidates more reliably than they can write an ideal response from scratch. That generation–evaluation gap motivates preference data. It does not make comparison cheap in every domain, remove the need for expertise, or turn a judgment into universal truth.

18.2. A preference is a measurement, not a value

One preference record contains a prompt x , a response judged better y_w , and a response judged worse y_l :

$$(x, y_w, y_l) \in \mathcal{D}_{\text{pref}}.$$

The subscript w means *winner under this comparison*—not that the response is absolutely correct, harmless, or good. The label depends on the rubric, the reviewer population, the order in which candidates were displayed, and the policy that generated the candidates. Ties and disagreement are data, too; silently forcing them into a total order hides uncertainty.

To turn comparisons into a trainable scalar score, many pipelines use the **Bradley–Terry model**. A reward model $r_\phi(x, y) \in \mathbb{R}$ assigns a score to each completed response. The probability of preferring one response depends on the score difference:

$$P_\phi(y_w \succ y_l \mid x) = \sigma(r_\phi(x, y_w) - r_\phi(x, y_l)). \quad (18.2)$$

This is the sigmoid from [Chapter 2](#). The corresponding negative log-likelihood is ordinary binary cross-entropy on a difference of scores:

$$\mathcal{L}_{\text{RM}}(\phi) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}_{\text{pref}}} \log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l)). \quad (18.3)$$

Only differences are identified. Adding any prompt-dependent constant $c(x)$ to every candidate reward leaves Equation 18.2 unchanged. The model learns a ruler—one with no identifiable zero point.

There is a deeper assumption: all comparisons for one prompt must be representable by one scalar ordering. Let $\mathbf{r} = (r_A, r_B, r_C)^\top \in \mathbb{R}^3$. Let the rows of $\mathbf{B} \in \mathbb{R}^{3 \times 3}$ encode $A - B$, $B - C$, and $C - A$. Then $\mathbf{B}\mathbf{r}$ contains their three score differences. A transitive set of probabilities can come from a score vector. A cycle that prefers each response to the next cannot:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Make the scalar-reward assumption fail on a preference cycle.
-

CODE

```
# [1]
incidence = torch.tensor([
    [1.0, -1.0, 0.0],
    [0.0, 1.0, -1.0],
    [-1.0, 0.0, 1.0],
])

transitive_scores = torch.tensor([1.0, 0.0, -1.0])
transitive_targets = torch.sigmoid(incidence @ transitive_scores)
fitted_transitive = torch.linalg.lstsq(
    incidence, torch.logit(transitive_targets)
).solution
fitted_transitive -= fitted_transitive.mean()
```



```

cyclic_targets = torch.full((3,), 0.70)
fitted_cyclic = torch.linalg.lstsq(
    incidence, torch.logit(cyclic_targets)
).solution
fitted_cyclic -= fitted_cyclic.mean()
cyclic_predictions = torch.sigmoid(incidence @ fitted_cyclic)
scalar_loss = F.binary_cross_entropy(cyclic_predictions, cyclic_targets)
edgewise_floor = F.binary_cross_entropy(cyclic_targets, cyclic_targets)

# [2]
shifted_probabilities = torch.sigmoid(incidence @ (fitted_transitive + 37.0))
print("recovered centered scores:", [round(x, 6) for x in
    ↪ fitted_transitive.tolist()])
print("cyclic scalar predictions:", [round(x, 6) for x in
    ↪ cyclic_predictions.tolist()])
print(f"cyclic scalar loss: {scalar_loss.item():.6f}")
print(f"unconstrained edgewise floor: {edgewise_floor.item():.6f}")
print(f"irreducible scalar gap: {(scalar_loss - edgewise_floor).item():.6f}")
print(
    "largest probability change after adding 37 to every reward:",
    f"{(shifted_probabilities - transitive_targets).abs().max().item():.2e}",
)

```

```

recovered centered scores: [1.0, -0.0, -1.0]
cyclic scalar predictions: [0.5, 0.5, 0.5]
cyclic scalar loss: 0.693147
unconstrained edgewise floor: 0.610864
irreducible scalar gap: 0.082283
largest probability change after adding 37 to every reward: 0.00e+00

```

The cycle is not a pathology in the code. Summing the three Bradley–Terry logits would require

$$(r_A - r_B) + (r_B - r_C) + (r_C - r_A) = 3 \logit(0.70),$$

but the left side cancels to zero while the right side is positive. Aggregated human judgments can contain cycles because reviewers differ, context changes, or preferences are genuinely plural. A scalar reward model compresses that structure; it does not discover a hidden universal value.

 **Trap:** a preferred response is not scalar ground truth

A pair says which presented response won under one prompt, rubric, candidate set, and review process. Report ties and disagreement, split by prompt rather than by correlated pairs, and state who supplied the judgments. Bradley–Terry is a useful model of those measurements, not proof that one population-independent reward exists.

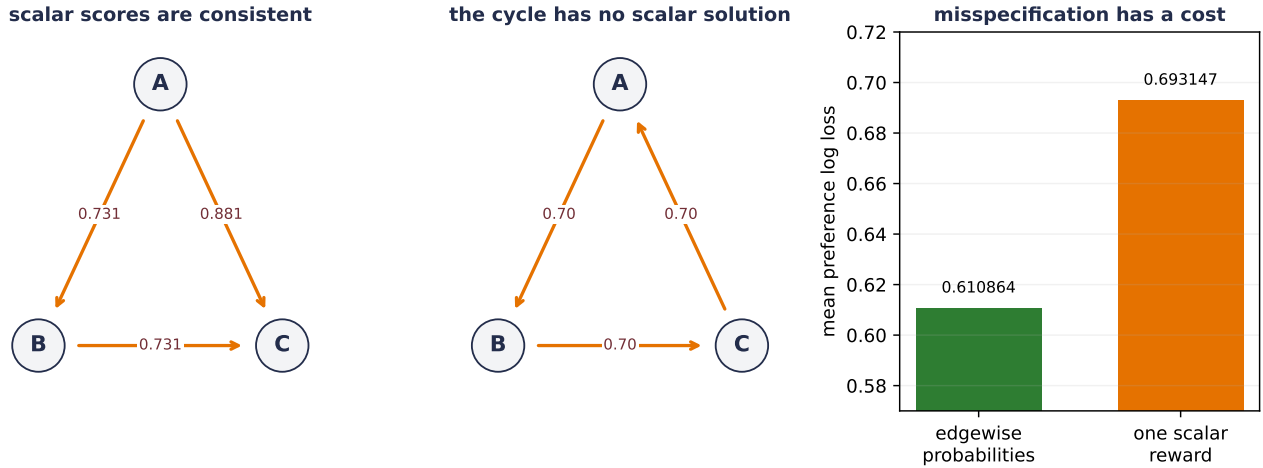


Figure 18.2.: A scalar reward can express the transitive comparisons generated by centered scores (1, 0, -1), but not a symmetric cycle in which A beats B, B beats C, and C beats A with probability 0.70. Least squares returns equal scalar scores for the cycle, so Bradley–Terry predicts 0.50 on every edge. Its mean log loss is $\log 2 = 0.693147$, above the unconstrained per-edge entropy 0.610864 by 0.082283.

18.3. Train a judge, then try to please the judge

What can we do with pairwise measurements? The classic language-model RLHF pipeline separates two learned objects:

1. a **reward model** learns to score completed responses from comparisons; and
2. a **policy** learns to generate responses that receive high score.

This is the course’s memorable description—*train a judge, then try to please the judge*. The word *judge* matters. A reward model is a learned proxy for a feedback process, not the feedback process itself.

In language generation, the **state** s_t is the prompt plus tokens generated so far, the **action** a_t is the next token, and the policy is $\pi_\theta(a_t | s_t)$. A reward model often scores the completed response. A separate value function $V_\psi(s_t)$ estimates expected future shaped return from a prefix so that policy-gradient updates can assign credit across token decisions. Calling the reward model PPO’s *critic* conflates those two roles—completed-response scoring and prefix-value forecasting.

For a fixed prompt, suppose there are K complete candidate responses. Let $\mathbf{r} \in \mathbb{R}^K$ contain their reward-model scores, and let $\boldsymbol{\pi}, \boldsymbol{\pi}_{\text{ref}} \in \mathbb{R}^K$ be positive probability vectors that each sum to one. A clean regularized objective is

$$J(\boldsymbol{\pi}) = \sum_{i=1}^K \pi_i r_i - \beta D_{\text{KL}}(\boldsymbol{\pi} \| \boldsymbol{\pi}_{\text{ref}}), \quad \beta > 0, \quad (18.4)$$

where

$$D_{\text{KL}}(\boldsymbol{\pi} \| \boldsymbol{\pi}_{\text{ref}}) = \sum_{i=1}^K \pi_i \log \frac{\pi_i}{\pi_{\text{ref},i}}.$$

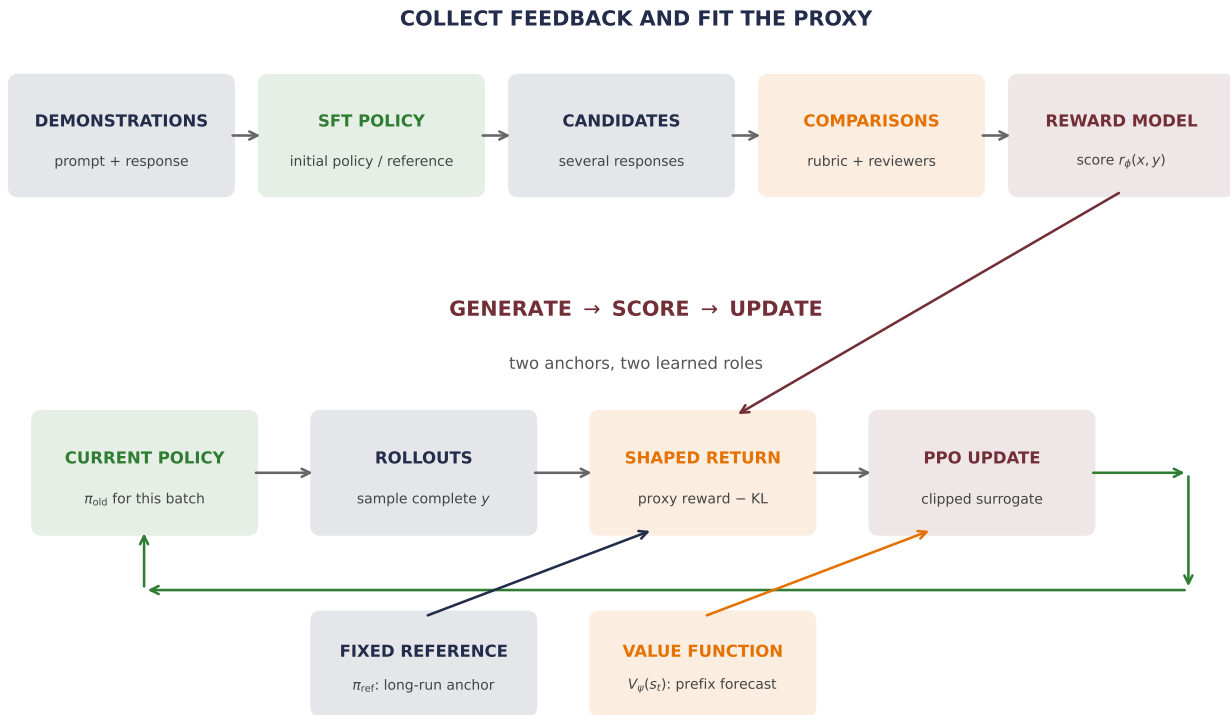


Figure 18.3.: An original map of a reward-model RLHF pipeline. Demonstrations produce an SFT policy; sampled candidates and comparisons train a reward model. During online policy optimization, the current rollout policy produces responses, the frozen reward model scores them, and a fixed reference policy supplies a separate KL anchor. A value function estimates future shaped return from prefixes; it is not the reward model. PPO’s old-policy anchor is refreshed with rollout batches, while the reference anchor usually remains fixed.

The first term seeks high proxy reward. The second charges for moving probability mass away from the reference. This is an objective to **maximize**—not a loss unless we negate it.

18.3.1. The exact finite-response policy

Before discussing an optimizer, let us solve the finite problem. Add a Lagrange multiplier λ for $\sum_i \pi_i = 1$. Differentiating with respect to one probability gives

$$r_i - \beta \left(\log \frac{\pi_i}{\pi_{\text{ref},i}} + 1 \right) + \lambda = 0.$$

Rearrange and absorb terms independent of i into a normalizer:

$$\pi^*(y_i | x) = \frac{\pi_{\text{ref}}(y_i | x) \exp(r(x, y_i)/\beta)}{Z(x)}, \quad Z(x) = \sum_{j=1}^K \pi_{\text{ref}}(y_j | x) \exp(r(x, y_j)/\beta). \quad (18.5)$$

Equation 18.5 is a reference distribution tilted toward higher reward. Large β resists movement; small β applies more reward pressure. It is a stay-near dial—not a truth or safety guarantee.

Consider four possible responses with

$$\pi_{\text{ref}} = (0.55, 0.25, 0.15, 0.05), \quad \mathbf{r} = (0, 1, 2, 3).$$

The reference favors response A while the proxy favors D. The exact solution lets us watch probability move without sampling noise or a language model:

PLAN

1. Define the reusable helpers: `gibbs_policy` and `kl_divergence`.
 2. Prepare the inputs and fixed settings for the example.
 3. Solve the KL-regularized finite-response policy exactly.
-

CODE

```
# [1]
def gibbs_policy(reference: Tensor, reward: Tensor, beta: float) -> Tensor:
    return torch.softmax(torch.log(reference) + reward / beta, dim=0)

def kl_divergence(policy: Tensor, reference: Tensor) -> Tensor:
    return (policy * (torch.log(policy) - torch.log(reference))).sum()

# [2]
reference = torch.tensor([0.55, 0.25, 0.15, 0.05])
```

```

reward = torch.tensor([0.0, 1.0, 2.0, 3.0])
shown_betas = [4.0, 1.0, 0.5]
shown_policies = [gibbs_policy(reference, reward, beta) for beta in shown_betas]

# [3]
sweep_betas = torch.logspace(math.log10(8.0), math.log10(0.2), 80)
sweep_policies = torch.stack([
    gibbs_policy(reference, reward, beta.item()) for beta in sweep_betas
])
sweep_rewards = sweep_policies @ reward
sweep_kls = torch.stack([
    kl_divergence(policy, reference) for policy in sweep_policies
])

target_policy = gibbs_policy(reference, reward, 1.0)
shifted_target = gibbs_policy(reference, reward + 37.0, 1.0)
print("beta expected reward KL to reference")
for beta in [4.0, 2.0, 1.0, 0.5, 0.25]:
    policy = gibbs_policy(reference, reward, beta)
    print(
        f"{beta:4.3g}      {(policy @ reward).item():.6f}"
        f"              {kl_divergence(policy, reference).item():.6f}"
    )
print("beta=1 policy:", [round(x, 6) for x in target_policy.tolist()])
print(
    "largest policy change after adding 37 to every reward:",
    f"{(shifted_target - target_policy).abs().max().item():.2e}",
)

```

```

beta  expected reward  KL to reference
  4      0.925670      0.029159
  2      1.191913      0.129706
  1      1.768029      0.561398
  0.5     2.559982      1.693801
  0.25    2.944634      2.719127
beta=1 policy: [0.164562, 0.20333, 0.331625, 0.300483]
largest policy change after adding 37 to every reward: 8.33e-16

```

The additive-reward audit echoes the preference experiment: multiplying every unnormalized policy weight by the same $e^{37/\beta}$ changes $Z(x)$ by that factor and leaves the normalized policy unchanged. Neither pairwise labels nor the regularized optimum identifies an absolute reward origin.

18.3.2. PPO in outline: a second anchor

Equation 18.5 optimizes directly over a finite probability vector—a language model instead has shared parameters, an enormous response space, and rewards that arrive mainly after a sequence is complete. Policy-gradient methods estimate how sampled token choices affected return. **Proximal Policy Optimization (PPO)** reuses a batch of trajectories while limiting the incentive for each sampled action ratio to move farther than a clipped interval.

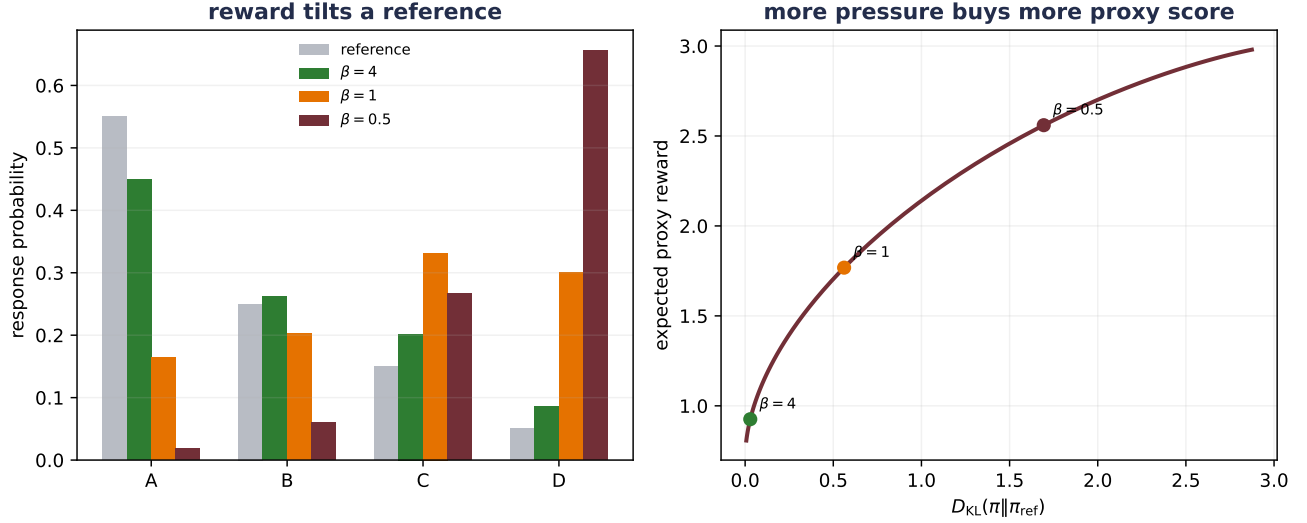


Figure 18.4.: The exact four-response policy tilts the reference toward proxy reward. At beta 4, expected reward is 0.925670 and KL is 0.029159; at beta 1 they are 1.768029 and KL is 0.561398; at beta 0.5 they are 2.559982 and KL is 1.693801. Smaller beta purchases more proxy reward with more measured drift. This identity test assumes a complete finite response set and a correct scalar reward.

For state s_t , sampled action a_t , current parameters θ , and the policy π_{old} that generated the batch, define

$$\rho_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{old}(a_t | s_t)}.$$

If $\hat{A}_t$ estimates whether that sampled action did better or worse than the value baseline, PPO maximizes the clipped surrogate

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(\rho_t(\theta) \hat{A}_t, \text{clip}(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]. \quad (18.6)$$

This gives us two anchors—each with a different job:

- π_{old} generated the current rollout batch. PPO refreshes it as training advances, and Equation 18.6 addresses stepwise optimization.
- π_{ref} is usually a fixed SFT policy. The KL term in Equation 18.4 addresses accumulated drift from that reference.

⚠ The old policy is not the reference policy

PPO clipping is not a hard wall on the new policy—it removes favorable incentive for sampled ratios to move farther outside the interval; a sufficiently large optimizer step can still overshoot, and unsampled behavior can change. The fixed-reference KL discourages drift, but neither mechanism certifies retained capability or prevents proxy exploitation. Measure the realized KL and the behaviors that matter.

18.4. When the proxy becomes the target

The pipeline now has a vulnerable handoff—the reward model is validated on comparisons, but the policy is optimized to seek responses with high predicted reward. Those are different distributions—if the policy finds a region that the comparison data did not identify, excellent held-out reward-model loss can coexist with poor policy behavior.

i A designed mechanism study

Evidence and length are planted features, the finite candidate set is fully known—its designed utility is available because we wrote it. The study demonstrates how coverage failure and proxy pressure can interact. It does not estimate the prevalence of reward hacking in natural-language systems or validate one alignment method over another.

Let us make that failure visible in a finite response world — and predict first: as reward pressure grows, will the narrow-feedback policy’s *designed* utility rise, plateau, or rise-then-collapse? Each synthetic response has two observable features:

- evidence e , where larger means that more relevant support is present; and
- length ℓ , a stand-in for surface polish or detail.

The designed utility is

$$u(e, \ell) = e + 0.6\ell - 1.2 \max(\ell - 1, 0)^2. \quad (18.7)$$

Inside the feedback range $0 \leq \ell \leq 1$, the last feature is always zero. More detail helps there, so comparisons cannot reveal that excessive length should eventually become costly. We generate 20,000 noisy Bradley–Terry comparisons per seed and fit the linear reward model

$$\hat{r}(e, \ell) = w_e e + w_\ell \ell + w_q \max(\ell - 1, 0)^2.$$

Five complete runs use seeds 6050–6054. A **narrow-feedback** arm sees only $\ell \in [0, 1]$. A **coverage-repair** arm draws 20% of individual comparison responses from $\ell \in [1, 4]$, where the missing curvature becomes observable. Both models have the same feature class and optimizer:

PLAN

1. Define the reusable helpers: `reward_features`, `designed_utility`, and `draw_responses`.
 2. Define the reusable helpers: `fit_reward_model` and `mean_and_seed_sd`.
 3. Prepare the inputs and fixed settings for the example.
 4. Fit five reward models with and without the missing feedback region.
 5. Report or visualize the measured result.
-

CODE

```

# [1]
def reward_features(responses: Tensor) -> Tensor:
    evidence = responses[:, 0]
    length = responses[:, 1]
    return torch.stack(
        (evidence, length, torch.relu(length - 1.0).square()), dim=1
    )

def designed_utility(responses: Tensor) -> Tensor:
    designed_weights = torch.tensor([1.0, 0.6, -1.2])
    return reward_features(responses) @ designed_weights

def draw_responses(
    count: int, generator: torch.Generator, out_of_range_fraction: float
) -> Tensor:
    evidence = torch.rand(count, generator=generator)
    length = torch.rand(count, generator=generator)
    if out_of_range_fraction > 0.0:
        outside = torch.rand(count, generator=generator) < out_of_range_fraction
        extended_length = 1.0 + 3.0 * torch.rand(count, generator=generator)
        length = torch.where(outside, extended_length, length)
    return torch.stack((evidence, length), dim=1)

# [2]
def fit_reward_model(
    seed: int, out_of_range_fraction: float
) -> tuple[Tensor, dict[str, float]]:
    generator = torch.Generator().manual_seed(seed)
    count = 20_000
    left = draw_responses(count, generator, out_of_range_fraction)
    right = draw_responses(count, generator, out_of_range_fraction)
    feature_differences = reward_features(left) - reward_features(right)
    preference_probabilities = torch.sigmoid(
        designed_utility(left) - designed_utility(right)
    )
    labels = torch.bernoulli(preference_probabilities, generator=generator)

    weights = nn.Parameter(torch.zeros(3))
    optimizer = torch.optim.Adam([weights], lr=0.05)
    for _ in range(900):
        logits = feature_differences @ weights
        loss = F.binary_cross_entropy_with_logits(logits, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```



```

held_left = draw_responses(10_000, generator, 0.0)
held_right = draw_responses(10_000, generator, 0.0)
held_differences = reward_features(held_left) - reward_features(held_right)
held_probabilities = torch.sigmoid(
    designed_utility(held_left) - designed_utility(held_right)
)
held_labels = torch.bernoulli(held_probabilities, generator=generator)
held_logits = held_differences @ weights.detach()
held_nll = F.binary_cross_entropy_with_logits(held_logits, held_labels)
oracle_nll = F.binary_cross_entropy(held_probabilities, held_labels)
order_accuracy = (
    torch.sign(held_logits)
    == torch.sign(designed_utility(held_left) - designed_utility(held_right))
).double().mean()
metrics = {
    "held_nll": held_nll.item(),
    "oracle_nll": oracle_nll.item(),
    "order_accuracy": order_accuracy.item(),
}
return weights.detach(), metrics

# [3]
seeds = list(range(6050, 6055))
# [4]
narrow_fits = [fit_reward_model(seed, 0.0) for seed in seeds]
repaired_fits = [fit_reward_model(seed, 0.20) for seed in seeds]
narrow_weights = torch.stack([fit[0] for fit in narrow_fits])
repaired_weights = torch.stack([fit[0] for fit in repaired_fits])

def mean_and_seed_sd(values: Tensor) -> tuple[Tensor, Tensor]:
    return values.mean(dim=0), values.std(dim=0, unbiased=True)

narrow_mean, narrow_sd = mean_and_seed_sd(narrow_weights)
repaired_mean, repaired_sd = mean_and_seed_sd(repaired_weights)
narrow_nlls = torch.tensor([fit[1]["held_nll"] for fit in narrow_fits])
oracle_nlls = torch.tensor([fit[1]["oracle_nll"] for fit in narrow_fits])
narrow_accuracies = torch.tensor([
    fit[1]["order_accuracy"] for fit in narrow_fits
])

# [5]
print("feedback          w_e          w_length  w_curve")
print("narrow mean        ", " ".join(f"{x:.6f}" for x in narrow_mean))
print("narrow seed sd      ", " ".join(f"{x:.6f}" for x in narrow_sd))
print("repaired mean        ", " ".join(f"{x:.6f}" for x in repaired_mean))
print("repaired seed sd     ", " ".join(f"{x:.6f}" for x in repaired_sd))
print(

```

```

    "narrow in-range NLL:",
    f"{narrow_nlls.mean().item():.6f} +/-
      ↪ {narrow_nlls.std(unbiased=True).item():.6f}",
  )
print(
    "oracle in-range NLL:",
    f"{oracle_nlls.mean().item():.6f} +/-
      ↪ {oracle_nlls.std(unbiased=True).item():.6f}",
  )
print(
    "narrow designed-order accuracy:",
    f"{narrow_accuracies.mean().item():.6f} +/- "
    f"{narrow_accuracies.std(unbiased=True).item():.6f}",
  )

```

```

feedback          w_e      w_length  w_curve
narrow mean        1.004469  0.634378  0.000000
narrow seed sd     0.042489  0.019772  0.000000
repaired mean      0.984938  0.600285  -1.208128
repaired seed sd   0.073378  0.007907  0.025924
narrow in-range NLL: 0.665633 +/- 0.002097
oracle in-range NLL: 0.665603 +/- 0.002092
narrow designed-order accuracy: 0.992940 +/- 0.002548

```

The narrow model looks excellent where it was tested—yet its mean held-out log loss is 0.665633, only 0.000030 above the oracle that generated the labels, and its latent ordering accuracy is 0.992940. The coefficient w_q remains exactly zero: that feature was zero in every narrow comparison, so no gradient could identify its coefficient. A validation split drawn from the same support cannot expose the missing behavior.

Now place 401 candidate responses at fixed evidence $e = 0.72$ and lengths from zero to four. The reference policy is a discretized bell-shaped distribution with mean length 0.776439. For each fitted reward model and each β , compute the exact policy from Equation 18.5. No sampled-policy optimizer is needed—any failure comes from the proxy and its feedback coverage:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable `policy_metrics` helper.
 3. Optimize the proxy outside its comparison coverage.
-

CODE

```

# [1]
lengths = torch.linspace(0.0, 4.0, 401)
evidence = torch.full_like(lengths, 0.72)
candidates = torch.stack((evidence, lengths), dim=1)
candidate_features = reward_features(candidates)
candidate_designed_utility = designed_utility(candidates)
reference_policy = torch.softmax(
    -0.5 * ((lengths - 0.695) / 0.5).square(), dim=0
)
pressure_betas = [1.0, 0.5, 0.25, 0.125]

# [2]
def policy_metrics(weights_by_seed: Tensor) -> dict[float, Tensor]:
    rows: dict[float, Tensor] = {}
    for beta in pressure_betas:
        seed_rows = []
        for weights in weights_by_seed:
            proxy = candidate_features @ weights
            policy = torch.softmax(
                torch.log(reference_policy) + proxy / beta, dim=0
            )
            seed_rows.append(torch.stack((
                policy @ lengths,
                policy @ proxy,
                policy @ candidate_designed_utility,
                kl_divergence(policy, reference_policy),
            )))
        rows[beta] = torch.stack(seed_rows)
    return rows

narrow_rows = policy_metrics(narrow_weights)
repaired_rows = policy_metrics(repaired_weights)
reference_length = reference_policy @ lengths
reference_designed_utility = reference_policy @ candidate_designed_utility

# [3]
print(f"reference mean length: {reference_length.item():.6f}")
print(f"reference designed utility: {reference_designed_utility.item():.6f}")
print("narrow feedback: beta length proxy designed KL")
for beta, values in narrow_rows.items():
    mean, seed_sd = mean_and_seed_sd(values)
    print(
        f"{beta:5.3f} {mean[0].item():.6f} {mean[1].item():.6f}"
        f" {mean[2].item():.6f} {mean[3].item():.6f}"
        f" (designed sd {seed_sd[2].item():.6f})"
    )
print("repaired feedback: beta length proxy designed KL")

```

```

for beta, values in repaired_rows.items():
    mean, seed_sd = mean_and_seed_sd(values)
    print(
        f"{beta:5.3f}  {mean[0].item():.6f}  {mean[1].item():.6f}"
        f"  {mean[2].item():.6f}  {mean[3].item():.6f}"
        f"  (designed sd {seed_sd[2].item():.6f})"
    )

```

reference mean length: 0.776439

reference designed utility: 1.130480

narrow feedback: beta length proxy designed KL

1.000 0.901311 1.295055 1.164883 0.040327 (designed sd 0.000874)

0.500 1.037928 1.381796 1.183447 0.170991 (designed sd 0.000480)

0.250 1.335088 1.570472 1.135363 0.739579 (designed sd 0.007630)

0.125 1.963796 1.969632 0.484541 3.138049 (designed sd 0.067970)

repaired feedback: beta length proxy designed KL

1.000 0.837956 1.157585 1.168536 0.018632 (designed sd 0.000484)

0.500 0.895553 1.190942 1.201869 0.068282 (designed sd 0.000905)

0.250 0.990698 1.243968 1.254875 0.224387 (designed sd 0.001374)

0.125 1.104397 1.305569 1.316470 0.577948 (designed sd 0.001310)

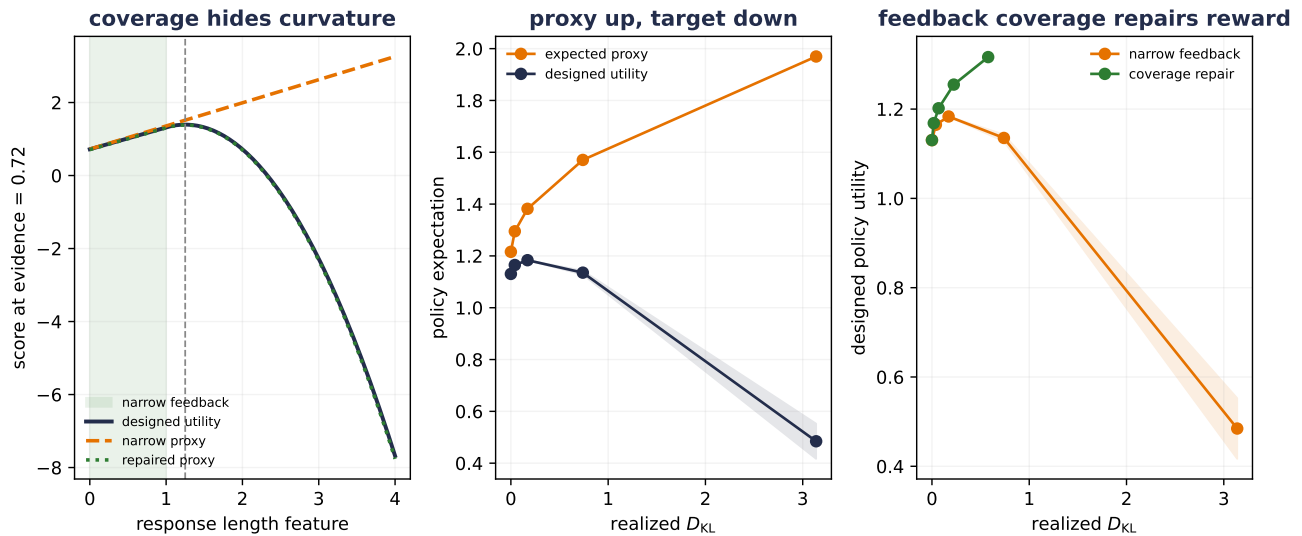


Figure 18.5.: A reward model can pass an in-range comparison test and fail under optimization. Narrow feedback covers only the shaded length interval, so the learned proxy remains linear after designed utility turns down (left). Under increasing reward pressure, its expected proxy score rises while designed utility is highest among the four tested settings at beta 0.5 and falls to 0.484541 at beta 0.125, with realized KL 3.138049 (center). Adding 20% out-of-range responses identifies the missing curvature; at beta 0.125 the repaired policies retain mean designed utility 1.316470 (right). Lines are means over five reward-model seeds; designed-utility bands show one seed SD where plotted.

At moderate pressure, the narrow proxy helps—mean designed utility rises from 1.130480 under the reference to 1.183447 at $\beta = 0.5$. More pressure then reverses the result. At $\beta = 0.125$, mean predicted reward has climbed to 1.969632 while designed utility has fallen to 0.484541. The KL penalty changes

how quickly the policy reaches the failure—it cannot invent the missing curvature.

The repair changes feedback coverage, not the policy optimizer—once 20% of individual responses expose the longer regime, the mean fitted curvature coefficient is -1.208128 (designed value -1.2). At $\beta = 0.125$, the repaired policies reach mean designed utility 1.316470 instead of 0.484541 . In this controlled world, better feedback coverage matters more than choosing a fancier route to the same proxy.

⚠ Reward-model validation is not policy validation

Held-out pair loss tests interpolation over a declared comparison distribution—policy optimization deliberately changes the response distribution. Evaluate current-policy outputs with independent judgments, track realized drift and length or style shifts, and refresh feedback where the policy now visits. A proxy score rising by itself is not evidence that the intended behavior improved.

18.5. DPO removes a stage, not the assumptions

Reward-model RLHF fits an explicit proxy, freezes it, then optimizes a policy against it. Can we use preference pairs without that separately trained middle model? The answer is the DPO “Aha!” move: invert Equation 18.5 and substitute the result into the preference model.

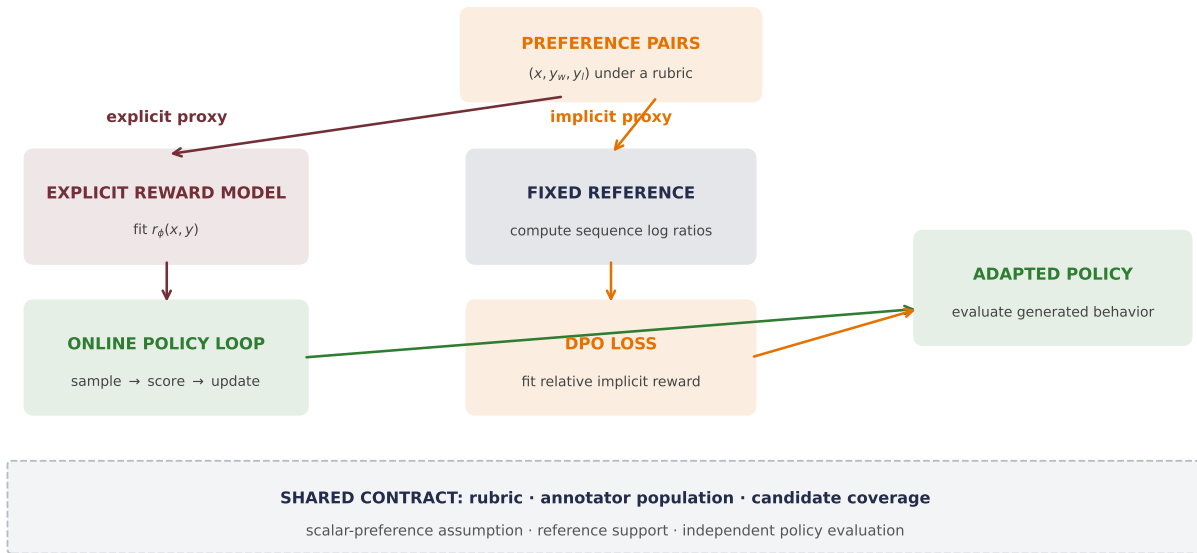


Figure 18.6.: Two routes can consume preference pairs. A reward-model route first fits an explicit scalar proxy, then scores fresh outputs during online policy optimization. DPO forms a reference-adjusted policy margin directly on a fixed pair dataset. It removes the explicit reward-model stage and online sampling during the optimization pass, but both routes still depend on the preference rubric, coverage, a scalar-reward model, and an adequate reference support.

Rearrange the exact optimum:

$$r(x, y) = \beta \log \frac{\pi^*(y | x)}{\pi_{\text{ref}}(y | x)} + \beta \log Z(x). \quad (18.8)$$

For two responses to the same prompt, the $\beta \log Z(x)$ term is identical and cancels from the reward difference. Define the current policy’s reference-adjusted score

$$s_\theta(x, y) = \log \pi_\theta(y \mid x) - \log \pi_{\text{ref}}(y \mid x).$$

Substituting Equation 18.8 into Bradley–Terry gives **Direct Preference Optimization (DPO)**:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}_{\text{pref}}} \log \sigma(\beta [s_\theta(x, y_w) - s_\theta(x, y_l)]). \quad (18.9)$$

For an autoregressive model, $\log \pi_\theta(y \mid x)$ is a **sum** over response token log probabilities, normally including the response end token:

$$\log \pi_\theta(y \mid x) = \sum_{t=1}^T \log \pi_\theta(y_t \mid x, y_{<t}).$$

The completion mask audited near Equation 18.1 applies here as well. Prompt and padding targets contribute zero—length-normalizing that sum would define a different objective and should be named as such.

Does the algebra work in the finite world? Use all six response pairs, but train on their exact stochastic Bradley–Terry probabilities rather than separable hard winners. For this identity check, set $\beta = 1$. The DPO logit for pair (i, j) is $\beta(s_i - s_j)$. Starting from the reference, optimize four unconstrained reference-relative offsets, identifiable only up to a shared additive constant:

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Recover the exact regularized policy with DPO.
-

CODE

```
# [1]
pairs = torch.tensor([
    [0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]
])
dpo_beta = 1.0
pair_probabilities = torch.sigmoid(
    reward[pairs[:, 0]] - reward[pairs[:, 1]]
)
target_policy = gibbs_policy(reference, reward, dpo_beta)

log_ratio = nn.Parameter(torch.zeros(4))
dpo_optimizer = torch.optim.Adam([log_ratio], lr=0.05)
for _ in range(4000):
    comparison_logits = dpo_beta * (
        log_ratio[pairs[:, 0]] - log_ratio[pairs[:, 1]]
```

```

    )
    dpo_loss = F.binary_cross_entropy_with_logits(
        comparison_logits, pair_probabilities
    )
    dpo_optimizer.zero_grad()
    dpo_loss.backward()
    dpo_optimizer.step()

dpo_policy = torch.softmax(
    torch.log(reference) + log_ratio.detach(), dim=0
)
# [2]
print("analytic policy:", [round(x, 6) for x in target_policy.tolist()])
print("DPO policy:      ", [round(x, 6) for x in dpo_policy.tolist()])
print(
    "largest policy difference:",
    f"{{(dpo_policy - target_policy).abs().max().item():.2e}}",
)

```

```

analytic policy: [0.164562, 0.20333, 0.331625, 0.300483]
DPO policy:      [0.164562, 0.20333, 0.331625, 0.300483]
largest policy difference: 5.55e-17

```

The match to floating-point tolerance is an identity test under deliberately strong conditions: a fixed positive reference, a complete finite response set, all pairwise comparisons, preferences generated by one scalar reward, adequate policy capacity, and successful optimization—not that natural-language DPO and PPO are universally equivalent.

Several details are easy to overstate:

- DPO avoids an **explicit standalone** reward model and an online rollout loop during the optimization pass. Its policy–reference log ratio is still an implicit reward representation, and candidate generation plus labeling happened upstream.
- The reference need not be the exact behavior policy that generated every pair. It is a density-ratio anchor with a support and modeling role.
- β appears in the regularized derivation, but finite-data DPO does not enforce a hard realized-KL budget. Measure the KL.
- The gradient rewards a *relative reference-adjusted margin*. Shared parameters mean one optimizer step need not raise the winner’s absolute probability and lower the loser’s absolute probability separately.
- Offline pairs cannot correct missing candidate coverage by themselves. DPO can fit a stale or narrow preference dataset efficiently.

i What the finite identity proves

The four-response test validates the derivation inside its scalar-preference, support, coverage, capacity, and optimization assumptions. The cyclic and proxy-pressure audits test two ways those assumptions fail.

18.6. Choose by the feedback and exploration you have

There is no universal method ranking. Start by naming the information source and whether new responses must be explored during optimization, as Table 18.1 makes explicit:

Table 18.1.: Feedback source and exploration needs determine which alignment route is available.

Route	Supervision	Fitted quantity	Fresh samples?	Main risk
Response-masked SFT	Demonstrated responses	Likelihood of demonstration tokens	No	Demonstration coverage and imitation target
DPO	Fixed preferred/rejected pairs	Reference-adjusted preference margin	No	Pair coverage, reference, scalar-preference assumptions
Explicit RM + online policy optimization	Comparisons plus a reward model; current responses can be scored	Reward proxy, value function, and policy objective	Yes	Proxy shift, exploration, credit assignment, optimization stability

SFT may be enough when demonstrations directly specify the desired behavior. Fixed preference data may favor an offline direct method when comparison coverage is already good. Online methods can generate from the changing policy and request or compute fresh feedback, which matters when exploration is part of the task. They also add a larger failure and evaluation surface.

Chapter 17’s axis remains orthogonal. SFT, DPO, or an online policy objective may update full weights, a LoRA branch, or another permitted state. Parameter efficiency changes the write surface and resource bill—not the rubric, comparison coverage, or reward fidelity.

Variants rearrange sampling, scoring, and labeling. Rejection-sampling fine-tuning selects high-scoring samples and imitates them. AI feedback replaces or supplements human labels but inherits the evaluator model’s coverage and biases. Constitutional AI, for example, combines supervised critique and revision with later preference-based RL guided by written principles. These are useful routes for further study—not a reason to turn the durable chapter into an acronym list.

18.7. Alignment is an evaluation contract

An optimization loss is one line in a much larger contract—a credible report connects the people and prompts that created feedback to independent tests of the policy that will actually be used:

The feedback report should identify the prompt distribution, selection policy for candidates, rubric, reviewer population, and aggregation rule. If several pairs came from one ranked set, they are correlated—keep their prompt together across splits and cluster uncertainty at the prompt level. Randomize A/B order, retain ties, and disclose disagreement rather than converting it silently into certainty.

Reward-model evaluation should include held-out pair log loss, calibration or Brier score, and tests on candidates from newer policies. Policy evaluation must go beyond reward: blinded independent preferences, task-specific success, realized KL to the reference, response-length and refusal shifts, retained

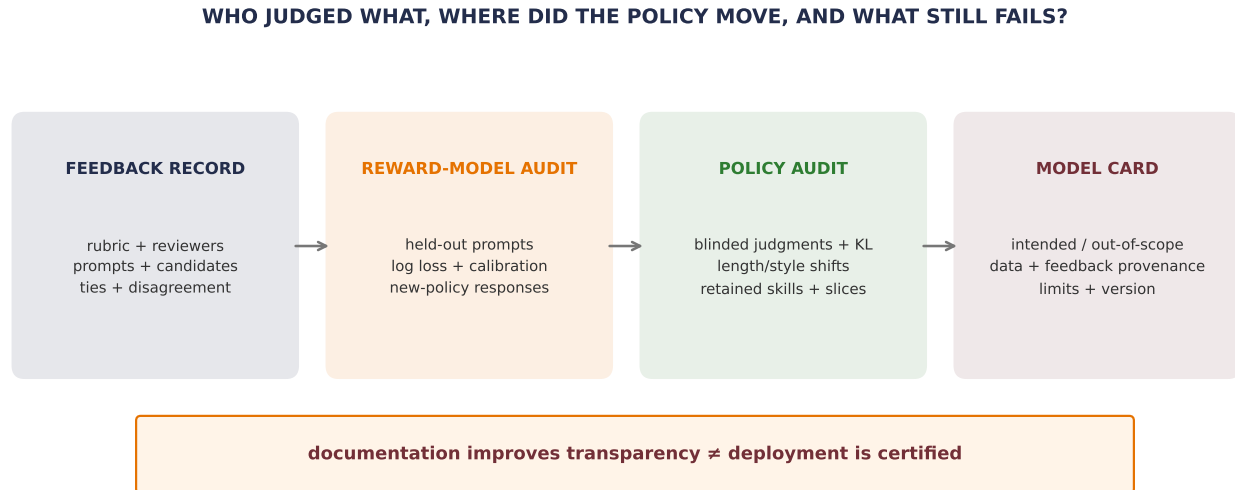


Figure 18.7.: Alignment evaluation is a chain of evidence, not one score. The feedback record identifies the rubric, reviewers, prompts, candidates, ties, and disagreement. Reward-model audits hold out prompts and test calibration and newer-policy responses. Policy audits add blinded judgments, realized KL, length and style shifts, retained capabilities, and relevant slices. A model card records intended and out-of-scope uses, data and feedback provenance, limitations, and versioning; it does not certify safety.

capabilities, and safety or domain slices relevant to intended use. Give sample counts and uncertainty. A raw gap between two slices is not automatically statistically meaningful.

An LLM can assist as a judge—another measurement instrument. Published audits have found position, verbosity, and self-enhancement biases. Blinding model identities, swapping candidate order, keeping ties, and validating against human judgments are important safeguards.

The instructor calls a **model card** a model’s nutritional label—the analogy works: record model and training details, intended users and uses, out-of-scope uses, data and feedback provenance, evaluation conditions, slice results, limitations, version, and contact. A label helps a reader make an informed decision—it does not prevent misuse, prove that omitted groups are safe, or convert a conditional evaluation into a certificate.

! Trap: alignment is not the whole ethics of a system

This chapter asks whether a model’s behavior matches a declared feedback and evaluation contract. That question matters, but it does not absorb every sociotechnical question around the system:

- **Data and rights:** privacy, consent, ownership, and the legitimacy of collection.
- **Distribution of effects:** who receives benefits, errors, exclusions, or recourse.
- **Production:** human labor, resource use, environmental cost, and supply chains.
- **Governance and use:** misuse, institutional incentives, accountability, oversight, and appeal.

A model card can expose these choices. It cannot confer consent, resolve values, assign accountability, or certify deployment. Appendix E scopes statistical evidence; ethical and governance judgment remains separate.

💡 Write the evaluation contract before optimization

Predeclare the rubric, feedback population, prompt and candidate coverage, ties, primary policy outcomes, retained capabilities, slices, decoding settings, and drift measure. Then a rising training objective has a declared external test. Without that contract, “better aligned” can collapse into “the optimizer increased its own score.”

18.8. The next question: where did the generator come from?

A judge is not a generator—a reward model can rank or score completed responses; it does not explain how a model learned a distribution from which responses can be sampled. RLHF and DPO begin with a generator already in hand.

The **autoencoder interlude** already traced reconstruction from PCA’s linear geometry to trainable nonlinear codes. **Chapter 19** begins at the next missing contract: how a model defines or learns a distribution and turns it into samples, then develops variational autoencoders, adversarial learning, and diffusion. Chapter 18 changes the behavior of a generator—Chapter 19 studies the machinery that makes generation possible.

i Check yourself

Close the book for one minute and separate update location from objective.

- Which tokens contribute to the supervised fine-tuning loss?
- What information does a preference pair add beyond one demonstration?
- How do reward modeling, RLHF, and DPO differ in where the learned signal enters?

18.9. Okay, so — where to update and what to optimize are separate

1. **Update location and objective are separate axes.** Full tuning, LoRA, and soft prompts decide where gradients may write. SFT, preference margins, and sampled rewards decide what assigns blame.
2. **SFT imitates demonstrated response tokens.** A completion mask keeps prompt and padding targets out of the loss. Demonstrations can encode quality; preference data supplies a different, comparative signal.
3. **A preference is conditional evidence.** Bradley–Terry turns score differences into probabilities, but reward zero points are unidentified and cyclic aggregate preferences need not fit one scalar ordering.
4. **RLHF separates judge from actor.** The reward model scores completed responses, the value function forecasts shaped return from prefixes, PPO uses a refreshed old policy, and reference KL uses a usually fixed anchor.
5. **Reference KL is a pressure dial.** The finite optimum is $\pi^* \propto \pi_{\text{ref}} e^{r/\beta}$. More pressure can raise proxy reward; it cannot repair a proxy’s missing feature.
6. **DPO removes an explicit stage, not the assumptions.** The four-response identity is exact under scalar preferences, compatible support, complete coverage, adequate capacity, and successful optimization. Realized KL and generated behavior still need measurement.

7. **Alignment is an evaluation contract, not a certificate.** State who judged which candidates under what rubric, then evaluate the current policy independently and document intended use, slices, uncertainty, and limitations. This contract is one part of a broader ethical and governance assessment, not a substitute for it.

Sources and further reading

- Wei et al., *Finetuned Language Models Are Zero-Shot Learners*.
- Christiano et al., *Deep Reinforcement Learning from Human Preferences*.
- Stiennon et al., *Learning to Summarize from Human Feedback*: feedback training and overoptimization.
- Askell et al., *A General Language Assistant as a Laboratory for Alignment*: helpful, honest, and harmless framing.
- Ouyang et al., *Training Language Models to Follow Instructions with Human Feedback*: SFT, reward modeling, PPO, and scope.
- Schulman et al., *Proximal Policy Optimization Algorithms*: the clipped policy-ratio surrogate.
- Rafailov et al., *Direct Preference Optimization*: the reward-policy identity and DPO loss.
- Gao et al., *Scaling Laws for Reward Model Overoptimization*: controlled proxy-overoptimization studies.
- Mitchell et al., *Model Cards for Model Reporting*: evaluation and documentation contracts.
- Selbst et al., *Fairness and Abstraction in Sociotechnical Systems*: why a technical model must be read inside the social system where it operates.
- Raji et al., *Closing the AI Accountability Gap*: an end-to-end framework for internal algorithmic auditing.
- Zheng et al., *Judging LLM-as-a-Judge*: model-judge biases.
- Bai et al., *Constitutional AI*: critique, revision, and AI feedback.

Exercises

1. **(Pencil.)** Given $P(A \succ B) = 0.8$ and $P(B \succ C) = 0.7$, compute the implied $P(A \succ C)$ under Bradley–Terry. Prove that adding the same $c(x)$ to every candidate reward leaves all pairwise probabilities unchanged. Then add $P(C \succ A) = 0.6$ and explain why the inconsistency diagnoses model misspecification—not reviewer error.
2. **(Code.)** Extend `completion_log_probability` to chosen and rejected batches with unequal padding. Include EOS but exclude prompt and padding targets; assert that changing excluded logits leaves both sums unchanged. Compute the DPO margin and verify identical policy/reference tokenizers, truncation, and masks.
3. **(Pencil.)** (a) Derive Equation 18.5 with a Lagrange multiplier. **(Code.)** (b) Reproduce the four-response β sweep and verify normalized policies and nonnegative KL. Train DPO using only A–B and C–D; initialize the two connected components with different shared offsets and show that their margins can agree while probability mass between components remains unidentified.
4. **(Code.)** Reproduce the proxy-pressure study. Predeclare one repair: expand feedback support, add a missing feature, or both. Report held-out comparison loss, learned coefficients, realized KL, proxy reward, and designed utility as mean $\pm$ seed SD over five seeds. Explain why improving in-range log loss alone may not change the policy failure.

5. (**Audit.**) Choose medicine, creative writing, or executable verification. Using Table 18.1, compare SFT, DPO, and online reward in a model card covering supervision/write surface, reference/exploration, evaluators, unsupported claims, retention/slices, and uncertainty.

19. Generative Models: From Codes to Samples

⚠ Trap: a judge is not a generator

A reward model can score a completed response, and a preference loss can move probability among compared responses. Both begin with a generator already in hand. Neither explains how a model first learns a distribution from which a new sample can be drawn.

The interlude before Part III made a different promise: **a code is not yet a distribution**. Reconstruction supplies encoded observations and a decoder, but no principled random start. This chapter harvests both promises by learning the missing sampling contract:

probabilistic latent variables $\longrightarrow$ adversarial sampling $\longrightarrow$ iterative denoising.

Variational autoencoders (VAEs), generative adversarial networks (GANs), and diffusion models make different choices along that route. Each supplies a sampling rule, a training signal, and a characteristic way to fail. A high training score does not make those choices interchangeable.

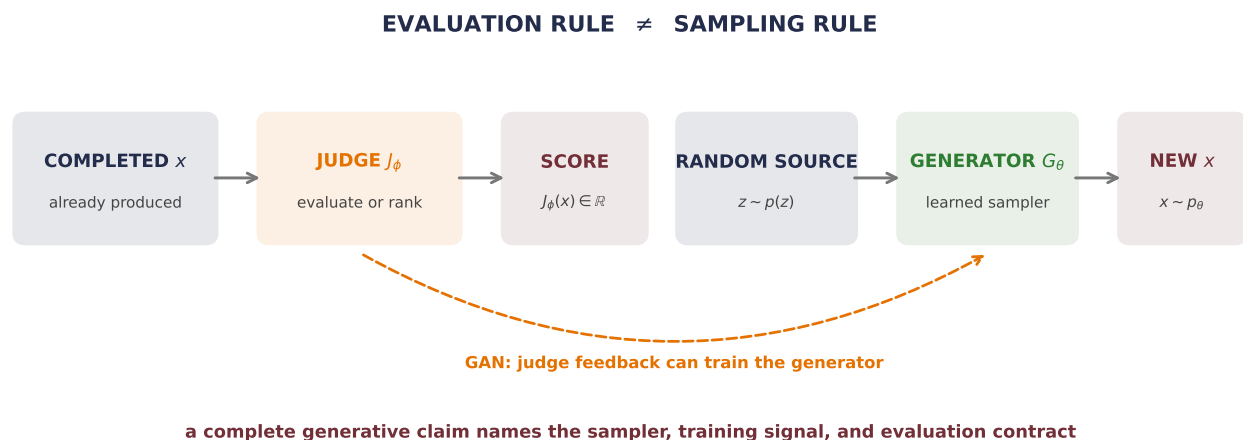


Figure 19.1.: A judge and a generator solve different problems. A judge maps a completed object to a score. A generator maps a random source through a learned sampling rule to a new object. A GAN connects the two during training, but the discriminator still does not become the sampler.

The first answer is to put an explicit probability law around the code.

19.1. Put probability around the code

A **variational autoencoder (VAE)** begins by defining generation—not reconstruction. Let $\mathbf{z} \in \mathbb{R}^k$ be a latent variable with a chosen prior, commonly

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I}_k).$$

A decoder with parameters θ specifies a conditional distribution $p_\theta(\mathbf{x} | \mathbf{z})$. Together they define the joint density

$$p_\theta(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p_\theta(\mathbf{x} | \mathbf{z})$$

and the marginal model

$$p_\theta(\mathbf{x}) = \int p(\mathbf{z})p_\theta(\mathbf{x} | \mathbf{z}) d\mathbf{z}. \quad (19.1)$$

Generation is now a declared procedure—draw $\mathbf{z} \sim p(\mathbf{z})$, then draw or decode $\mathbf{x} \sim p_\theta(\mathbf{x} | \mathbf{z})$. The encoder is not used on this path—it belongs to inference and training.

Training is harder because the posterior $p_\theta(\mathbf{z} | \mathbf{x})$ in Bayes' rule contains the integral in Equation 19.1. A VAE introduces an **approximate posterior**, or inference model, whose parameters come from an encoder:

$$q_\phi(\mathbf{z} | \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{x}), \text{diag } \boldsymbol{\sigma}_\phi^2(\mathbf{x})). \quad (19.2)$$

The decoder p_θ defines the generative model. The encoder q_ϕ makes latent inference and training practical. Their arrows point in opposite directions—their probabilistic roles are not symmetric.

19.1.1. The ELBO is an exact gap identity

Why does this approximate posterior give us a trainable objective? Start from the log evidence and add a quantity that equals zero:

$$\begin{aligned} \log p_\theta(\mathbf{x}) &= \mathbb{E}_{q_\phi(\mathbf{z} | \mathbf{x})} \left[\log \frac{p_\theta(\mathbf{x}, \mathbf{z})}{q_\phi(\mathbf{z} | \mathbf{x})} \right] \\ &\quad + D_{\text{KL}}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x})). \end{aligned} \quad (19.3)$$

The expectation in the first line expands into

$$\mathcal{L}_{\text{ELBO}}(\theta, \phi; \mathbf{x}) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z} | \mathbf{x})} [\log p_\theta(\mathbf{x} | \mathbf{z})]}_{\text{expected reconstruction log-likelihood}} - \underbrace{D_{\text{KL}}(q_\phi(\mathbf{z} | \mathbf{x}) \| p(\mathbf{z}))}_{\text{prior regularization}}. \quad (19.4)$$

KL divergence is nonnegative, so the **evidence lower bound (ELBO)** is no larger than $\log p_\theta(\mathbf{x})$. More importantly, Equation 19.3 says exactly how loose it is—the gap is the KL divergence from the

approximate posterior to the model’s true posterior. This is not merely a convenient inequality—it is an identity.

For the diagonal Gaussian in Equation 19.2 and a standard-normal prior, the regularizer is available coordinate by coordinate:

$$D_{\text{KL}}(q_{\phi}(\mathbf{z} \mid \mathbf{x}) \parallel p(\mathbf{z})) = \frac{1}{2} \sum_{j=1}^k (\mu_{\phi,j}(\mathbf{x})^2 + \sigma_{\phi,j}(\mathbf{x})^2 - \log \sigma_{\phi,j}(\mathbf{x})^2 - 1). \quad (19.5)$$

This closed form is why an encoder commonly returns $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$: the KL does not need Monte Carlo estimation.

Let us pin every term in a case where the true posterior is available. Take

$$z \sim \mathcal{N}(0, 1), \quad x \mid z \sim \mathcal{N}(z, 0.36), \quad x = 1.2.$$

Gaussian conditioning gives

$$p(z \mid x) = \mathcal{N}\left(\frac{x}{1 + 0.36}, \frac{0.36}{1 + 0.36}\right) = \mathcal{N}(0.882353, 0.264706).$$

The next audit compares the exact posterior with a deliberately mismatched $q(z \mid x) = \mathcal{N}(0.35, 0.75)$.

PLAN

1. Define the reusable helpers: `gaussian_kl`, `gaussian_elbo`, and `gaussian_density`.
 2. Prepare the inputs and fixed settings for the example.
 3. Verify the exact Gaussian ELBO gap.
-

CODE

```
# [1]
def gaussian_kl(
    mean_q: float, variance_q: float, mean_p: float, variance_p: float
) -> float:
    return 0.5 * (
        math.log(variance_p / variance_q)
        + (variance_q + (mean_q - mean_p) ** 2) / variance_p
        - 1.0
    )

def gaussian_elbo(
    x: float, likelihood_variance: float, mean_q: float, variance_q: float
) -> float:
    expected_log_likelihood = -0.5 * (
        math.log(2.0 * math.pi * likelihood_variance)
```

```

        + ((x - mean_q) ** 2 + variance_q) / likelihood_variance
    )
    return expected_log_likelihood - gaussian_kl(mean_q, variance_q, 0.0, 1.0)

def gaussian_density(grid: Tensor, mean: float, variance: float) -> Tensor:
    return torch.exp(-(grid - mean).square() / (2.0 * variance)) / math.sqrt(
        2.0 * math.pi * variance
    )

# [2]
x_observed = 1.2
likelihood_variance = 0.36
posterior_mean = x_observed / (1.0 + likelihood_variance)
posterior_variance = likelihood_variance / (1.0 + likelihood_variance)
log_evidence = -0.5 * (
    math.log(2.0 * math.pi * (1.0 + likelihood_variance))
    + x_observed**2 / (1.0 + likelihood_variance)
)
exact_elbo = gaussian_elbo(
    x_observed, likelihood_variance, posterior_mean, posterior_variance
)
mismatch_mean, mismatch_variance = 0.35, 0.75
mismatch_elbo = gaussian_elbo(
    x_observed, likelihood_variance, mismatch_mean, mismatch_variance
)
posterior_gap = gaussian_kl(
    mismatch_mean, mismatch_variance, posterior_mean, posterior_variance
)

# [3]
print(f"posterior mean:      {posterior_mean:.12f}")
print(f"posterior variance: {posterior_variance:.12f}")
print(f"log evidence:       {log_evidence:.12f}")
print(f"exact ELBO:          {exact_elbo:.12f}")
print(f"mismatched ELBO:      {mismatch_elbo:.12f}")
print(f"evidence - ELBO:      {log_evidence - mismatch_elbo:.12f}")
print(f"KL(q || posterior): {posterior_gap:.12f}")

```

```

posterior mean:      0.882352941176
posterior variance: 0.264705882353
log evidence:       -1.602092647785
exact ELBO:         -1.602092647785
mismatched ELBO:    -2.533342834553
evidence - ELBO:     0.931250186769
KL(q || posterior): 0.931250186769

```

When q equals the true posterior, the gap vanishes to floating-point tolerance. The mismatched q leaves almost one nat of evidence unexplained—we can improve the ELBO by improving the decoder likelihood,

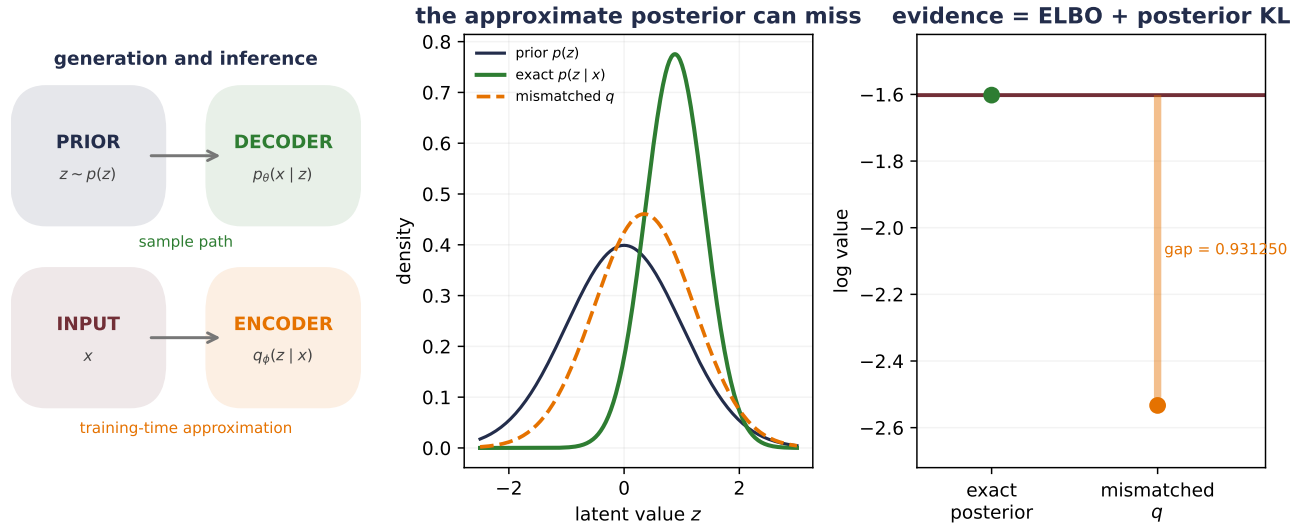


Figure 19.2.: The VAE lower bound is audited in a one-dimensional Gaussian model. For $x = 1.2$ and observation variance 0.36, the exact posterior has mean 0.882353 and variance 0.264706. The log evidence and exact-posterior ELBO are both -1.602093. For $q = N(0.35, 0.75)$, the ELBO is -2.533343; its 0.931250 gap to the evidence exactly matches $\text{KL}(q \parallel p(z | x))$.

the approximate posterior, or both—one scalar objective couples those jobs.

If $p_\theta(\mathbf{x} | \mathbf{z})$ is an isotropic Gaussian with fixed variance, its negative log-likelihood is a scaled squared reconstruction term plus a constant. This explains the familiar VAE shorthand “reconstruction plus KL.” The variance fixes the relative scale—treating the coefficient as decoration changes the model.

A β -weighted variant deliberately multiplies the KL term by $\beta \neq 1$. That defines a different optimization objective from the ordinary ELBO for the same decoder model. It can change the reconstruction–regularization trade-off; it does not guarantee discovery of independent true factors.

i Why this batches

The training objective is a population average of *per-example* quantities,

$$\frac{1}{N} \sum_i \left[\mathbb{E}_{q_\phi(\mathbf{z} | \mathbf{x}_i)} \log p_\theta(\mathbf{x}_i | \mathbf{z}) - \beta \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}_i) \parallel p(\mathbf{z})) \right],$$

so a minibatch mean is an unbiased estimate of it — the same linearity-of-expectation license that justified SGD in Chapter 4. The KL that looks like “one global term” in the single- $\mathbf{x}$ derivation is, in training, one KL *per datapoint* inside the sum. The reduction sequence to remember: **sum coordinates within an example, then average examples across the batch**. In the five-row template — *Target*: the population ELBO; *Estimator*: batch mean of per-example ELBOs; *Reduction*: sum over pixels and latent coordinates, mean over examples; *Validity*: the objective is a per-example sum (estimator case i); *Boundary*: any term defined through the *aggregate* posterior $q_{\text{agg}}(\mathbf{z}) = \frac{1}{N} \sum_i q_\phi(\mathbf{z} | \mathbf{x}_i)$ — as in total-correlation variants — is a nonlinear functional of the whole dataset (case ii), and naive per-batch code for it is silently biased. Exercises 4 and 5 make both boundaries fail in your hands.

19.1.2. Move randomness outside the learned map

Sampling $\mathbf{z} \sim q_\phi(\mathbf{z} \mid \mathbf{x})$ appears to interrupt backpropagation. The **reparameterization trick** writes the same draw as

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_k), \quad \mathbf{z} = \boldsymbol{\mu}_\phi(\mathbf{x}) + \boldsymbol{\sigma}_\phi(\mathbf{x}) \odot \epsilon. \quad (19.6)$$

Randomness has not disappeared—it has moved into a parameter-independent input, while the path from $\boldsymbol{\mu}_\phi$ and $\boldsymbol{\sigma}_\phi$ to $\mathbf{z}$ is now differentiable.

💡 Reparameterization hygiene

For a batch, `mu`, `log_variance`, `epsilon`, and `z` should all have shape (B, k) . Compute `standard_deviation = exp(0.5 * log_variance)` and then `z = mu + standard_deviation * epsilon`. Predicting log variance keeps the variance positive without clipping and avoids confusing variance with standard deviation.

The KL term pressures each approximate posterior toward the prior—but it does not prove that every prior draw decodes to a valid sample, that latent distance equals semantic distance, or that coordinates recover independent true factors. Too much pressure can also make $q_\phi(\mathbf{z} \mid \mathbf{x})$ ignore $\mathbf{x}$, a failure called **posterior collapse**. A VAE supplies a probability contract; its fit and coverage still need evaluation.

19.2. An adversarial detour: let the judge move

A VAE writes down a latent-variable likelihood and optimizes a lower bound. A **generative adversarial network (GAN)** takes a different route—a generator G_θ turns noise $\mathbf{z} \sim p_z$ into samples. A discriminator $D_\phi(\mathbf{x}) \in [0, 1]$ tries to separate data from generated samples (a finite sigmoid parameterization stays inside that interval):

$$\min_{\theta} \max_{\phi} \left\{ \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log D_\phi(\mathbf{x}) + \mathbb{E}_{\mathbf{z} \sim p_z} \log (1 - D_\phi(G_\theta(\mathbf{z}))) \right\}. \quad (19.7)$$

Here the Chapter 18 distinction returns with a twist—the judge and generator both move. The discriminator is not a fixed reward model, and its score is not a stationary evaluation metric.

Training therefore alternates roles. Hold G_θ fixed while updating D_ϕ on real and generated examples; then hold the discriminator parameters fixed while gradients pass through the current D_ϕ to update G_θ . Each player learns against a moving opponent.

For fixed generator density p_g and an unrestricted discriminator, the maximizer can be found point by point:

$$D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}. \quad (19.8)$$

This expression applies where $p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x}) > 0$. Outside their combined support, the objective does not identify D^* .

Substituting D^* into the value gives

$$V(D^*, G) = -\log 4 + 2D_{\text{JS}}(p_{\text{data}} \| p_g), \quad (19.9)$$

where D_{JS} is Jensen–Shannon divergence. The ideal function-space minimum occurs at $p_g = p_{\text{data}}$, where $D^* = 1/2$ —an observed neural discriminator near one half does not certify that equilibrium; it may also be weak, undertrained, or locally confused.

PLAN

1. Define the reusable `js_divergence` helper.
 2. Prepare the inputs and fixed settings for the example.
 3. Audit finite GAN coverage and generator saturation.
-

CODE

```
# [1]
def js_divergence(p: Tensor, q: Tensor) -> Tensor:
    midpoint = 0.5 * (p + q)
    return 0.5 * (
        (p * torch.log(p / midpoint)).sum()
        + (q * torch.log(q / midpoint)).sum()
    )

# [2]
data_mass = torch.tensor([0.45, 0.45, 0.10])
generator_masses = {
    "covered": torch.tensor([0.40, 0.45, 0.15]),
    "collapsed": torch.tensor([0.02, 0.96, 0.02]),
}
gan_rows = {}
for name, generator_mass in generator_masses.items():
    optimal_discriminator = data_mass / (data_mass + generator_mass)
    value = (
        (data_mass * torch.log(optimal_discriminator)).sum()
        + (generator_mass * torch.log(1.0 - optimal_discriminator)).sum()
    )
    divergence = js_divergence(data_mass, generator_mass)
    identity = -math.log(4.0) + 2.0 * divergence
    gan_rows[name] = optimal_discriminator, divergence, value, identity

# [3]
logit_grid = torch.linspace(-6.0, 6.0, 300)
discriminator_output = torch.sigmoid(logit_grid)
minimax_magnitude = discriminator_output
nonsaturating_magnitude = 1.0 - discriminator_output
```

```

print("candidate   JSD           V(D*,G)       -log(4)+2JSD   error")
for name, (_, divergence, value, identity) in gan_rows.items():
    print(
        f"{name:9s} {divergence.item():.12f} {value.item():.12f} "
        f"{identity.item():.12f} {(value - identity).abs().item():.2e}"
    )
for logit in [-6.0, -2.0, 0.0, 2.0]:
    probability = torch.sigmoid(torch.tensor(logit)).item()
    print(
        f"logit {logit:4.1f}: D={probability:.6f}, "
        f"minimax |grad|={probability:.6f}, "
        f"non-saturating |grad|={1.0 - probability:.6f}"
    )

```

```

candidate   JSD           V(D*,G)       -log(4)+2JSD   error
covered     0.003252657945 -1.379789045231 -1.379789045231 2.22e-16
collapsed   0.183269823016 -1.019754715087 -1.019754715087 0.00e+00
logit -6.0: D=0.002473, minimax |grad|=0.002473, non-saturating |grad|=0.997527
logit -2.0: D=0.119203, minimax |grad|=0.119203, non-saturating |grad|=0.880797
logit  0.0: D=0.500000, minimax |grad|=0.500000, non-saturating |grad|=0.500000
logit  2.0: D=0.880797, minimax |grad|=0.880797, non-saturating |grad|=0.119203

```

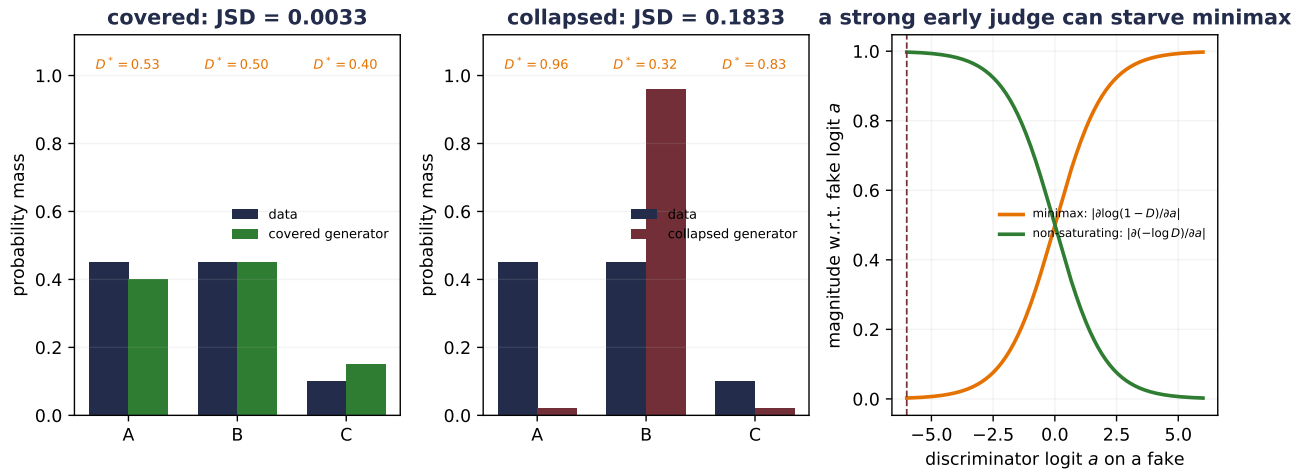


Figure 19.3.: A finite three-mode GAN audit. The covered generator $q = (0.40, 0.45, 0.15)$ has Jensen–Shannon divergence 0.003253 and optimal-discriminator value -1.379789. The collapsed $q = (0.02, 0.96, 0.02)$ has divergence 0.183270 and value -1.019755; both satisfy $V(D^*, G) = -\log 4 + 2 \text{JSD}$ to floating-point precision. When the discriminator assigns a fake logit -6, its output is 0.002473: the original minimax generator’s logit-gradient magnitude is 0.002473, versus 0.997527 for the non-saturating loss.

The non-saturating generator loss $-\mathbb{E}_{\mathbf{z}} \log D(G(\mathbf{z}))$ changes the early gradient signal without changing the ideal target. GANs can produce a sample in one generator pass and need not expose a tractable density. Their coupled optimization can be delicate—a generator may map many inputs into only a few data modes. That **mode collapse** is a coverage failure. A discriminator loss alone will not tell us which modes are missing.

19.3. Destroy data on purpose

Diffusion begins with a procedure that looks backward—add noise until structure is almost gone. The **forward process** is fixed, not learned. Choose a schedule $0 < \beta_t < 1$ for $t = 1, \dots, T$, and define

$$\alpha_t = 1 - \beta_t, \quad \bar{\alpha}_t = \prod_{s=1}^t \alpha_s. \quad (19.10)$$

For $\mathbf{x}_t \in \mathbb{R}^d$, one forward transition is

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}(\sqrt{\alpha_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I}_d). \quad (19.11)$$

Repeated substitution combines all earlier Gaussian noise into one standard-normal vector:

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I}_d), \quad (19.12)$$

or, equivalently,

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d). \quad (19.13)$$

This direct form is the training shortcut—choose a clean example, choose a timestep, draw one noise vector, and jump there without simulating all previous transitions.

Let us watch a balanced one-dimensional mixture

$$p_0(x) = \frac{1}{2} \mathcal{N}(-2, 0.25) + \frac{1}{2} \mathcal{N}(2, 0.25)$$

under a 100-step linear schedule from $\beta_1 = 10^{-4}$ to $\beta_{100} = 0.1$. Its two component means shrink by $\sqrt{\bar{\alpha}_t}$, while each component variance becomes $0.25\bar{\alpha}_t + (1 - \bar{\alpha}_t)$.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Audit a forward noise schedule and its direct marginal.
-

CODE

```

# [1]
diffusion_steps = 100
diffusion_beta = torch.linspace(1e-4, 0.1, diffusion_steps)
diffusion_alpha = 1.0 - diffusion_beta
diffusion_alpha_bar = torch.cumprod(diffusion_alpha, dim=0)
diffusion_alpha_bar_with_zero = torch.cat((torch.ones(1), diffusion_alpha_bar))

# [2]
torch.manual_seed(6050)
audit_x0 = torch.randn(10_000)
step_noises = torch.randn(diffusion_steps, audit_x0.numel())
sequential_state = audit_x0.clone()
accumulated_noise = torch.zeros_like(audit_x0)
for step in range(diffusion_steps):
    sequential_state = (
        torch.sqrt(diffusion_alpha[step]) * sequential_state
        + torch.sqrt(diffusion_beta[step]) * step_noises[step]
    )
    accumulated_noise = (
        torch.sqrt(diffusion_alpha[step]) * accumulated_noise
        + torch.sqrt(diffusion_beta[step]) * step_noises[step]
    )
direct_state = (
    torch.sqrt(diffusion_alpha_bar[-1]) * audit_x0 + accumulated_noise
)
effective_epsilon = accumulated_noise / torch.sqrt(
    1.0 - diffusion_alpha_bar[-1]
)

print("t  alpha_bar      signal      noise")
for time_index in [1, 10, 50, 100]:
    alpha_bar_at_time = diffusion_alpha_bar[time_index - 1]
    print(
        f"{time_index:3d}  {alpha_bar_at_time.item():.12f}  "
        f"{torch.sqrt(alpha_bar_at_time).item():.12f}  "
        f"{torch.sqrt(1.0 - alpha_bar_at_time).item():.12f}"
    )
print(
    "largest sequential/direct difference at T:",
    f"{(sequential_state - direct_state).abs().max().item():.2e}",
)
print(
    "effective epsilon mean / variance:",
    f"{effective_epsilon.mean().item():.6f}",
    f"{effective_epsilon.var(unbiased=False).item():.6f}",
)

```

```

t  alpha_bar      signal      noise
1  0.999900000000  0.999949998750  0.010000000000

```

```

10  0.954507754431  0.976989127079  0.213289112637
50  0.282980837397  0.531959432097  0.846769840395
100 0.005618761019  0.074958395256  0.997186662055
largest sequential/direct difference at T: 1.78e-15
effective epsilon mean / variance: 0.003264 0.991711
    
```

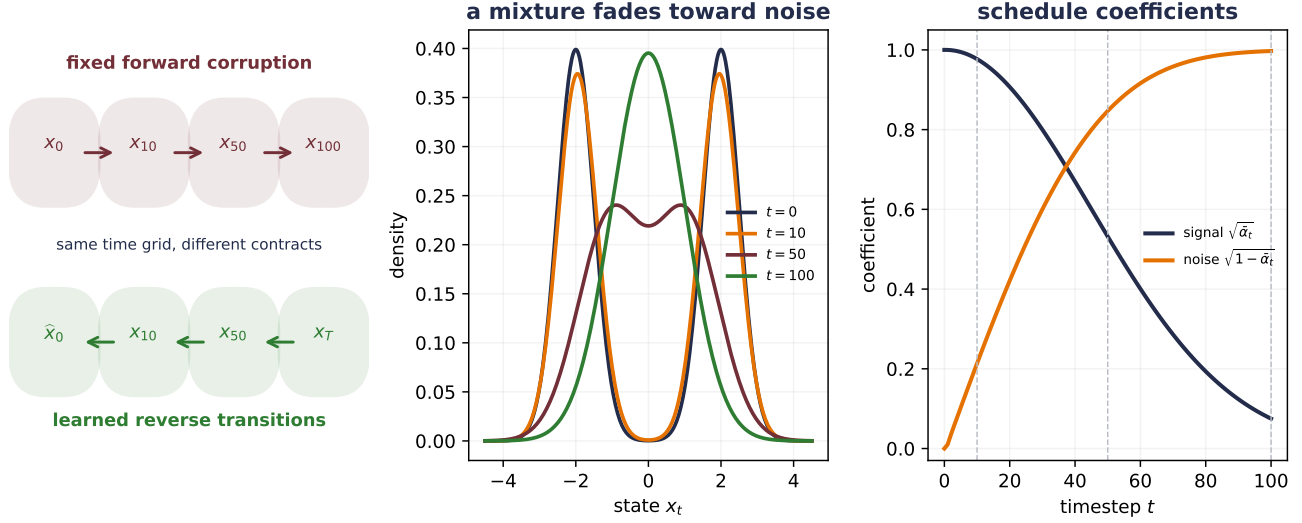


Figure 19.4.: Forward diffusion replaces a two-mode distribution by an almost standard-normal endpoint. Under the displayed 100-step schedule, α -bar is 0.954508 at $t = 10$, 0.282981 at $t = 50$, and 0.005619 at $t = 100$. The final signal and noise coefficients are 0.074958 and 0.997187. The lower chain is learned in the opposite direction; it is not the algebraic inverse of one realized forward chain.

At $t = 100$, a trace of the original mixture remains because $\bar{\alpha}_T > 0$. The endpoint is close to—but not exactly—a standard normal. That detail matters when generation starts from $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$: the schedule must make this starting law a good approximation to the actual forward marginal $q(\mathbf{x}_T)$.

19.4. Learn the path back

If the clean $\mathbf{x}_0$ were known, Gaussian conditioning would give the exact posterior for one reverse step:

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\boldsymbol{\beta}}_t \mathbf{I}_d), \quad (19.14)$$

where, with $\bar{\alpha}_0 = 1$,

$$\begin{aligned} \tilde{\boldsymbol{\mu}}_t &= \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0 + \frac{\sqrt{\bar{\alpha}_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t, \\ \tilde{\boldsymbol{\beta}}_t &= \beta_t \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}. \end{aligned} \quad (19.15)$$

During generation—of course— $\mathbf{x}_0$ is the unknown destination. A neural network $\epsilon_\theta(\mathbf{x}_t, t)$ instead predicts the noise coordinate in Equation 19.13. The common reverse-mean parameterization is

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right). \quad (19.16)$$

One widely used training objective samples t , $\mathbf{x}_0$, and ϵ , builds $\mathbf{x}_t$ directly, and minimizes

$$\mathcal{L}_{\text{simple}}(\theta) = \mathbb{E} \left[\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|_2^2 \right]. \quad (19.17)$$

At the population MSE optimum, the network predicts $\mathbb{E}[\epsilon \mid \mathbf{x}_t, t]$. It cannot recover the unknowable realized noise perfectly when several clean examples could have produced the same noisy point. For Gaussian corruption, that conditional mean is also related to the score of the noisy marginal:

$$\nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t) = -\frac{1}{\sqrt{1 - \bar{\alpha}_t}} \mathbb{E}[\epsilon \mid \mathbf{x}_t, t]. \quad (19.18)$$

The score points locally toward higher noisy-data density—generation is not simple gradient ascent and not literal subtraction of one predicted noise image. It follows a time-dependent sequence of learned means plus calibrated stochastic terms.

For a one-dimensional Gaussian mixture with component means μ_j , shared variance s^2 , and mixture weights π_j , the same normalized-weight machine from Chapter 12 appears inside the score:

$$\omega_j(x) = \frac{\pi_j \exp[-(x - \mu_j)^2 / (2s^2)]}{\sum_k \pi_k \exp[-(x - \mu_k)^2 / (2s^2)]}, \quad \frac{\partial}{\partial x} \log p_t(x) = \sum_j \omega_j(x) \frac{\mu_j - x}{s^2}.$$

The coefficients deserve their own audit before any sampler runs.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Pin reverse-posterior coefficients and the final no-noise branch.
-

CODE

```
# [1]
previous_alpha_bar = torch.cat((torch.ones(1), diffusion_alpha_bar[:-1]))
posterior_variance = (
    diffusion_beta * (1.0 - previous_alpha_bar)
    / (1.0 - diffusion_alpha_bar)
)
x0_coefficient = (
    torch.sqrt(previous_alpha_bar) * diffusion_beta
    / (1.0 - diffusion_alpha_bar)
)
xt_coefficient = (
    torch.sqrt(diffusion_alpha) * (1.0 - previous_alpha_bar)
```

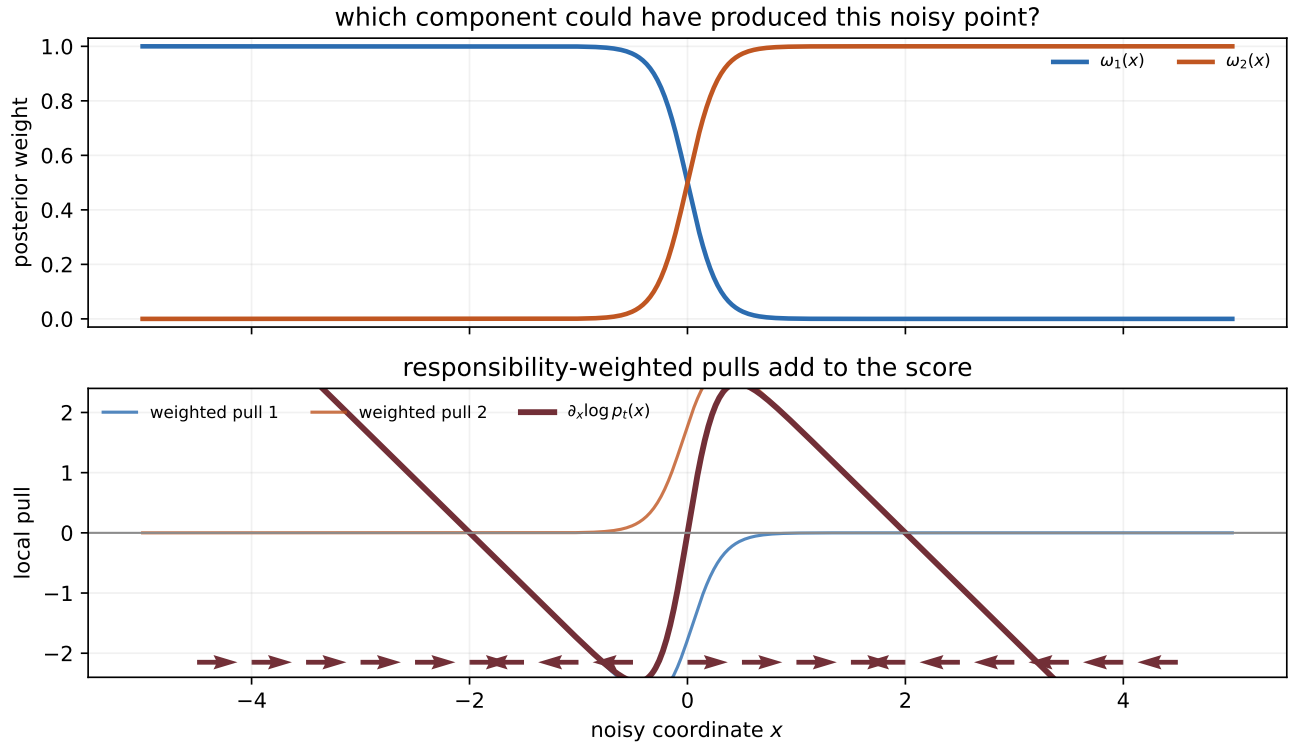



Figure 19.5.: A Gaussian-mixture score as a normalized local pull. The upper panel shows posterior component weights: each location asks which noisy component could have produced it. The lower panel adds the two responsibility-weighted pulls. Their sum is the score, which vanishes at critical points and points locally toward increasing density. This static field explains the score identity; reverse diffusion is a time-dependent stochastic process, not plain gradient ascent on this curve.

```

    / (1.0 - diffusion_alpha_bar)
)

print("t  x0 coefficient  xt coefficient  posterior variance")
for time_index in [1, 2, 10, 50, 100]:
    index = time_index - 1
    print(
        f"{time_index:3d}  {x0_coefficient[index].item():.12f}  "
        f"{xt_coefficient[index].item():.12f}  "
        f"{posterior_variance[index].item():.12f}"
    )
# [2]
assert posterior_variance[0].item() == 0.0
print("t=1 stochastic-noise coefficient:",
      ↪ f"{posterior_variance[0].sqrt().item():.1f}")

```

```

t  x0 coefficient  xt coefficient  posterior variance
1  1.000000000000  0.000000000000  0.000000000000
2  0.917331513473  0.082668472655  0.000091737738
10 0.198099810715  0.801857142882  0.007396541650
50 0.037703866786  0.954855655930  0.048526153318
100 0.007945955048  0.948087682011  0.099937216557
t=1 stochastic-noise coefficient: 0.0

```

At $t = 1$, the exact posterior conditioned on $\mathbf{x}_0$ has zero variance. With the fixed reverse-variance choice $\sigma_t^2 = \tilde{\beta}_t$ used below, the sampler therefore adds no fresh random noise after computing its final mean. That edge case is easy to miss in code.

⚠ Trap: a diffusion derivation can hide approximations

A finite forward endpoint is only approximately normal. The learned reverse transition approximates an unknown marginal reverse law. The simple noise MSE is a particular weighting and simplification of the variational objective, and reverse variance may be fixed or learned. State the schedule, parameterization, variance choice, and starting law before treating “diffusion loss” as one universal object.

19.4.1. Why the denoiser needs time

The same observed value can mean different things at different noise levels. Near the first noising step, a point beside +2 is probably a lightly perturbed right-mode example. Near $t = T$, the same coordinate carries much less evidence about its origin. This is why a diffusion denoiser receives the timestep along with $\mathbf{x}_t$.

We can test that claim without images or a large network. The target is the known two-Gaussian mixture used above. A 4,417-parameter MLP with two 64-unit SiLU hidden layers predicts scalar noise for the 100-step schedule. The time-conditioned version receives (x_t, t) ; the ablation has the identical architecture and seeded initialization but its time channel is fixed at zero, so it sees only x_t . Both train for 5,000 updates on the same generated-data protocol over seeds 6050–6054. Each sampler then draws 20,000 points from a standard-normal endpoint and takes 100 reverse steps with variance $\tilde{\beta}_t$.

The equations count timesteps from $t = 1$, while arrays use zero-based index $j = t - 1$. Model index zero therefore represents the equation's first noising step.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Define the reusable helpers: `ScalarNoisePredictor` and `mixture_batch`.
 3. Define the reusable helpers: `oracle_noise_mean` and `fit_diffusion_ablation`.
 4. Train a tiny diffusion denoiser with and without timestep conditioning.
 5. Report or visualize the measured result.
-

CODE

```
# [1]
training_beta = diffusion_beta.float()
training_alpha = 1.0 - training_beta
training_alpha_bar = torch.cumprod(training_alpha, dim=0)
training_previous_bar = torch.cat((
    torch.ones_like(training_alpha_bar[:1]), training_alpha_bar[:-1]
))
training_posterior_variance = (
    training_beta * (1.0 - training_previous_bar)
    / (1.0 - training_alpha_bar)
)

# [2]
class ScalarNoisePredictor(nn.Module):
    def __init__(self, time_conditioned: bool) -> None:
        super().__init__()
        self.time_conditioned = time_conditioned
        self.network = nn.Sequential(
            nn.Linear(2, 64),
            nn.SiLU(),
            nn.Linear(64, 64),
            nn.SiLU(),
            nn.Linear(64, 1),
        )

    def forward(self, x_t: Tensor, timestep: Tensor) -> Tensor:
        if self.time_conditioned:
            scaled_time = 2.0 * timestep[:, None].float() / (diffusion_steps - 1) -
            ↪ 1.0
        else:
            scaled_time = torch.zeros_like(x_t)
        model_input = torch.cat((x_t, scaled_time), dim=1)
        return self.network(model_input)
```

```

torch.manual_seed(6050)
conditioned_initialization = ScalarNoisePredictor(True)
torch.manual_seed(6050)
no_time_initialization = ScalarNoisePredictor(False)
assert sum(p.numel() for p in conditioned_initialization.parameters()) == 4_417
assert all(
    torch.equal(conditioned, no_time)
    for conditioned, no_time in zip(
        conditioned_initialization.state_dict().values(),
        no_time_initialization.state_dict().values(),
    )
)
del conditioned_initialization, no_time_initialization

def mixture_batch(generator: torch.Generator, batch_size: int) -> Tensor:
    component = 2 * torch.randint(0, 2, (batch_size, 1), generator=generator) - 1
    noise = torch.randn(batch_size, 1, generator=generator, dtype=torch.float32)
    return 2.0 * component.float() + 0.5 * noise

# [3]
def oracle_noise_mean(x_t: Tensor, timestep: Tensor) -> Tensor:
    alpha_bar_at_time = training_alpha_bar[timestep, None]
    root_bar = torch.sqrt(alpha_bar_at_time)
    noisy_variance = 0.25 * alpha_bar_at_time + 1.0 - alpha_bar_at_time
    component_means = torch.cat((-2.0 * root_bar, 2.0 * root_bar), dim=1)
    log_weights = -0.5 * (x_t - component_means).square() / noisy_variance
    weights = torch.softmax(log_weights, dim=1)
    posterior_gain = 0.25 * root_bar / noisy_variance
    clean_component_means = torch.cat((
        -2.0 + posterior_gain * (x_t + 2.0 * root_bar),
        2.0 + posterior_gain * (x_t - 2.0 * root_bar),
    ), dim=1)
    expected_x0 = (weights * clean_component_means).sum(dim=1, keepdim=True)
    return (x_t - root_bar * expected_x0) / torch.sqrt(1.0 - alpha_bar_at_time)

def fit_diffusion_ablation(
    seed: int, time_conditioned: bool
) -> tuple[dict[str, float], Tensor]:
    torch.manual_seed(seed)
    train_generator = torch.Generator().manual_seed(seed + 19_000)
    model = ScalarNoisePredictor(time_conditioned).float()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.002)

    for _ in range(5_000):

```

```

clean = mixture_batch(train_generator, 512)
timestep = torch.randint(diffusion_steps, (512,), generator=train_generator)
epsilon = torch.randn(512, 1, generator=train_generator, dtype=torch.float32)
alpha_bar_at_time = training_alpha_bar[timestep, None]
noisy = (
    torch.sqrt(alpha_bar_at_time) * clean
    + torch.sqrt(1.0 - alpha_bar_at_time) * epsilon
)
loss = (model(noisy, timestep) - epsilon).square().mean()
optimizer.zero_grad()
loss.backward()
optimizer.step()

evaluation_generator = torch.Generator().manual_seed(91_900)
evaluation_clean = mixture_batch(evaluation_generator, 50_000)
evaluation_timestep = torch.randint(
    diffusion_steps, (50_000,), generator=evaluation_generator
)
evaluation_epsilon = torch.randn(
    50_000, 1, generator=evaluation_generator, dtype=torch.float32
)
evaluation_bar = training_alpha_bar[evaluation_timestep, None]
evaluation_noisy = (
    torch.sqrt(evaluation_bar) * evaluation_clean
    + torch.sqrt(1.0 - evaluation_bar) * evaluation_epsilon
)

sampling_generator = torch.Generator().manual_seed(seed + 29_000)
generated = torch.randn(
    20_000, 1, generator=sampling_generator, dtype=torch.float32
)

with torch.no_grad():
    evaluation_prediction = model(evaluation_noisy, evaluation_timestep)
    evaluation_mse = (
        evaluation_prediction - evaluation_epsilon
    ).square().mean().item()
    oracle_mse = (
        oracle_noise_mean(evaluation_noisy, evaluation_timestep)
        - evaluation_epsilon
    ).square().mean().item()

for index in range(diffusion_steps - 1, -1, -1):
    timestep = torch.full((generated.shape[0],), index, dtype=torch.long)
    predicted_epsilon = model(generated, timestep)
    reverse_mean = (
        generated
        - training_beta[index]
        / torch.sqrt(1.0 - training_alpha_bar[index])
        * predicted_epsilon
    )

```

```

    ) / torch.sqrt(training_alpha[index])
    if index > 0:
        reverse_noise = torch.randn(
            generated.shape, generator=sampling_generator,
            dtype=torch.float32,
        )
    else:
        reverse_noise = torch.zeros_like(generated)
    generated = (
        reverse_mean
        + torch.sqrt(training_posterior_variance[index]) * reverse_noise
    )

    target_generator = torch.Generator().manual_seed(seed + 39_000)
    target = mixture_batch(target_generator, 20_000)[: , 0]
    generated_vector = generated[: , 0]
    wasserstein_one = (
        torch.sort(generated_vector).values - torch.sort(target).values
    ).abs().mean().item()
    metrics = {
        "noise MSE": evaluation_mse,
        "oracle MSE": oracle_mse,
        "mean": generated_vector.mean().item(),
        "standard deviation": generated_vector.std(unbiased=False).item(),
        "positive mass": (generated_vector > 0).float().mean().item(),
        "central mass": (generated_vector.abs() < 1).float().mean().item(),
        "Wasserstein-1": wasserstein_one,
    }
    return metrics, generated_vector

torch.set_num_threads(1)
diffusion_results: dict[str, list[dict[str, float]]] = {
    "time conditioned": [], "no time": []
}
example_samples = {}
# [4]
conditions = [("time conditioned", True), ("no time", False)]
for condition_name, time_conditioned in conditions:
    for diffusion_seed in range(6050, 6055):
        metrics, generated_samples = fit_diffusion_ablation(
            diffusion_seed, time_conditioned
        )
        diffusion_results[condition_name].append(metrics)
        if diffusion_seed == 6050:
            example_samples[condition_name] = generated_samples
torch.set_num_threads(6)

true_standard_deviation = math.sqrt(4.25)

```

```

normal_cdf = lambda value: 0.5 * (1.0 + math.erf(value / math.sqrt(2.0)))
true_central_mass = normal_cdf(-2.0) - normal_cdf(-6.0)

# [5]
print("condition      metric      mean      sample SD")
for condition_name, rows in diffusion_results.items():
    for metric_name in [
        "noise MSE", "oracle MSE", "mean", "standard deviation",
        "positive mass", "central mass", "Wasserstein-1",
    ]:
        values = torch.tensor([row[metric_name] for row in rows])
        print(
            f"{condition_name:17s} {metric_name:18s} "
            f"{values.mean().item():.9f} {values.std().item():.9f}"
        )
print(f"target standard deviation: {true_standard_deviation:.9f}")
print(f"target positive mass:      {0.5:.9f}")
print(f"target central mass:       {true_central_mass:.9f}")
print(f"alpha_bar_T:                {training_alpha_bar[-1].item():.12f}")

```

condition	metric	mean	sample SD
time conditioned	noise MSE	0.429707128	0.000926689
time conditioned	oracle MSE	0.425920248	0.000000000
time conditioned	mean	0.034373117	0.056509078
time conditioned	standard deviation	2.062094545	0.026976564
time conditioned	positive mass	0.506739992	0.010188865
time conditioned	central mass	0.019470000	0.002502649
time conditioned	Wasserstein-1	0.058569731	0.042296170
no time	noise MSE	0.750688660	0.000319157
no time	oracle MSE	0.425920248	0.000000000
no time	mean	-0.001286332	0.066899247
no time	standard deviation	1.778861904	0.025886268
no time	positive mass	0.498869997	0.012373481
no time	central mass	0.223489997	0.009864037
no time	Wasserstein-1	0.379805672	0.024180092
target standard deviation: 2.061552813			
target positive mass: 0.500000000			
target central mass: 0.022750131			
alpha_bar_T: 0.005618760828			

The time-conditioned network nearly matches the fixed-evaluation MSE of the analytic conditional-mean oracle and recovers the mixture's two-mode geometry. With its time channel zeroed, the identically initialized network must use one answer for incompatible noise regimes. It places about ten times too much mass in the low-density interval $|x| < 1$. The experiment isolates one design requirement; it does not rank diffusion against VAEs or GANs.

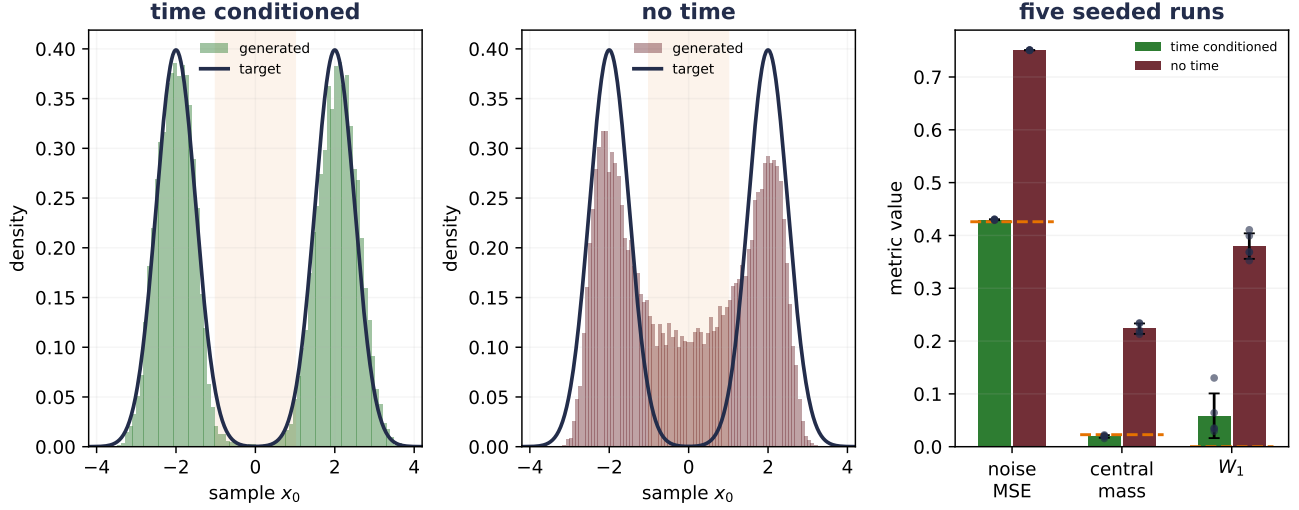


Figure 19.6.: A designed one-dimensional diffusion sampler isolates the value of time conditioning with identical 4,417-parameter networks and seeded initial weights. Across seeds 6050–6054, the time-conditioned denoiser has evaluation noise MSE 0.429707 (SD 0.000927), close to the analytic conditional-mean oracle’s fixed-evaluation MSE of 0.425920. Its generated samples have mean standard deviation 2.062095, central mass 0.019470, and Wasserstein-1 distance 0.058570 from a matched target sample; the population targets are 2.061553 and 0.022750 for standard deviation and central mass. Zeroing the time channel raises noise MSE to 0.750689, central mass to 0.223490, and Wasserstein distance to 0.379806. This is a CPU mechanism study of a planted scalar mixture, not an image-generation result.

19.5. From a scalar denoiser to structured data

The scalar network makes the learning problem visible—yet image, audio, and video models must predict arrays with spatial or temporal structure. A diffusion network usually receives three kinds of information:

1. the noisy state $\mathbf{x}_t$;
2. an embedding of the timestep or noise level; and
3. optional conditioning information $\mathbf{c}$, such as a class, text representation, or another observed modality.

When the condition is a sequence, cross-attention reopens the routing machinery from [Chapter 13](#) and [Chapter 14](#). Let noisy-state features be $\mathbf{H}_t \in \mathbb{R}^{n_x \times d}$, condition features be $\mathbf{H}_c \in \mathbb{R}^{n_c \times d}$, and each projection matrix have shape $d \times d_h$. Then

$$\mathbf{Q} = \mathbf{H}_t \mathbf{W}_Q, \quad \mathbf{K} = \mathbf{H}_c \mathbf{W}_K, \quad \mathbf{V} = \mathbf{H}_c \mathbf{W}_V.$$

The noisy representation supplies the queries; the condition supplies keys and values. One common combination is **classifier-free guidance**. With the convention $w \geq 0$,

$$\tilde{\epsilon}_\theta(\mathbf{x}_t, t, \mathbf{c}) = (1 + w)\epsilon_\theta(\mathbf{x}_t, t, \mathbf{c}) - w\epsilon_\theta(\mathbf{x}_t, t, \emptyset).$$

At $w = 0$ this is the conditional prediction; increasing w amplifies the direction that distinguishes conditional from unconditional denoising. Stronger guidance can improve condition adherence while reducing diversity or exposing calibration errors, so the convention, scale sweep, and trade-off belong in the evaluation contract.

For images, a U-Net is a natural denoiser. Its contracting path gathers broad context—its expanding path returns to the original resolution. Skip connections carry fine-resolution features across the bottleneck so that global and local evidence can meet. This is the same architectural issue we met in [Chapter 11](#): a fixed summary alone can discard detail that the output still needs. The diffusion-specific addition is time conditioning throughout the network.

Conditioning changes the target from $p_\theta(\mathbf{x})$ to $p_\theta(\mathbf{x} \mid \mathbf{c})$. It does not remove the need to test the unconditional support, the quality of the conditioning data, or whether the sampler follows $\mathbf{c}$ for the intended population. A strong condition score can coexist with repeated, memorized, or low-coverage samples.

There is also a clean composition of the autoencoder interlude and this chapter’s diffusion model. A pretrained autoencoder can map high-dimensional observations into a lower-dimensional latent space. Diffusion then learns a distribution over those codes, and the decoder maps a sampled code back to observation space:

$$\mathbf{x} \xrightarrow{f_\phi} \mathbf{z}_0 \xrightarrow{\text{forward noise / learned reverse}} \tilde{\mathbf{z}}_0 \xrightarrow{g_\theta} \tilde{\mathbf{x}}.$$

This is the mechanism behind **latent diffusion**. It separates two approximation bills: the autoencoder may discard information, and the latent diffusion model may miss parts of the code distribution. Running the reverse process in a smaller space changes the computational problem, but no unmeasured speed ratio follows from the diagram.

19.6. A generator needs an evaluation contract too

The original data distribution is unknown; that is why we are learning it. No single training loss can therefore certify generation. A useful evaluation contract separates at least these questions:

- **Fidelity:** do samples lie near supported data regions and satisfy domain-specific constraints?
- **Coverage:** are important modes, subgroups, or rare cases missing?
- **Memorization:** are outputs new combinations or near-copies of training examples?
- **Condition adherence:** for conditional models, does the sample match the supplied context without sacrificing the first three properties?
- **Distributional fit:** when a likelihood, ELBO, or other bound is available, on which held-out data and under which measurement assumptions was it computed?
- **Sampling cost and uncertainty:** how many sequential steps, what decoding choices, and how much variation across training and sampling seeds?
- **Data and use context:** whose data supplied the distribution, with what license, consent, provenance, and known bias; what misuse risks and disclosure or watermarking limits remain?

Our finite studies made some of these quantities observable by design. The GAN example had three named modes, so missing mass was visible. The diffusion mixture had an exact central probability and a one-dimensional Wasserstein distance. Natural data needs domain-appropriate measurements and

human inspection, with sample counts and uncertainty. A favorable metric is evidence under its own scope, not a declaration that $p_\theta = p_{\text{data}}$.

Family	What training fits	How a new sample begins	Density information	Characteristic audit
PCA / deterministic AE	Reconstruction on encoded observations	No built-in random start	None by itself	Held-out reconstruction; behavior between and outside codes
VAE	ELBO: expected decoder log-likelihood minus posterior-to-prior KL	$\mathbf{z} \sim p(\mathbf{z})$, then decoder	A latent-variable density, commonly evaluated through a bound	Held-out ELBO, prior-sample quality, collapse, coverage; exact gap only when the true posterior is known
GAN	A moving discriminator–generator game	$\mathbf{z} \sim p_z$, then one generator pass	Usually implicit rather than tractable	Mode coverage, optimization balance, memorization
Diffusion	Denoising or score predictions across a noise schedule	$\mathbf{x}_T$ from a declared terminal law, then reverse steps	Variational or score-based connections; simple MSE is not itself likelihood	Terminal match, schedule coefficients, reverse calibration, coverage, step cost

Generation and judgment are separate contracts. A generator tells us how to draw a sample; a judge or metric tells us how that draw will be evaluated. A complete system may need both, but neither makes the other correct.

i Designed studies are measuring instruments

The interlude’s curve and three-code decoder, together with this chapter’s finite GAN, Gaussian ELBO, and scalar diffusion mixture, were chosen so that reconstruction, bound gaps, mode mass, and schedule coefficients could be checked exactly. They expose mechanisms and failure boundaries. They do not estimate natural-image quality, settle model-family rankings, or substitute for an application-specific evaluation contract.

One final numerical seed is harvested in Appendix C: **a small schedule coefficient is not a zero coefficient**. Long products such as $\bar{\alpha}_t = \prod_{s \leq t} \alpha_s$ must be audited in the dtype that will carry them. Mathematical smallness, floating-point underflow, and a rounded-away update are different failures. For long schedules, accumulate log coefficients when appropriate and test the endpoint actually used by the sampler.

i Check yourself

Close the book for one minute and rebuild the sampling contracts from memory.

- Why can a discriminator score samples but not, by itself, generate them?

- What extra contract turns a reconstruction model into a sampler?
- Why must a diffusion denoiser know the timestep?

19.7. Okay, so — generation needs a sampling contract

1. **A judge is not a generator.** Scoring completed samples and defining a law that produces new samples are different contracts.
2. **A code is not yet a distribution.** The autoencoder interlude supplied a decoder and exposed the missing random start; a generative model must add a latent law or another sampling procedure.
3. **A VAE supplies a probability contract.** Its ELBO is expected decoder log likelihood minus prior KL, and the exact gap to log evidence is posterior KL. Reparameterization moves randomness outside the learned map; it does not remove it.
4. **A GAN lets a judge train a sampler.** The ideal discriminator yields a Jensen–Shannon identity, while finite networks still face saturation, coupled optimization, and mode-coverage failures.
5. **Diffusion learns denoising across calibrated noise levels.** The forward marginal is available in closed form, the reverse coefficients must be pinned, and the denoiser needs the timestep. A finite endpoint and learned reverse law remain approximations that must be audited.
6. **Generation needs external evaluation.** Fidelity, coverage, memorization, conditioning, distributional fit, cost, and seed uncertainty answer different questions; no one score certifies them all.

Sources and further reading

- Kingma and Welling, *Auto-Encoding Variational Bayes*: variational latent models and reparameterized gradient estimation.
- Higgins et al., *beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*: deliberate KL reweighting.
- Locatello et al., *Challenging Common Assumptions in the Unsupervised Learning of Disentangled Representations*: identifiability limits without additional inductive biases.
- Goodfellow et al., *Generative Adversarial Nets*: the original minimax game, optimal discriminator, and divergence result.
- Ho, Jain, and Abbeel, *Denoising Diffusion Probabilistic Models*: Gaussian diffusion transitions, reverse parameterization, and the simplified noise objective.
- Song et al., *Score-Based Generative Modeling through Stochastic Differential Equations*: the score perspective and continuous-time forward/reverse processes.
- Ho and Salimans, *Classifier-Free Diffusion Guidance*: combines conditional and unconditional denoiser predictions without a separate classifier.
- Chen et al., *Isolating Sources of Disentanglement in Variational Autoencoders*: the total-correlation decomposition whose aggregate-posterior KL is Exercise 5’s boundary case — the term that does not batch naively.
- Rombach et al., *High-Resolution Image Synthesis with Latent Diffusion Models*: composing learned compression with diffusion in latent space.

Exercises

1. **(Pencil.)** (a) Derive the posterior and marginal evidence for the scalar Gaussian VAE audit. **(Code.)** (b) Verify Equation 19.3 for three approximate posteriors. Then use reparameterized samples to estimate its gradient with respect to the approximate posterior mean; compare it with the analytic gradient and report Monte Carlo uncertainty.
2. **(Code.)** Implement Equation 19.13 for a batch whose timesteps have shape $(B,)$. Gather coefficients into shape $(B, 1)$ before multiplying data of shape (B, d) . Compare iterative and direct moments, verify Equation 19.15, and test the $t = 1$ no-noise branch. Compare two schedules without promising which will train better.
3. **(Pencil.)** Maximize the GAN value pointwise to derive $D^*(x) = p(x)/(p(x)+q(x))$, then substitute it back and recover $2 \text{JSD}(p\|q) - \log 4$. **(Code.)** Verify the identity by numerical integration for two one-dimensional Gaussians as the mean of q moves toward the mean of p . **(Audit.)** Explain why an observed $D \approx 0.5$ does not prove convergence, then design separate fidelity, coverage, and memorization checks for supplied samples from a VAE, GAN, and diffusion model. State the boundary explicitly: the divergence identity describes the objective at the inner optimum, not the dynamics that attempt to reach it.
4. **(Audit.)** A colleague’s β -VAE uses `F.mse_loss(recon, x) + beta * kl.mean()`, with `kl` holding one KL per example. Before any algebra, predict the *direction* of the defect: which term was silently divided by what? Then prove the identity — with default `mean` reductions the loss equals $\frac{1}{D_x} [\text{recon}_{\text{sum}} + (\beta D_x) \text{KL}]$ exactly, so a nominal $\beta = 1$ on 28×28 inputs trains an effective $\beta = 784$ — posterior collapse by construction. *Named wrong answer:* “the two `means` cancel, so the trade-off is unchanged” — they average over different axis sets ($B \cdot D_x$ versus B), and the mismatch is the bug.
5. **(Audit.)** Now audit a “diversity” term that penalizes $\text{KL}(q_{\text{agg}} \| p)$ by computing it on each batch’s mixture and averaging. Which of the three estimator cases is this, and in which direction is the batch estimate biased? Verify numerically on synthetic Gaussian posteriors: KL is convex in its first argument and a size- B batch mixture is a noisier q_{agg} , so Jensen drives the estimate *upward* (in one run of this audit: true 0.100, batch estimate 0.148 ± 0.006 , while the per-example mean KL stayed unbiased). *Named wrong answer:* “it averages out over batches like any minibatch loss” — averaging removes noise around a biased center, not the bias.

20. Multimodal Learning: One Space, Two Views

We have learned vision and language in separate coordinate systems. Chapter 16 turned an image into patch tokens. Chapters 14 and 15 turned text into contextual token representations. Those encoders can each organize their own modality—but nothing says that an image vector and a text vector mean the same thing merely because they have the same length.

Let us make the problem concrete. Suppose row 7 of an image table and row 7 of a text table describe the same event. The image has 14 measured coordinates; the text has 11. We do not need either raw coordinate system to imitate the other. We need two learned maps that preserve the *pair relation*:

$$\text{image view} \longrightarrow \text{shared coordinates} \longleftarrow \text{text view}.$$

This is the central move in **multimodal representation learning**. Different modalities keep different encoders, while paired observations teach those encoders how to become comparable. The learned space can support cross-modal retrieval and, under a carefully bounded candidate construction, zero-shot classification. It does not by itself create a caption, synthesize an image, or prove that the representation has captured every meaning we care about—that boundary will matter throughout the chapter.

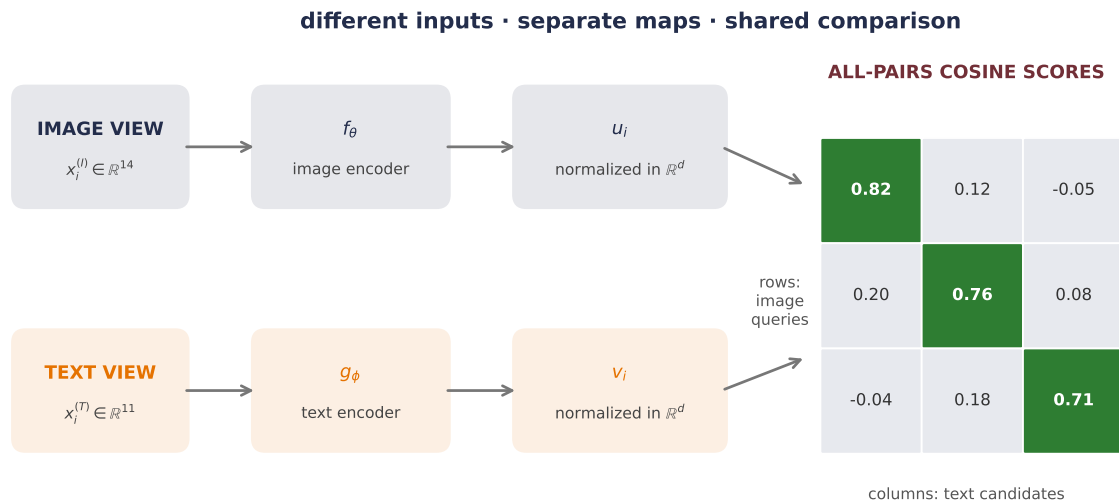


Figure 20.1.: Two modalities enter different encoders and leave as normalized vectors in one comparison space. For a batch of three declared pairs, every image is scored against every text; the green diagonal marks the three pair relations supplied by the data.

The small matrix already tells us what the training problem must do. Each row is one image query. Each column is one text candidate. Row 1 should select column 1, row 2 should select column 2, and so on. We have seen this machine before.

20.1. Chapter 2 returns: scores become weights

Let a minibatch contain B declared pairs $(x_i^{(I)}, x_i^{(T)})$. The image encoder $f_\theta : \mathbb{R}^{d_I} \rightarrow \mathbb{R}^d$ and text encoder $g_\phi : \mathbb{R}^{d_T} \rightarrow \mathbb{R}^d$ may have completely different internal architectures. Their interiors may differ completely—their outputs must agree only in the last dimension. We normalize each output to unit length:

$$u_i = \frac{f_\theta(x_i^{(I)})}{\|f_\theta(x_i^{(I)})\|_2}, \quad v_i = \frac{g_\phi(x_i^{(T)})}{\|g_\phi(x_i^{(T)})\|_2}, \quad u_i, v_i \in \mathbb{R}^d. \quad (20.1)$$

Now the dot product is cosine similarity. For every cross-modal pair in the batch, form

$$S_{ij} = \frac{u_i^\top v_j}{\tau}, \quad \mathbf{S} \in \mathbb{R}^{B \times B}, \quad (20.2)$$

where $\tau > 0$ is the **temperature**. Dividing by a small temperature spreads the logits farther apart; a larger temperature softens them. Temperature does not decide which pair is correct—it decides how sharply the softmax reacts to score differences.

i What if the temperature were learnable?

Chapter 12 read temperature as bandwidth: for its Gaussian score, $\tau = 2h^2$. Chapter 13 learned the comparison itself while retaining a fixed scale. Here one scalar can move too. Write the inverse temperature as $a = e^\gamma = 1/\tau$:

$$S_{ij} = e^\gamma u_i^\top v_j.$$

The encoders learn where the two modalities sit; γ learns how sharply a given angular gap should count. CLIP trained this log scale jointly with its two towers and initialized it at the equivalent of $\tau = 0.07$. The paper reports capping e^γ at 100 for stability. That cap is a training-recipe choice: the released reference model initializes and exponentiates the parameter but does not apply the cap.

This scalar belongs inside the training objective. It is not the post-hoc temperature scaling in [Chapter 16](#), Exercise 6, where the model is frozen and validation labels calibrate its confidence after training.

Chapter 2 introduced softmax as the **scores-to-weights machine**. Apply it across row i :

$$p_{ij}^{I \rightarrow T} = \frac{\exp(S_{ij})}{\sum_{k=1}^B \exp(S_{ik})}. \quad (20.3)$$

The declared mate of image i is text i , so the image-to-text loss is ordinary cross-entropy with labels $0, 1, \dots, B-1$:

$$\mathcal{L}_{I \rightarrow T} = -\frac{1}{B} \sum_{i=1}^B \log p_{ii}^{I \rightarrow T}. \quad (20.4)$$

This form is commonly called a contrastive or **InfoNCE-style** objective. The batch does two jobs at once—it supplies B declared matches along the diagonal, and for each query it supplies $B - 1$ alternative candidates off the diagonal. We do not need to write a separate binary loss for every pair—the score matrix organizes the whole comparison. And note which case this is in [Chapter 4](#)’s taxonomy: **Case 3**. The batch is not estimating some full-data contrastive loss — the $B - 1$ candidates *define* the objective, so changing B changes what is being optimized, and batch size here is protocol, reported like any other design choice.

But one direction asks only whether each image can choose its text. Turn the score matrix around and ask whether each text can choose its image:

$$\mathcal{L}_{T \rightarrow I} = -\frac{1}{B} \sum_{j=1}^B \log \frac{\exp(S_{jj})}{\sum_{k=1}^B \exp(S_{kj})}. \quad (20.5)$$

The **symmetric batch contrastive loss** is

$$\mathcal{L}_{\text{sym}} = \frac{1}{2} (\mathcal{L}_{I \rightarrow T} + \mathcal{L}_{T \rightarrow I}). \quad (20.6)$$

Notice the exact reuse: Equation 20.3 is Chapter 2’s multiclass classifier, but the classes are the candidates currently in the batch. The class set changes with the batch—what remains fixed is the rule that the paired row and column share an index.

i Paired data is supervision

No manual object class is required, but the correspondence $(x_i^{(I)}, x_i^{(T)})$ is still a training label. In web data that relation may be weak, incomplete, or wrong. Calling the procedure “self-supervised” must not erase the work, noise, consent, or provenance involved in creating those pairs.

20.1.1. What counts as a negative?

The loss knows **declared mates**—not semantic truth. An off-diagonal caption may describe the same concept as the diagonal caption. Two photographs may show the same event. In those cases the standard one-positive loss treats a plausible match as an alternative to suppress. Larger batches provide more alternatives—but they can also provide more false negatives.

This distinction is easy to miss because the code is so compact:

PLAN

1. Encode both modalities and normalize them into one comparison space.
 2. Score every image against every text at the chosen temperature.
 3. Declare the batch diagonal as the supplied pair relation.
 4. Average image-to-text and text-to-image cross-entropy.
-

CODE

```
def symmetric_contrastive_loss(
    image: Tensor,
    text: Tensor,
    image_encoder: nn.Module,
    text_encoder: nn.Module,
    temperature: float,
) -> Tensor:
    # [1]
    image_embedding = F.normalize(image_encoder(image), dim=-1)
    text_embedding = F.normalize(text_encoder(text), dim=-1)
    # [2]
    logits = image_embedding @ text_embedding.T / temperature
    # [3]
    labels = torch.arange(logits.shape[0], device=logits.device)
    # [4]
    return 0.5 * (
        F.cross_entropy(logits, labels)
        + F.cross_entropy(logits.T, labels)
    )

identity_batch = torch.eye(3)
loss_check = symmetric_contrastive_loss(
    identity_batch, identity_batch, nn.Identity(), nn.Identity(), 0.1
)
assert torch.isfinite(loss_check)
```

Every line has a contract. The two encoders end at the same width; normalization makes the dot product angular; the transpose reverses retrieval direction; and the target indices assert that the batch diagonal contains the pair relation. If the data loader shuffles one modality without shuffling the other, the program still runs—the supervision has simply become false.

20.2. Make the pair relation learnable

Let us test that last claim directly. We will not download images or tokenize a corpus. Instead, one hidden five-dimensional event generates two noisy nonlinear views: a 14-coordinate “image” measurement and an 11-coordinate “text” measurement. The raw features differ, but row i in one table really does share a latent event with row i in the other.

The protocol is fixed before the endpoint is opened:

- 1,200 unique pairs are generated once, then split into 720 fit, 180 validation, and 300 held-out endpoint rows. Standardization uses fit statistics only—the validation and endpoint rows contribute no preprocessing estimates.
- Each tower is a two-layer MLP and ends at a normalized 16-dimensional embedding. The complete model has 2,392 parameters.
- Five model seeds, 6050–6054, share the same paired initialization and minibatch schedule between conditions. AdamW, 160 epochs, batch size 120, and $\tau = 0.08$ are fixed.

- The control replaces the fit-pair relation with one study-wide fixed derangement—every fit image is assigned the wrong text, with no fixed points. Architecture, initialization, optimizer, update count, and schedules stay matched.
- Both conditions select checkpoints by the correct validation-pair loss. The 300-row endpoint is not used for model, hyperparameter, checkpoint, stopping, or analysis-design decisions within this study; it is consulted after those choices are fixed.

The metric is **Recall@ k** —for each query, rank every candidate in the other modality. Recall@ k is one when the declared mate lies among the first k candidates and zero otherwise, then averaged across queries. With 300 unique candidates, random ranking has expected Recall@1 of $1/300 = 0.0033$ and Recall@5 of $5/300 = 0.0167$.

PLAN

1. Define the reusable helpers: `PairedSplit`, `make_paired_data`, `TwoTower`, `symmetric_loss`, and `pair_loss`.
 2. Define the reusable helpers: `retrieval_metrics`, `derangement`, `train_tower`, `report`, and `report_paired_contrast`.
 3. Prepare the inputs and fixed settings for the example.
 4. Run the five-seed paired-versus-shuffled retrieval study.
 5. Report or visualize the measured result.
-

CODE

```
# [1]
@dataclass(frozen=True)
class PairedSplit:
    image: Tensor
    text: Tensor

def make_paired_data() -> tuple[PairedSplit, PairedSplit, PairedSplit]:
    generator = torch.Generator().manual_seed(1206050)
    latent_dim, image_dim, text_dim = 5, 14, 11
    latent = torch.randn(1_200, latent_dim, generator=generator)
    image_map = torch.randn(latent_dim, image_dim, generator=generator)
    text_map = torch.randn(latent_dim, text_dim, generator=generator)

    image = torch.tanh(latent @ image_map / latent_dim**0.5)
    text = torch.tanh(latent @ text_map / latent_dim**0.5)
    image += 0.045 * torch.randn(image.shape, generator=generator)
    text += 0.045 * torch.randn(text.shape, generator=generator)

    fit_end, validation_end = 720, 900
    image_mean = image[:fit_end].mean(0, keepdim=True)
    image_std = image[:fit_end].std(0, keepdim=True).clamp_min(1e-6)
    text_mean = text[:fit_end].mean(0, keepdim=True)
    text_std = text[:fit_end].std(0, keepdim=True).clamp_min(1e-6)
```

```

image = (image - image_mean) / image_std
text = (text - text_mean) / text_std

return (
    PairedSplit(image[:fit_end], text[:fit_end]),
    PairedSplit(image[fit_end:validation_end], text[fit_end:validation_end]),
    PairedSplit(image[validation_end:], text[validation_end:]),
)

class TwoTower(nn.Module):
    def __init__(
        self, image_dim: int, text_dim: int,
        width: int = 40, embed_dim: int = 16,
    ):
        super().__init__()
        self.image_encoder = nn.Sequential(
            nn.Linear(image_dim, width), nn.ReLU(), nn.Linear(width, embed_dim)
        )
        self.text_encoder = nn.Sequential(
            nn.Linear(text_dim, width), nn.ReLU(), nn.Linear(width, embed_dim)
        )

    def forward(self, image: Tensor, text: Tensor) -> tuple[Tensor, Tensor]:
        image_embedding = F.normalize(self.image_encoder(image), dim=-1)
        text_embedding = F.normalize(self.text_encoder(text), dim=-1)
        return image_embedding, text_embedding

    def symmetric_loss(
        image_embedding: Tensor, text_embedding: Tensor, tau: float = 0.08,
    ) -> Tensor:
        logits = image_embedding @ text_embedding.T / tau # (B, B)
        labels = torch.arange(logits.shape[0])
        return 0.5 * (
            F.cross_entropy(logits, labels) + F.cross_entropy(logits.T, labels)
        )

    @torch.no_grad()
    def pair_loss(model: TwoTower, split: PairedSplit) -> float:
        model.eval()
        image_embedding, text_embedding = model(split.image, split.text)
        return float(symmetric_loss(image_embedding, text_embedding))

# [2]
@torch.no_grad()
def retrieval_metrics(model: TwoTower, split: PairedSplit) -> dict[str, float]:

```

```

model.eval()
image_embedding, text_embedding = model(split.image, split.text)
scores = image_embedding @ text_embedding.T
target = torch.arange(scores.shape[0])

def recall_at_k(matrix: Tensor, k: int) -> float:
    candidates = matrix.topk(k, dim=1).indices
    return float((candidates == target[:, None]).any(dim=1).float().mean())

return {
    "I→T R01": recall_at_k(scores, 1),
    "T→I R01": recall_at_k(scores.T, 1),
    "I→T R05": recall_at_k(scores, 5),
    "T→I R05": recall_at_k(scores.T, 5),
}

def derangement(size: int, generator: torch.Generator) -> Tensor:
    target = torch.arange(size)
    while True:
        order = torch.randperm(size, generator=generator)
        if not bool((order == target).any()):
            return order

def train_tower(
    seed: int,
    fit_split: PairedSplit,
    validation_split: PairedSplit,
    shuffled_pairs: bool,
) -> TwoTower:
    torch.manual_seed(seed)
    model = TwoTower(fit_split.image.shape[1], fit_split.text.shape[1])
    optimizer = torch.optim.AdamW(model.parameters(), lr=2e-3, weight_decay=1e-3)

    pair_generator = torch.Generator().manual_seed(88_000)
    text_order = (
        derangement(len(fit_split.text), pair_generator)
        if shuffled_pairs else torch.arange(len(fit_split.text))
    )
    schedule_generator = torch.Generator().manual_seed(99_000 + seed)
    schedules = [
        torch.randperm(len(fit_split.image), generator=schedule_generator)
        for _ in range(160)
    ]

    best_validation = float("inf")
    best_state: dict[str, Tensor] | None = None
    for epoch, order in enumerate(schedules, start=1):

```

```

model.train()
for rows in order.split(120):
    image_embedding, text_embedding = model(
        fit_split.image[rows], fit_split.text[text_order[rows]]
    )
    loss = symmetric_loss(image_embedding, text_embedding)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

if epoch % 5 == 0:
    validation_loss = pair_loss(model, validation_split)
    if validation_loss < best_validation:
        best_validation = validation_loss
        best_state = copy.deepcopy(model.state_dict())

assert best_state is not None
model.load_state_dict(best_state)
return model

# [3]
fit_split, validation_split, held_out_split = make_paired_data()
models: dict[tuple[int, str], TwoTower] = {}
records: list[dict[str, object]] = []

# [4]
for seed in range(6050, 6055):
    for condition, shuffled_pairs in (("paired", False), ("shuffled", True)):
        model = train_tower(seed, fit_split, validation_split, shuffled_pairs)
        models[(seed, condition)] = model
        records.append({
            "seed": seed,
            "condition": condition,
            "validation": retrieval_metrics(model, validation_split),
        })

# The endpoint is consulted after every design and checkpoint decision above.
for record in records:
    model = models[(int(record["seed"]), str(record["condition"]))]
    record["held_out"] = retrieval_metrics(model, held_out_split)

def report(split_name: str, key: str) -> None:
    print(split_name)
    for condition in ("paired", "shuffled"):
        chosen = [record for record in records if record["condition"] == condition]
        fields = list(chosen[0][key].keys())
        summary = []

```

```

    for field in fields:
        values = [float(record[key][field]) for record in chosen]
        summary.append(f"{field} {mean(values):.4f} (SD {stdev(values):.4f})")
    print(f" {condition:8s}: " + "; ".join(summary))

def report_paired_contrast(field: str) -> None:
    differences = []
    for seed in range(6050, 6055):
        paired = next(
            record for record in records
            if record["seed"] == seed and record["condition"] == "paired"
        )
        shuffled = next(
            record for record in records
            if record["seed"] == seed and record["condition"] == "shuffled"
        )
        differences.append(
            float(paired["held_out"][field]) - float(shuffled["held_out"][field])
        )
    print(
        f"paired-shuffled held-out {field}: "
        f"{mean(differences):.4f} (paired SD {stdev(differences):.4f})"
    )

reference_model = models[(6050, "paired")]
parameter_count = sum(p.numel() for p in reference_model.parameters())
# [5]
split_sizes = (
    len(fit_split.image), len(validation_split.image), len(held_out_split.image)
)
print(
    f"split sizes: fit={split_sizes[0]}, validation={split_sizes[1]}, "
    f"held-out={split_sizes[2]}"
)
print(f"parameters: {parameter_count:,}; model seeds: 6050-6054")
report("validation", "validation")
report("held-out endpoint", "held_out")
report_paired_contrast("I→T R@1")
report_paired_contrast("T→I R@1")
print(
    f"held-out random-ranking expectations: "
    f"R@1={1 / 300:.4f}; R@5={5 / 300:.4f}"
)

```

```

split sizes: fit=720, validation=180, held-out=300
parameters: 2,392; model seeds: 6050-6054
validation

```

```

paired   : I→T R@1 0.9911 (SD 0.0084); T→I R@1 0.9900 (SD 0.0061); I→T R@5 1.0000
           ↪ (SD 0.0000); T→I R@5 1.0000 (SD 0.0000)
shuffled: I→T R@1 0.0044 (SD 0.0046); T→I R@1 0.0078 (SD 0.0063); I→T R@5 0.0233
           ↪ (SD 0.0133); T→I R@5 0.0222 (SD 0.0152)
held-out endpoint
paired   : I→T R@1 0.9747 (SD 0.0038); T→I R@1 0.9660 (SD 0.0092); I→T R@5 1.0000
           ↪ (SD 0.0000); T→I R@5 1.0000 (SD 0.0000)
shuffled: I→T R@1 0.0027 (SD 0.0028); T→I R@1 0.0027 (SD 0.0043); I→T R@5 0.0113
           ↪ (SD 0.0084); T→I R@5 0.0100 (SD 0.0097)
paired-shuffled held-out I→T R@1: 0.9720 (paired SD 0.0038)
paired-shuffled held-out T→I R@1: 0.9633 (paired SD 0.0105)
held-out random-ranking expectations: R@1=0.0033; R@5=0.0167

```

Across the five seeds, the paired condition reaches validation Recall@1 of 0.9911 image-to-text and 0.9900 text-to-image. The shuffled control reaches only 0.0044 and 0.0078. That validation separation is the predeclared mechanism check—the towers can align unseen views only when the fit pairs carry the true correspondence.

On the single held-out endpoint, paired image-to-text Recall@1 is **0.9747** (sample SD 0.0038 across model seeds), and paired text-to-image Recall@1 is **0.9660** (SD 0.0092). Both Recall@5 means are 1.0000. The shuffled condition stays at 0.0027 (SD 0.0028) and 0.0027 (SD 0.0043) for Recall@1, close to the 0.0033 random-ranking expectation. Its Recall@5 means, 0.0113 and 0.0100, remain in the same small-sample neighborhood as the 0.0167 random expectation.

Because the two conditions share seeds, the primary top-1 comparison is paired within seed rather than reconstructed from those marginal summaries. The paired-minus-shuffled held-out contrast is **0.9720** (paired SD 0.0038) image-to-text and **0.9633** (paired SD 0.0105) text-to-image. The endpoint made none of the model, hyperparameter, checkpoint, stopping, or analysis-design decisions above; its values are reported and interpreted only after that lock. It is not being reserved for unrelated future work.

These standard deviations describe optimization variation on one fixed synthetic dataset and one fixed derangement—they are not uncertainty intervals for a population of image-text tasks. The study establishes one finite mechanism: correct pair structure is sufficient for these small towers to learn cross-view retrieval, while the matched derangement removes that generalizable signal. It does not estimate what CLIP, ALIGN, or any natural-data model will achieve.

PLAN

1. Define the reusable helpers: `held_values` and `score_window`.
-

CODE

```

# [1]
def held_values(condition: str, field: str) -> list[float]:
    return [
        float(record["held_out"][field])
        for record in records
        if record["condition"] == condition
    ]

```

]

```

@torch.no_grad()
def score_window(model: TwoTower, split: PairedSplit, size: int = 40) -> Tensor:
    model.eval()
    image_embedding, text_embedding = model(split.image[:size], split.text[:size])
    return image_embedding @ text_embedding.T

```

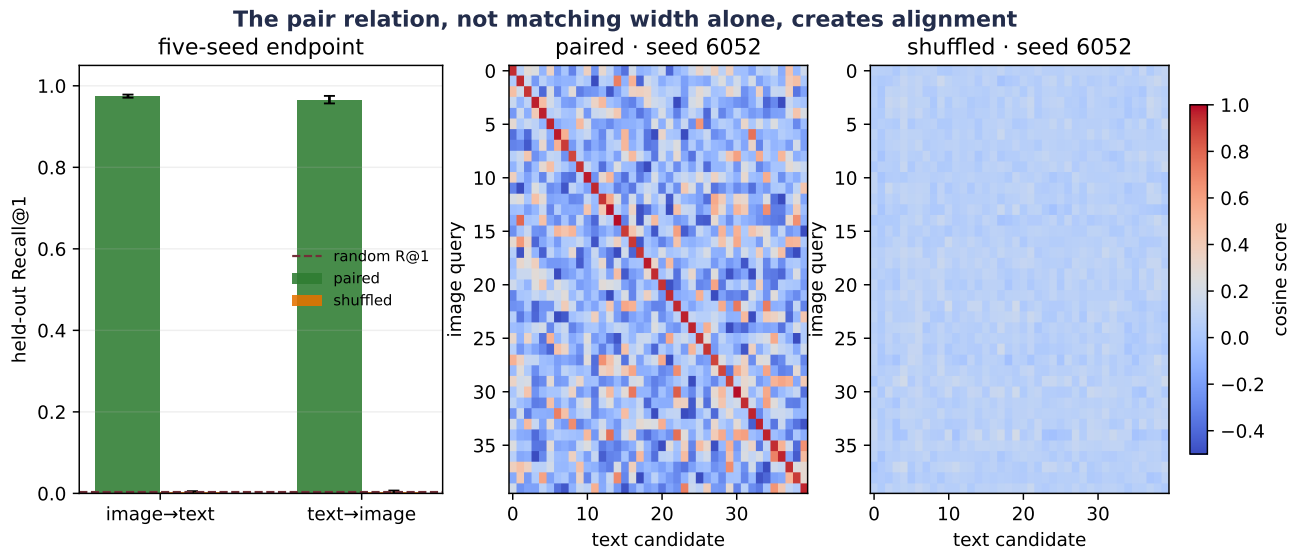


Figure 20.2.: The held-out audit separates aggregate retrieval from one score-matrix view. Bars report the mean and sample SD across five model seeds on all 300 held-out pairs; the dashed line is random Recall@1. The matrices show the first 40 held-out pairs for seed 6052. Correct training concentrates high scores near the declared-pair diagonal, while shuffled fit pairs do not.

The control is stronger than showing a falling training loss—with shuffled pairs, the model can spend capacity fitting accidental associations in the fit table. The correct validation relation refuses to reward that memorization. This is exactly why the experimentation interlude insisted: **tune the contender; ablate the claim**. Here the claim is that paired correspondence teaches alignment, so pairing is the variable the ablation breaks.

There is also a picture of what training *did* to the geometry. Project the two towers' embeddings of the same held-out rows into two dimensions, before and after training:

PLAN

1. Define the reusable `two_dim_view` helper.
 2. Prepare the inputs and fixed settings for the example.
-

CODE

```
# [1]
def two_dim_view(model: TwoTower, count: int = 20) -> tuple[Tensor, Tensor]:
    model.eval()
    with torch.no_grad():
        image_embedding, text_embedding = model(
            held_out_split.image[:count], held_out_split.text[:count]
        )
    both = torch.cat([image_embedding, text_embedding])
    centered = both - both.mean(0)
    _, _, right = torch.linalg.svd(centered, full_matrices=False)
    projected = centered @ right[:2].T
    return projected[:count], projected[count:]

# [2]
torch.manual_seed(6050)
untrained = TwoTower(held_out_split.image.shape[1], held_out_split.text.shape[1])
views = [(untrained, "before training:\n same width, unrelated coordinates"),
         (models[(6050, "paired")], "after paired training:\n mates land together")]
```

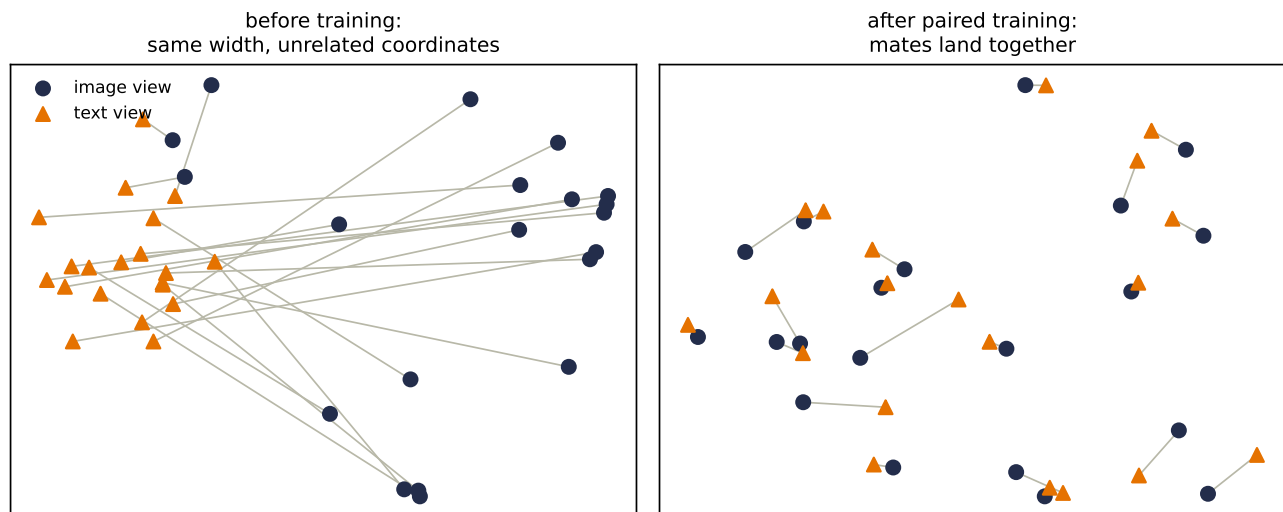


Figure 20.3.: What the contrastive loss does to the shared space. Twenty held-out events, embedded by freshly initialized towers (left) and by the trained paired model (right); each gray segment connects an image embedding to its declared text mate, and each panel is a 2-D principal-component view of the 16-dimensional embeddings. Before training the two views occupy unrelated coordinates and the mate segments sprawl; after training, mates land next to each other — which is precisely why the diagonal of the score matrix lights up. A projection understates 16-D distances, so read the segments qualitatively.

20.3. Retrieval is not generation

What has the two-tower model learned to do? Given an image query and a finite text candidate set, it ranks the candidates. Given a text query and a finite image set, it ranks those. This is **cross-modal retrieval**:

$$\hat{j}(x^{(I)}) = \arg \max_{j \in \{1, \dots, M\}} u(x^{(I)})^\top v(x_j^{(T)}). \quad (20.7)$$

The answer must already be present among the M candidates. The model has no decoder over text tokens and no sampling distribution over images. Chapter 19 made the generative contract explicit—a random source, a conditional or marginal distribution, and a sampling rule. A similarity function supplies none of those by itself.

This distinction prevents a common category error—treating every shared embedding as a generator:

Task	Input	Output rule	What must already exist?
image-to-text retrieval	image query	rank text embeddings	text candidate pool
text-to-image retrieval	text query	rank image embeddings	image candidate pool
image classification by text	image query	rank class descriptions	chosen descriptions
image captioning	image	sample or decode text tokens	conditional language decoder
text-to-image generation	text	sample an image-valued process	conditional image generator

A larger vision–language system may combine a vision encoder, an adapter, and a language decoder. It may train with contrastive, matching, and generative objectives. That composition can support generation—but the new capability comes from the decoder and its objective, not from renaming retrieval.

20.3.1. Zero-shot classification is retrieval over descriptions

Suppose a task supplies C class names. Write one description for each class, encode those descriptions once, and retrieve the description closest to a new image:

$$\hat{c} = \arg \max_{c \in \{1, \dots, C\}} \frac{f_\theta(x)^\top g_\phi(t_c)}{\|f_\theta(x)\|_2 \|g_\phi(t_c)\|_2}. \quad (20.8)$$

This is **zero-shot classification** in the CLIP-style operational sense: no task-specific parameter update is made before this candidate-ranking step. The phrase does not mean that the underlying concept was absent from pretraining. It does not mean that no human chose the class set or wrote the descriptions. It does not mean that the same prediction survives a different prompt, candidate pool, language, or distribution.

Chapter 17 used “zero-shot” for prompting a fixed language model without demonstrations. Here it means classifying with fixed encoders and text prototypes. Both avoid a task-specific weight update—but their inputs, outputs, and evaluation contracts differ.

 **Trap:** a softmax score is conditional on its candidate set

Add a near-synonym, remove a competing description, or change the prompt template and the softmax denominator changes. A high weight within one candidate set is not a candidate-independent probability that the description is true. Report the candidate construction and prompt policy with the metric.

20.4. Shortcuts enter through the pair definition

Contrastive learning rewards whatever makes the diagonal distinguishable. We hope that the useful signal is semantic: the image and text describe the same scene. The model may find an easier route.

- A camera watermark and a caption-source tag may identify the same website. The towers can align sources rather than content.
- Near-duplicate images may cross the split with paraphrased captions. Retrieval then measures leakage, not transfer.
- A broad caption such as “an outdoor scene” may fit many images. The diagonal-only target pretends there is one correct mate.
- A candidate pool may make the task artificially easy. Retrieving a dog caption among unrelated equations is different from retrieving it among fine-grained dog breeds.
- Aggregate recall may hide poor behavior for a language, geographic region, image style, or social group that is rare in the paired data.

These are not side issues—the data contract defines what the objective can learn. Natural-language supervision is attractive because paired text is abundant and rich. The same scale can carry noise, stereotypes, copyrighted material, private information, and uneven coverage. A responsible model card therefore describes how pairs were obtained, filtered, licensed, and audited, not only how many there were.

20.4.1. A minimum evaluation contract

Before accepting a multimodal retrieval result, ask seven questions:

1. **What is the unit of pairing?** One image–caption record, one entity with several views, one time-aligned segment, or something else?
2. **What can cross a split?** Split by the underlying entity or source when row-level splitting would leak near duplicates.
3. **What is a valid match?** If several candidates are correct, use multi-positive labels or a metric that does not mark valid matches wrong.
4. **What is the candidate pool?** State its size, construction, duplicates, and difficulty. Recall@1 over 100 candidates is not Recall@1 over one million.
5. **Which direction is measured?** Image-to-text and text-to-image can behave differently—the symmetric training loss does not guarantee identical retrieval.

6. **Which shortcuts were ablated?** Shuffle pairs, mask a suspected nuisance, remove one modality, or construct a hard candidate set that defeats metadata matching.
7. **Where were decisions made?** Report the seed panel, validation choices, final endpoint policy, subgroup slices, and qualitative failure audit.

Our synthetic study satisfies a narrow version of this contract. The unique latent row is the pairing unit; splits do not overlap; the candidate pool is the complete split; both directions are reported; the shuffled-pair ablation receives a matched budget; and the held-out endpoint makes no model, hyperparameter, checkpoint, stopping, or analysis-design decision. Natural multimodal data makes every one of those lines harder—not optional.

i Check yourself

Close the book for one minute and retrieve the contract before reading the recap.

- Why do equal embedding widths fail to align two modalities by themselves?
- What experiment distinguishes a learned pair relation from accidental memorization?
- Why can a dual encoder retrieve a caption without being able to generate one?

20.5. Okay, so — two towers learn a comparison, not a world model

Multimodal learning begins with a relation between views. Separate encoders map those views into normalized shared coordinates; cosine similarities fill a cross-modal score matrix; and Chapter 2’s scores-to-weights machine turns each row and column into a contrastive classification problem. The symmetric objective teaches retrieval in both directions.

The shuffled-pair study shows what carries the supervision. With the true fit relation, small towers retrieve held-out mates far above random ranking. Break the relation while holding the training budget fixed, and the generalizable signal disappears. Matching embedding width was never enough—the rows had to mean something together.

From there, keep the capability boundary clear. Retrieval selects an existing candidate. Zero-shot classification retrieves among human-supplied descriptions. Generation needs a decoder and sampling contract. In every case, the pair definition, candidate pool, split unit, and failure slices belong beside the metric. The model can only learn the alignment that the data actually declares.

Sources and further reading

- van den Oord, Li, and Vinyals, “[Representation Learning with Contrastive Predictive Coding](#)” (2018), introduced the InfoNCE name and probabilistic contrastive formulation in predictive representation learning.
- Radford et al., “[Learning Transferable Visual Models From Natural Language Supervision](#)” (2021), introduced CLIP’s paired image–text dual-encoder training and zero-shot transfer program; the reported study used 400 million internet image–text pairs. The paper directly optimized a log-parameterized score scale, initialized it at the equivalent of $\tau = 0.07$, and reports capping the scale at 100. The [released reference model](#) initializes and exponentiates that parameter but does not implement the cap.

- Jia et al., “Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision” (2021), developed ALIGN’s dual-encoder contrastive approach with a deliberately noisy large paired corpus.
- PyTorch’s documentation for [normalization](#) and [cross-entropy](#) specifies the two framework operations used in the executable study (`F.normalize` and `F.cross_entropy`).

Exercises

1. **(Pencil.)** Let a batch have score matrix $\mathbf{S} \in \mathbb{R}^{4 \times 4}$. Write the image-to-text loss for row 2 and the text-to-image loss for column 2. Then explain why transposing the logits changes the competition even though both losses send gradients into both encoders.
2. **(Code.)** Re-run the synthetic study with only $\mathcal{L}_{I \rightarrow T}$. Report both retrieval directions and use an open-ended conclusion: does one-sided training change the two directions equally in this finite regime? Keep the initialization and minibatch schedule paired with the symmetric run.
3. **(Code.)** Duplicate 20% of the validation texts so that several image rows have the same valid description. Explain why diagonal Recall@1 can now mark a semantically acceptable retrieval wrong. Design either a multi-positive loss or a group-aware retrieval metric, then state exactly what counts as correct.
4. **(Code.)** Add a shared one-coordinate “source tag” to both fit modalities and train the paired model. At validation, randomize the tag while preserving the true latent pairs. Compare ordinary and tag-randomized retrieval, and diagnose whether the model used the intended semantic signal or the shortcut.
5. **(Pencil.)** A fixed image encoder chooses among “a photo of a crane,” “a photo of a bird,” and “a photo of construction equipment.” List at least three ambiguities in the candidate construction. Propose a prompt and evaluation policy that makes the zero-shot claim bounded and reproducible without claiming the resulting softmax is a universal probability of truth.
6. **(Pencil.)** (a) Let $C_{ij} = u_i^\top v_j$, write $S_{ij} = e^{\gamma C_{ij}}$, and derive $\partial \mathcal{L} / \partial \gamma$ for the symmetric loss. Express each directional term as a softmax-weighted mean similarity minus the diagonal similarity. Determine the gradient’s sign when every diagonal similarity exceeds that weighted mean, predict how gradient descent moves γ , and explain why the gradient fades as diagonal probabilities approach one.
 - (b) **(Code.)** Add a learned `logit_scale` to the two-tower study. Initialize it at $\log(1/0.08)$ and use `logit_scale.exp().clamp(max=100)` in the score matrix. Keep initialization, minibatch order, checkpoint policy, and evaluation queries paired with the fixed- τ run. Report Recall@1 in both directions, the final learned temperature, and fixed-temperature baselines for $\tau \in \{0.03, 0.08, 0.2\}$. Does the learned value lie inside the sweep’s performance plateau?
 - (c) **(Audit.)** State why this jointly trained sharpness parameter is not the post-hoc calibration temperature in [Chapter 16](#), Exercise 6. Name which parameters and data split each procedure is permitted to use.

Epilogue: The Question Is Yours

We began with a line. Its features were fixed, its prediction rule was transparent, and its limitations were easy to draw. Then we asked the question that has followed us through every part of this book:

What if we made this learnable?

At first, that question sounded like permission to add flexibility. By now it should sound more demanding. Every successful step in the book paired something learnable with something deliberately fixed — an objective, a structure, a data contract, an evaluation, or a compute budget. Flexibility without those constraints can fit almost anything. A result without an audit can appear to prove almost anything.

The point of the journey was therefore not to collect architectures — it was to learn a way of seeing a problem: find the rigid part, understand why it fails, decide what the data should be allowed to change, and design evidence that can tell whether the change helped.

The ladder we climbed

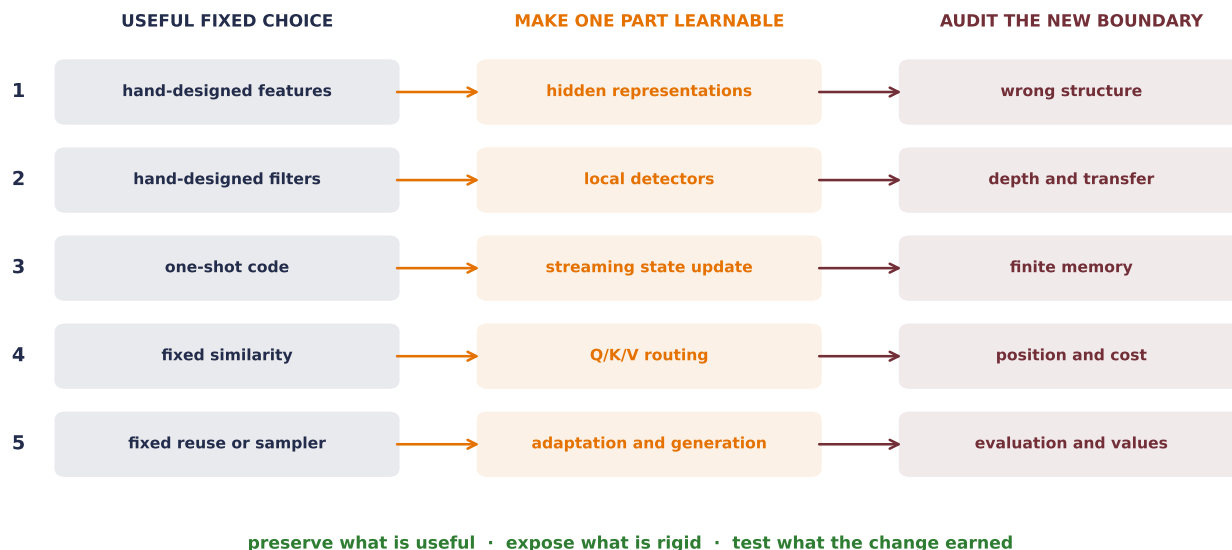


Figure E.1: The book’s recurring move was never flexibility by itself. Each rung begins with a useful fixed choice, makes one part learnable, and then audits the new failure boundary. The rightmost column is what keeps the ladder moving.

The first rung was **representation**. Linear regression could only combine the features we handed it. A multilayer perceptron made intermediate features learnable, while backpropagation assigned blame to millions of knobs and stochastic gradient descent turned that blame into updates. That bought

expressive power. Chapter 6’s shift cliff then taught the necessary counterlesson: the ability to represent a good solution does not make the learner likely to find the right one. Inductive bias tells the search what kinds of solutions deserve preference.

The second rung was **spatial structure**. A classical image filter already knew how to look locally and reuse one detector across positions, but a person chose its numbers. Convolution kept locality and sharing fixed while making the detector weights learnable. Deeper CNNs then made feature hierarchies learnable; residual paths protected the flow of information while depth increased; global pooling and transfer learning changed what had to be relearned for each task. The lesson was not simply that CNNs are good for images — it was that a useful constraint can reduce the search space and improve the learner at the same time.

The third rung was **compression and memory**. PCA supplied a closed-form linear recipe for learning a low-dimensional subspace. **The autoencoder interlude** rewrote that reconstruction as a trainable encoder and decoder, then let the maps bend around nonlinear structure. The same detour exposed a limit: one fixed-width code is a difficult place to store an arbitrarily long sequence. Recurrence made the code an evolving state, updated step by step. The LSTM added learned valves so information and gradients could travel farther. Encoder–decoder models made the handoff between two sequences explicit, and their fixed-state bottleneck prepared the next move.

The fourth rung was **comparison**. Kernel regression predicted by comparing a query with stored examples, but its notion of similarity was fixed. Attention made the query, key, and value maps learnable. Self-attention let every token build context from other tokens, while positional representations restored order that the content comparison alone could not see. The Transformer did not erase inductive bias — it redistributed it into tokenization, position, masking, normalization, architecture, data, and scale. Chapter 20 carried the same comparison rule across modalities: paired observations made image and text encoders comparable in one normalized space. The boundary remained sharp—a shared embedding is a comparison rule, not a generator.

The fifth rung was **reuse**. Pretraining changed the default from “learn every task from a random start” to “begin from a representation shaped by a broader objective and adapt it.” BERT made a bidirectional language representation learnable from masked context. Vision Transformers carried the same attention machinery into patches. PEFT and quantization asked which parts of a large trained system must change, and at what precision, to make adaptation affordable. Alignment asked what behavior the adaptation objective actually rewards. Generative models completed a different contract: not only score an observed object, but specify how a new one is sampled.

These are not five unrelated inventions — they are five passes through the same loop:

expose a limitation → preserve the useful structure → make the rigid part learnable → test the new failure boundary.

The choices no architecture makes for you

A model does not learn what we meant. It learns what the training signal rewards on the data it sees, as far as the optimization procedure can reach. Three choices therefore sit outside every impressive architecture.

The objective chooses what counts as progress. Squared error, cross-entropy, reconstruction error, an ELBO, a preference loss, and a denoising objective reward different behavior. Each is a useful proxy under stated assumptions — not a direct measurement of every property we care about. Making the reward model or judge learnable moves the question one level outward: who trained the judge, on which comparisons, and what shortcuts can satisfy it?

The data chooses the world the learner can encounter. Sampling, labels, preprocessing, augmentation, missing groups, duplicated records, and historical choices all shape the learned function. Chapter 6 named augmentation as data-side inductive bias because its transformations declare which changes should preserve the answer. Pretraining at scale does not remove this dependence; it enlarges the data contract and makes its provenance harder to inspect.

The evaluation chooses what can count as evidence. Clean accuracy did not reveal the shift cliff. Reconstruction did not give an autoencoder a sampling distribution. A discriminator near one half did not certify GAN convergence. Preference accuracy did not settle downstream safety or usefulness. Every metric has a scope, so a serious evaluation names the held-out distribution, relevant slices, perturbations, baselines, uncertainty, and failure conditions before celebrating a number.

Objective, data, and evaluation are not paperwork around the model — they are part of the model’s meaning. When deployment changes any one of them, the original claim may no longer travel.

Learning about learning

There were two learners in this book. Inside one run, weights learned from gradients. Across runs, we learned which designs, recipes, and explanations survived evidence. **The experimentation interlude** made that outer loop explicit.

The habits are simple to state and difficult to practice. Debug before searching. Fix the question before choosing the comparison. Tune a contender enough that a broken recipe is not mistaken for a broken idea. Ablate the claim with a controlled change. Use paired seeds when pairing removes irrelevant variation, and show the individual runs when a mean would hide instability. Keep validation decisions away from the final test audit. Record the budget and the failures, not only the winning configuration.

This discipline changes how we read a plot. A curve is no longer decoration — it is an answer to a declared question under a declared regime. A negative result is no longer an embarrassment; it is a boundary, provided the experiment was capable of detecting the effect. A successful toy problem is no longer a miniature benchmark victory — it is a measuring instrument for one mechanism. “It worked” becomes the beginning of the explanation, not the end.

From fitting the past to evaluating futures

Chapters 12, 14, and 17 answered with evidence already observed: memory fits the past. Could a layer instead evaluate possible futures and return the first action of a plan? Wang, Yang, Vidal and colleagues call that proposal **test-time control**. The operational distinction is backward-looking fitting versus forward-looking evaluation. Their “System 1/System 2” language is a metaphor—a useful cartoon in some settings, not a psychological law or a demonstrated partition inside a model.

The following map extends the paper’s adaptation taxonomy to this book’s landmarks.

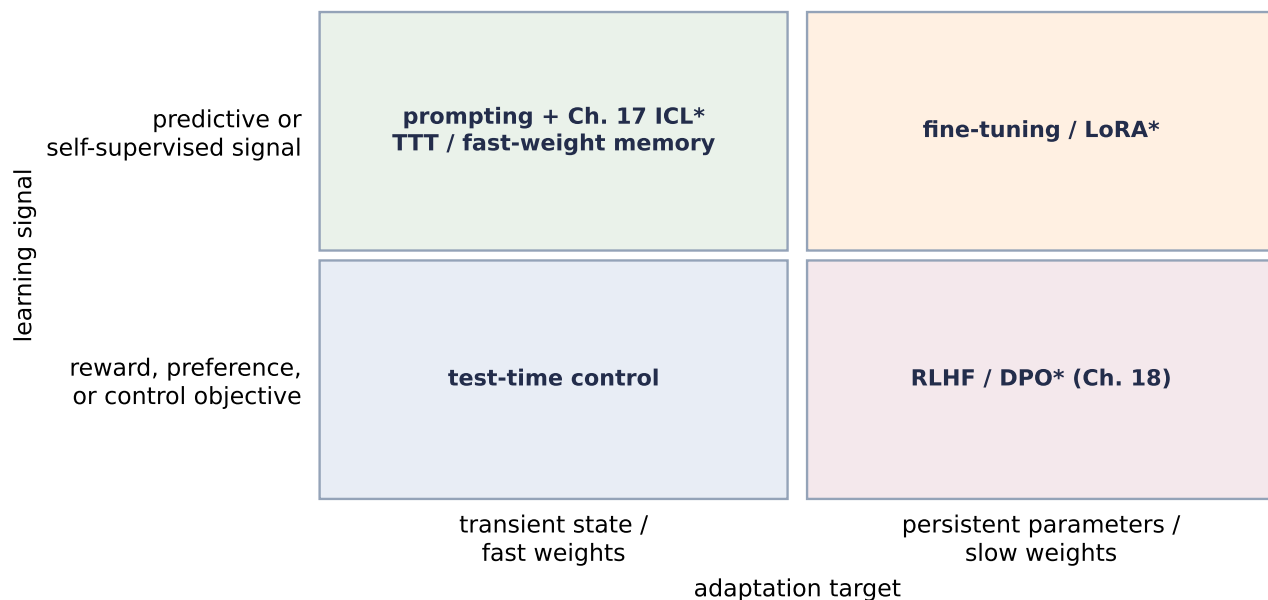


Figure E.2: Adapted and extended from Table 3 in Wang, Yang, Vidal et al. (2026). The axes broaden the paper’s fast/slow weight and self-supervised/reward taxonomy. Asterisks mark analogies: prompting and ICL need not optimize fast weights; LoRA is an update parameterization; DPO uses preferences without requiring policy-gradient RL. This is an organizing map, not an equivalence claim.

The asterisks matter. Prompting changes transient computation without necessarily running an optimizer; test-time training performs an inner update. LoRA says *which* persistent weights may move, not which signal moves them. Chapter 18 trained a judge outside the policy and then changed persistent parameters. Test-time control makes a narrower lower-left bet: place a tractable cost inside the layer and plan against it before each token.

i Deeper dive: plan before predict

The smallest honest example is scalar finite-horizon linear–quadratic control:

$$x_{s+1} = ax_s + bu_s, \quad \mathcal{J} = \sum_{s=0}^{T-1} (q_c x_s^2 + r_c u_s^2) + q_c x_T^2,$$

For $q_c, r_c > 0$, minimize the terms $r_c u_s^2 + P(ax_s + bu_s)^2$ with respect to u_s . Differentiation makes the best action linear in x_s ; substitution gives the scalar Riccati step

$$P \leftarrow q_c + a^2 P - \frac{(abP)^2}{r_c + b^2 P}, \quad K_1 = -\frac{bPa}{r_c + b^2 P}, \quad u_0 = K_1 x_0.$$

Initialize $P = q_c$, update it $T - 1$ times, then form K_1 . Here P is the scalar coefficient of the quadratic value-to-go; a longer horizon performs more backward value updates before choosing the first action.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify that planning horizon changes the first action.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
a, b, q_cost, r_cost = 0.9, 1.0, 1.0, 0.1
first_gains = []
# [2]
for horizon in range(1, 11):
    P = q_cost
    for _ in range(horizon - 1):
        P = q_cost + a*a*P - (a*b*P)**2 / (r_cost + b*b*P)
    first_gains.append(-(r_cost + b*b*P)**-1 * b*P*a)
# [3]
print(f"K1 at T=1: {first_gains[0]:.7f}")
print(f"K1 at T=10: {first_gains[-1]:.7f}")
```

```
K1 at T=1: -0.8181818
K1 at T=10: -0.8233456
```

The horizon changes the *first* output before another token is generated. Increasing T therefore spends more internal computation per token. The full proposal predicts local dynamics and costs and differentiates through a control solver; OptNet and differentiable MPC established related optimization-as-layer machinery. We stop at the scalar recursion: it makes “plan before predict” inspectable without pretending that a toy LQR demonstrates language reasoning.

Wang et al. (2026) make an architectural bet, not a settled claim. A rematch must count internal horizon work and give baselines the same compute. Related differentiable-solver foundations include OptNet (2017) and differentiable MPC (2018). The book’s question lands one rung higher: *what if the planner were learnable?*

Roads this book did not take

The learnability question continues well beyond our route. Graph and geometric models ask how to learn while respecting relational structure and symmetry. Detection and segmentation ask for structured spatial outputs rather than one image-level verdict. State-space sequence models ask which long-memory dynamics can replace or complement attention. Retrieval and tool-using systems make the choice of external evidence or action part of the computation. Reinforcement learning and control extend credit assignment across decisions whose consequences arrive later. Probabilistic and Bayesian methods keep uncertainty visible; causal methods ask which relationships survive intervention rather than mere observation. Sparse mixture-of-experts routing — reviving a design Hinton and colleagues proposed in the early 1990s — makes *which parameters fire* an input-dependent choice, and has become a default at frontier scale. Privacy, formal or certified robustness, mechanistic interpretability, distributed training, and hardware–software co-design each add constraints that accuracy alone cannot express.

These roads require new mathematics and experiments, but not a new intellectual habit. In each case, ask what is fixed, what may adapt, which structure must be protected, and what observation could falsify the claim. A fashionable model name cannot answer those questions for us.

Nor does “learnable” mean “desirable.” A system can become extraordinarily good at an objective that was incomplete, trained on data collected without adequate consent, or deployed where its errors fall unevenly. The more of a pipeline we allow data to tune, the more carefully people must choose its boundaries — and remain responsible for its consequences.

Sources and further reading

- Wang et al., “[Beyond Test-Time Memory: State-Space Optimal Control for LLM Reasoning](#)” (ICML 2026), introduces the Test-Time Control layer and the finite-horizon optimal-control interpretation used in the epilogue.
- Jacobs, Jordan, Nowlan, and Hinton, “[Adaptive Mixtures of Local Experts](#)” (*Neural Computation* 3(1), 1991), supplies the historical learned-gating design behind the mixture-of-experts callback.

One question, now with better follow-ups

When you meet the next rigid heuristic, brittle pipeline, or hand-designed component, begin where this book began: *what if we made this learnable?* Then do not stop there. Ask:

1. What failure makes learning necessary?
2. Which part should adapt, and which constraints carry knowledge we must preserve?
3. What objective and data would teach the intended behavior rather than a shortcut?
4. What baseline, ablation, shift, and held-out evidence would make the claim credible?
5. Who bears the cost when the assumptions fail?

The next important idea may begin as a filter someone chose by hand, a similarity rule that cannot adapt, a memory that is too small, or an evaluation that rewards the wrong thing. You now know how to recognize that moment. Preserve what is true, expose what is rigid, and make only the right part learnable.

The question is yours now.

A. Linear Algebra and the SVD

You have used linear algebra since the first page of this book. A dot product scored a pattern in [Chapter 1](#). Matrix products routed queries to keys in [Chapter 13](#). Low-rank factors limited an update in [Chapter 17](#), and PCA supplied the flat reconstruction baseline in [the autoencoder interlude](#). Why return to the subject now?

Because the notation is compact enough to hide mistakes—a product can be legal while representing the wrong convention; an inverse can exist while being the wrong computation; and an SVD can be exact while its singular vectors are not uniquely identifiable. This appendix gathers the geometry and the PyTorch habits that let you audit those cases. It is a working reference—not a compressed linear-algebra course.

A.1. Read the map from its dimensions

Let us start with one concrete map. A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ accepts a column vector $\mathbf{x} \in \mathbb{R}^n$ and returns $\mathbf{y} = \mathbf{Ax} \in \mathbb{R}^m$. The input has n coordinates, the output has m , and column j of $\mathbf{A}$ records where the j th standard basis vector goes. Once those columns are known, every input follows—by linear combination:

$$\mathbf{Ax} = x_1 \mathbf{A}_{:,1} + \cdots + x_n \mathbf{A}_{:,n}.$$

That is the geometric reading of matrix multiplication. The computational reading is the same operation viewed by rows: output coordinate i is the dot product $\mathbf{A}_{i,:} \mathbf{x}$. Switching between the two views is useful—the columns show what the map can produce, while the rows show how each output is measured.

A map T is **linear** when, for every pair of vectors and scalars,

$$T(a\mathbf{x} + b\mathbf{z}) = aT(\mathbf{x}) + bT(\mathbf{z}).$$

This definition includes $T(\mathbf{0}) = \mathbf{0}$ —a fact worth checking before calling an operation linear. Adding a nonzero bias breaks that condition, so $\mathbf{x} \mapsto \mathbf{Ax} + \mathbf{b}$ is **affine**. PyTorch’s `nn.Linear` uses that affine form by default despite its name—a useful vocabulary distinction whenever a theorem assumes a genuinely linear map.

A.1.1. One vector on paper, a row batch in PyTorch

The mathematical convention above stores one sample as a column. The book's PyTorch code stores B samples as rows of $\mathbf{X} \in \mathbb{R}^{B \times n}$. The same map is therefore

$$\mathbf{Y} = \mathbf{X}\mathbf{A}^\top \in \mathbb{R}^{B \times m}. \quad (\text{A.1})$$

PyTorch stores an `nn.Linear(n, m)` weight with shape (m, n) , and its matrix part uses exactly this row-batch convention. With the default bias, the full affine operation is $\mathbf{Y} = \mathbf{X}\mathbf{A}^\top + \mathbf{1}\mathbf{b}^\top$. What do we expect before running the code? One contract is already fixed—the batch shaped $(3, 2)$ multiplied by $\mathbf{A}^\top \in \mathbb{R}^{2 \times 2}$ must remain $(3, 2)$. Let us check the values as well as the shapes.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Reconcile column-vector algebra with PyTorch row batches.
 3. Report or visualize the measured result.
-

CODE

```
import math

import matplotlib.pyplot as plt
import torch

# [1]
torch.set_default_dtype(torch.float64)
torch.manual_seed(6050)

transform = torch.tensor([[2.0, 1.0], [-1.0, 3.0]])      # (out, in)
vector = torch.tensor([4.0, 2.0])                      # (in,)
batch = torch.tensor([[1.0, 0.0], [2.0, 1.0], [0.0, -1.0]]) # (B, in)

row_outputs = batch @ transform.T                      # (B, out)
column_outputs = (transform @ batch.T).T

rotation = torch.tensor([[0.0, -1.0], [1.0, 0.0]])
scaling = torch.tensor([[2.0, 0.0], [0.0, 3.0]])
order_probe = torch.tensor([1.0, 2.0])
rotate_then_scale = (scaling @ rotation) @ order_probe
scale_then_rotate = (rotation @ scaling) @ order_probe

# [2]
assert torch.equal(transform @ vector, torch.tensor([10.0, 2.0]))
assert torch.equal(row_outputs, column_outputs)
assert not torch.equal(rotate_then_scale, scale_then_rotate)
```

```
# [3]
print("single output:", (transform @ vector).tolist())
print("row batch shapes:", tuple(batch.shape), "->", tuple(row_outputs.shape))
print("row batch outputs:", row_outputs.tolist())
print("rotate then scale:", rotate_then_scale.tolist())
print("scale then rotate:", scale_then_rotate.tolist())
```

```
single output: [10.0, 2.0]
row batch shapes: (3, 2) -> (3, 2)
row batch outputs: [[2.0, -1.0], [5.0, 1.0], [-1.0, -3.0]]
rotate then scale: [-4.0, 3.0]
scale then rotate: [-6.0, 2.0]
```

The two batch calculations agree exactly. The composition check exposes a second habit: matrix order reads from right to left. If $\mathbf{R}$ rotates and $\mathbf{S}$ scales, then $\mathbf{SRx}$ rotates first. Reversing the factors changes the answer from $(-4, 3)$ to $(-6, 2)$ —both products are dimensionally legal, so a shape check alone cannot catch the semantic mistake.

A three-line shape audit

Before every product, write the operand shapes; identify which two dimensions contract; then write the surviving shape. For high-rank tensors, $\textcircled{\cdot}$ treats the last two axes as matrix axes and the leading axes as batch axes. Broadcasting and physical layout are gathered in Appendix B.

A.2. Span, rank, and orthogonality

The **span** of a set of vectors is every linear combination they can produce. A **basis** spans a space without redundant directions. These words make a matrix's geometry precise:

- The **column space** $\text{col}(\mathbf{A}) \subseteq \mathbb{R}^m$ is every reachable output $\mathbf{Ax}$.
- The **null space** $\text{null}(\mathbf{A}) \subseteq \mathbb{R}^n$ is every input direction that the map sends to zero.
- The **rank** is the dimension of the column space—the number of independent output directions. It is at most $\min(m, n)$, and it counts independent input directions that survive the map.

The dot product adds geometry. For $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$,

$$\mathbf{x}^\top \mathbf{z} = \|\mathbf{x}\|_2 \|\mathbf{z}\|_2 \cos \theta.$$

It is zero when nonzero vectors are perpendicular. If $\mathbf{Q} \in \mathbb{R}^{m \times k}$ has orthonormal columns, so that $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}_k$, then

$$\mathbf{P} = \mathbf{Q}\mathbf{Q}^\top, \quad \hat{\mathbf{y}} = \mathbf{P}\mathbf{y} \tag{A.2}$$

projects $\mathbf{y}$ onto $\text{col}(\mathbf{Q})$. The projector is symmetric and idempotent: $\mathbf{P}^\top = \mathbf{P}$ and $\mathbf{P}^2 = \mathbf{P}$. The residual $\mathbf{r} = \mathbf{y} - \hat{\mathbf{y}}$ is orthogonal to every column of $\mathbf{Q}$.

This is the picture behind the normal equations in [Chapter 1](#)—the fit uses every reachable direction and leaves an orthogonal residual. It also explains why [the autoencoder interlude](#) compares PCA and a tied linear autoencoder through their projectors, not through individual basis vectors. Rotating or reflecting an orthonormal basis changes its coordinates while leaving $\mathbf{Q}\mathbf{Q}^\top$ unchanged—the subspace is the stable object.

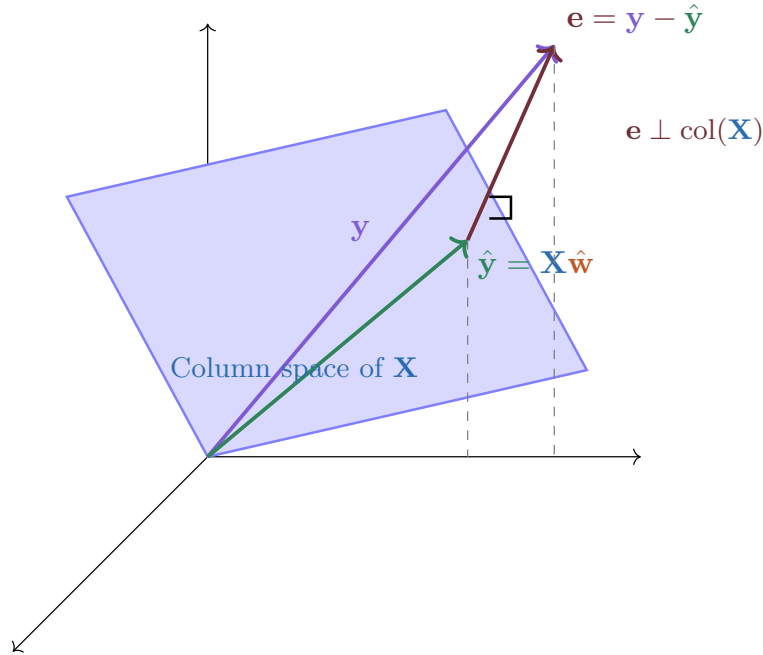


Figure A.1.: Projection separates what the columns can express from what they cannot. The fitted vector $\hat{\mathbf{y}}$ lies in the column space; the residual is orthogonal to every column direction. The diagram labels that residual $\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}}$; the appendix uses $\mathbf{r}$ for the same quantity.

A.3. Solve the problem you actually have

Suppose $\mathbf{A} \in \mathbb{R}^{n \times n}$ is nonsingular and $\mathbf{b} \in \mathbb{R}^n$. The equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ has one solution. The notation $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ describes it, but it does not prescribe a good algorithm. In PyTorch, use `torch.linalg.solve(A, b)`—it solves the system without first materializing an inverse.

Most data problems are rectangular—the target may not lie in the column space. For $\mathbf{X} \in \mathbb{R}^{N \times d}$ and $\mathbf{y} \in \mathbb{R}^N$, least squares asks for

$$\hat{\mathbf{w}} = \arg \min_{\mathbf{w} \in \mathbb{R}^d} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2. \quad (\text{A.3})$$

At an optimum, the residual—a record of what the columns cannot explain— $\mathbf{r} = \mathbf{y} - \mathbf{X}\hat{\mathbf{w}}$ satisfies $\mathbf{X}^\top \mathbf{r} = \mathbf{0}$. If $\mathbf{X}$ has full column rank, expanding that condition gives the familiar normal-equation expression

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

Again, this is an identity—not the preferred recipe. `torch.linalg.lstsq(X, y)` works on the rectangular system directly. It also reports rank-related information for supported algorithms, which matters when columns are redundant or nearly so.

Why avoid forming $\mathbf{X}^\top \mathbf{X}$? Let the nonzero singular values of a full-column-rank matrix be $\sigma_{\max}$ and $\sigma_{\min}$. Its 2-norm condition number is

$$\kappa_2(\mathbf{X}) = \frac{\sigma_{\max}}{\sigma_{\min}}, \quad \kappa_2(\mathbf{X}^\top \mathbf{X}) = \kappa_2(\mathbf{X})^2. \quad (\text{A.4})$$

Directions that were difficult to distinguish become much more difficult after the square—the next cell uses a matrix whose columns differ by only 0.001.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Solve square and rectangular systems, then expose conditioning.
3. Check the claimed identities, shapes, or invariants.
4. Report or visualize the measured result.

CODE

```
# [1]
square = torch.tensor([[3.0, 1.0], [1.0, 2.0]])
right_hand_side = torch.tensor([9.0, 8.0])
# [2]
solution = torch.linalg.solve(square, right_hand_side)

design = torch.tensor([[1.0, 0.0], [1.0, 1.0],
                      [1.0, 2.0], [1.0, 3.0]])          # (N, d)
target = torch.tensor([1.0, 2.0, 2.0, 4.0])           # (N,)
least_squares = torch.linalg.lstsq(design, target)
weights = least_squares.solution
residual = target - design @ weights
normal_residual = torch.linalg.vector_norm(design.T @ residual)

near_collinear = torch.tensor([[1.0, 1.0], [1.0, 1.001]])
condition = torch.linalg.cond(near_collinear)
normal_condition = torch.linalg.cond(near_collinear.T @ near_collinear)

# [3]
assert torch.allclose(solution, torch.tensor([2.0, 3.0]))
assert normal_residual < 1e-12
assert torch.allclose(normal_condition, condition.square(), rtol=1e-7)

# [4]
print(f"solve solution: ({solution[0]:.6f}, {solution[1]:.6f})")
print(f"least-squares solution: ({weights[0]:.6f}, {weights[1]:.6f})")
```

```
print(f"normal-equation residual: {normal_residual:.3e}")
print(f"condition(A): {condition:.6f}")
print(f"condition(A.T @ A): {normal_condition:.6f}")
print(f"squaring ratio: {normal_condition / condition.square():.12f}")
```

```
solve solution: (2.000000, 3.000000)
least-squares solution: (0.900000, 0.900000)
normal-equation residual: 2.220e-15
condition(A): 4002.000750
condition(A.T @ A): 16016009.992981
squaring ratio: 0.999999999312
```

The least-squares solution is $(0.9, 0.9)$, and the norm of $\mathbf{X}^\top \mathbf{r}$ is about 2.2×10^{-15} in float64. For the nearly collinear matrix, the condition number grows from about 4.002×10^3 to 1.602×10^7 after forming $\mathbf{A}^\top \mathbf{A}$ —the predicted square to displayed precision.

⚠ More digits do not repair missing information

Float64 makes this tiny audit easier to inspect; it does not make an ill-conditioned problem well-conditioned. Check inputs with `torch.isfinite`, inspect `torch.linalg.cond` or the singular values when stability matters, and verify the residual. Near a numerical-rank threshold, reported rank and solutions can change with dtype, tolerance, algorithm, or platform. Appendix C separates those numerical questions from the underlying algebra.

A.4. Eigenvectors are a square-matrix special case

An eigenvector asks whether one nonzero direction stays on its own line under a square map:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}, \quad \mathbf{A} \in \mathbb{R}^{n \times n}.$$

This question requires the input and output spaces to be the same—even then, a real matrix may have complex eigenvalues or too few independent eigenvectors. Symmetric real matrices are the friendly case: their eigenvalues are real and they admit an orthonormal eigenbasis. Use `torch.linalg.eigh`, rather than the more general `eig`, when that symmetry is known.

Deep-learning weight matrices are often rectangular, so we need a decomposition that uses one set of directions in the input space and another in the output space. That is the role of the singular value decomposition—the two spaces no longer need to be the same.

A.5. SVD: two spaces and one ordered scale

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and set $k = \min(m, n)$. Its reduced SVD is

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top, \tag{A.5}$$

with

A. Linear Algebra and the SVD

$$\mathbf{U} \in \mathbb{R}^{m \times k}, \quad \mathbf{\Sigma} = \text{diag}(\sigma_1, \dots, \sigma_k) \in \mathbb{R}^{k \times k}, \quad \mathbf{V}^\top \in \mathbb{R}^{k \times n}.$$

The singular values are real, nonnegative, and ordered—a scale ledger from the most amplified direction to the least: $\sigma_1 \geq \dots \geq \sigma_k \geq 0$. The columns satisfy

$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i, \quad \mathbf{A}^\top \mathbf{u}_i = \sigma_i \mathbf{v}_i \quad \text{when } \sigma_i > 0. \quad (\text{A.6})$$

Geometrically, $\mathbf{V}^\top$ selects orthogonal input coordinates, $\mathbf{\Sigma}$ scales or removes them, and $\mathbf{U}$ places the result in the output space. Strang’s rotate–stretch–rotate picture is useful in the square, two-dimensional case—but orthogonal factors may include reflections. In a reduced rectangular SVD, $\mathbf{V}^\top$ maps n coordinates to k and $\mathbf{U}$ maps those k coordinates into an m -dimensional output; the dimension change belongs to the whole factorization, not to the displayed $k \times k$ diagonal matrix alone.

The eigenvalue bridge follows—without using $\mathbf{A}^\top \mathbf{A}$ as a numerical SVD algorithm:

$$\mathbf{A}^\top \mathbf{A} \mathbf{v}_i = \sigma_i^2 \mathbf{v}_i, \quad \mathbf{A} \mathbf{A}^\top \mathbf{u}_i = \sigma_i^2 \mathbf{u}_i. \quad (\text{A.7})$$

`torch.linalg.svd(A, full_matrices=False)` returns $\mathbf{U}$, $\mathbf{S}$, $\mathbf{Vh}$; $\mathbf{S}$ is a one-dimensional tensor, not a diagonal matrix. Scaling the columns of $\mathbf{U}$ reconstructs the map without building that diagonal matrix: `(U * S.unsqueeze(-2)) @ Vh`.

A.5.1. Rank-one layers and truncation

Expanding Equation A.5 gives

$$\mathbf{A} = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^\top. \quad (\text{A.8})$$

Each outer product has rank at most one—a complicated map becomes an ordered sum of simple layers. Keeping the first r terms gives

$$\mathbf{A}_r = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^\top. \quad (\text{A.9})$$

For $0 \leq r < k$, the Eckart–Young–Mirsky result makes “best” precise—among matrices of rank at most r , $\mathbf{A}_r$ minimizes both the operator-norm and Frobenius-norm errors, with

$$\|\mathbf{A} - \mathbf{A}_r\|_2 = \sigma_{r+1}, \quad \|\mathbf{A} - \mathbf{A}_r\|_F^2 = \sum_{i=r+1}^k \sigma_i^2. \quad (\text{A.10})$$

Let us construct a map whose singular values are exactly 3 and 1. The full map sends the unit circle to an ellipse. The rank-one approximation keeps the longer axis and collapses the other—a visible instance of the theorem rather than a claim inferred from a complicated dataset.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Decompose a known map and expose the rank-one error.

CODE

```
# [1]
left_angle, right_angle = math.pi / 6.0, -math.pi / 4.0
left_basis = torch.tensor([
    [math.cos(left_angle), -math.sin(left_angle)],
    [math.sin(left_angle), math.cos(left_angle)],
])
right_basis = torch.tensor([
    [math.cos(right_angle), -math.sin(right_angle)],
    [math.sin(right_angle), math.cos(right_angle)],
])
linear_map = left_basis @ torch.diag(torch.tensor([3.0, 1.0])) @ right_basis.T

u, singular_values, vh = torch.linalg.svd(linear_map, full_matrices=False)
reconstructed = (u * singular_values.unsqueeze(0)) @ vh
rank_one = (u[:, :1] * singular_values[:1]) @ vh[:1]
frobenius_error = torch.linalg.matrix_norm(linear_map - rank_one, ord="fro")
operator_error = torch.linalg.matrix_norm(linear_map - rank_one, ord=2)
eigenvalues = torch.linalg.eigvalsh(linear_map.T @ linear_map)

assert torch.allclose(singular_values, torch.tensor([3.0, 1.0]))
assert torch.allclose(linear_map, reconstructed)
assert torch.allclose(eigenvalues.flip(0), singular_values.square())
assert torch.allclose(frobenius_error, singular_values[1:].square().sum().sqrt())
assert torch.allclose(operator_error, singular_values[1])

# [2]
print("singular values:", [round(value, 6) for value in singular_values.tolist()])
print(f"reconstruction max error: {(linear_map - reconstructed).abs().max():.3e}")
print("eigenvalues of A.T @ A:", [round(value, 6) for value in eigenvalues.tolist()])
print(f"rank-one Frobenius error: {frobenius_error:.6f}")
print(f"rank-one operator error: {operator_error:.6f}")

angles = torch.linspace(0.0, 2.0 * math.pi, 361)
unit_circle = torch.stack((torch.cos(angles), torch.sin(angles))) # (2, points)
full_image = linear_map @ unit_circle
rank_one_image = rank_one @ unit_circle

fig, axes = plt.subplots(1, 3, figsize=(10, 3.4))
axes[0].plot(unit_circle[0], unit_circle[1], color="#232D4B", lw=2)
for index, color in enumerate(["#E57200", "#2E7D32"]):
```

```

    direction = vh[index]
    axes[0].plot([0.0, direction[0]], [0.0, direction[1]], color=color, lw=2)
axes[0].set_title("input directions")

axes[1].plot(full_image[0], full_image[1], color="#232D4B", lw=2)
for index, color in enumerate(["#E57200", "#2E7D32"]):
    scaled_axis = singular_values[index] * u[:, index]
    axes[1].plot([0.0, scaled_axis[0]], [0.0, scaled_axis[1]], color=color, lw=2)
axes[1].set_title("full map: stretch 3 and 1")

axes[2].plot(full_image[0], full_image[1], color="#9AA5B1", lw=1.5, ls="--")
axes[2].plot(rank_one_image[0], rank_one_image[1], color="#E57200", lw=3)
axes[2].set_title("rank one: one axis retained")

for axis in axes:
    axis.axhline(0.0, color="#D6DCE5", lw=0.8)
    axis.axvline(0.0, color="#D6DCE5", lw=0.8)
    axis.set_aspect("equal")
    axis.set_xlim(-3.4, 3.4)
    axis.set_ylim(-3.4, 3.4)
    axis.set_xlabel("coordinate 1")
axes[0].set_ylabel("coordinate 2")
plt.tight_layout()
plt.show()

```

```

singular values: [3.0, 1.0]
reconstruction max error: 8.882e-16
eigenvalues of A.T @ A: [1.0, 9.0]
rank-one Frobenius error: 1.000000
rank-one operator error: 1.000000

```

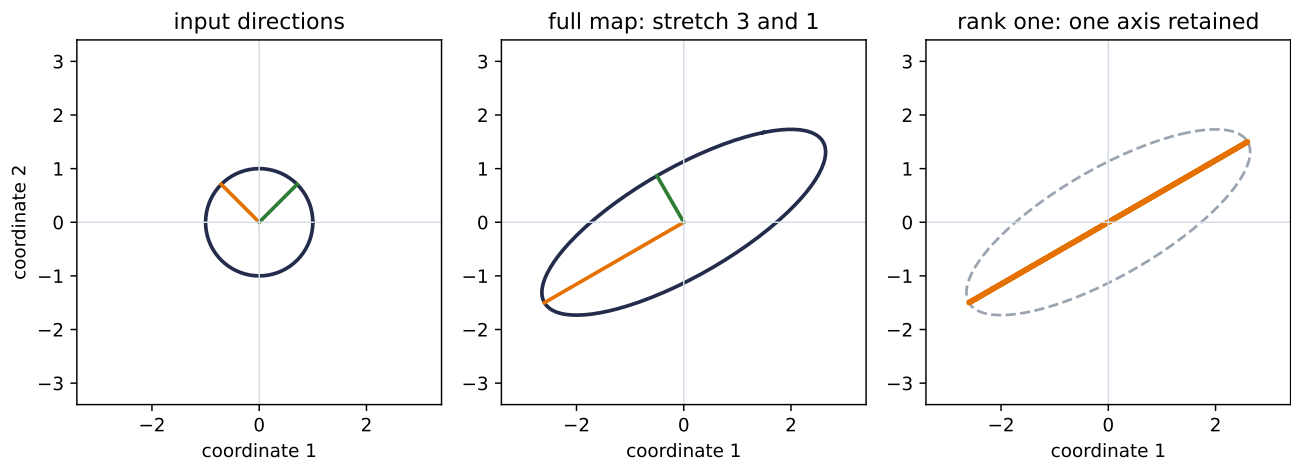


Figure A.2.: A two-dimensional SVD with singular values 3 and 1. The right singular vectors choose orthogonal input directions; the left singular vectors orient the ellipse's scaled axes. Keeping one rank-one term collapses the ellipse to its longer axis. Because there is only one omitted singular value, both the operator error and Frobenius error are exactly 1.

The singular values are unique, but their vectors need not be—each singular pair can flip sign without changing $\mathbf{A}$. When singular values repeat, any orthonormal basis of the repeated subspace is valid. Accordingly, tests should compare the reconstruction or a subspace projector—not demand one particular $\mathbf{U}$ or $\mathbf{V}_h$.

Magnitude is not meaning

A truncated SVD is optimal for the matrix norms in Equation A.10. That does not prove that a large singular direction is semantic signal or that a small one is noise. Feature scaling, the downstream task, and the data-generating process decide whether discarding a direction is useful. A low-rank weight approximation can be close as a matrix yet still change a model's predictions; measure the downstream behavior.

A.6. Batched linear algebra

The same decomposition can be applied to many matrices at once—the last two dimensions remain the matrix. For an input shaped (B, m, n) , `torch.linalg.svd(..., full_matrices=False)` returns shapes (B, m, k) , (B, k) , and (B, k, n) . The singular values need one inserted axis so they scale columns of $\mathbf{U}$ rather than a batch axis.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Reconstruct a batch of rectangular matrices from reduced SVDs.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
rectangular_batch = torch.stack((
    torch.tensor([[3.0, 0.0], [0.0, 1.0], [0.0, 0.0]]),
    torch.tensor([[0.0, 2.0], [0.5, 0.0], [0.0, 0.0]]),
))
# (B, m, n)

batch_u, batch_s, batch_vh = torch.linalg.svd(
    rectangular_batch, full_matrices=False
)
batch_reconstruction = (batch_u * batch_s.unsqueeze(-2)) @ batch_vh

# [2]
assert torch.allclose(rectangular_batch, batch_reconstruction)

# [3]
print("A batch:", tuple(rectangular_batch.shape))
print("U, S, Vh:", tuple(batch_u.shape), tuple(batch_s.shape), tuple(batch_vh.shape))
```

```
print(
    f"batched reconstruction max error: "
    f"{(rectangular_batch - batch_reconstruction).abs().max():.3e}"
)
print("singular values by matrix:", batch_s.tolist())
```

```
A batch: (2, 3, 2)
U, S, Vh: (2, 3, 2) (2, 2) (2, 2, 2)
batched reconstruction max error: 0.000e+00
singular values by matrix: [[3.0, 1.0], [2.0, 0.5]]
```

The leading dimension stays a batch throughout. A Python loop and a batched call are mathematically equivalent here, but floating-point implementations need not be bitwise identical. Use tolerances such as `torch.allclose`, and choose them from the dtype and the scale of the quantity being checked—not from the number of decimals you hope to see.

A.7. PCA is SVD after centering

Let $\mathbf{X} \in \mathbb{R}^{N \times d}$ contain observations in rows, and let $\boldsymbol{\mu} \in \mathbb{R}^d$ be the training mean. PCA begins with

$$\mathbf{X}_c = \mathbf{X} - \mathbf{1}\boldsymbol{\mu}^\top.$$

If $\mathbf{X}_c = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$, then the first r principal directions are the columns of $\mathbf{V}_r$. The scores and reconstruction are

$$\mathbf{Z} = \mathbf{X}_c \mathbf{V}_r, \quad \widehat{\mathbf{X}} = \mathbf{Z}\mathbf{V}_r^\top + \mathbf{1}\boldsymbol{\mu}^\top. \quad (\text{A.11})$$

For the usual sample-covariance convention, component i explains variance $\sigma_i^2/(N-1)$. Centering is not cosmetic—it changes the question. Without it, the leading singular direction of the raw data matrix can mostly describe the offset from the origin.

The next five points all have first coordinate 10 and vary only in the second coordinate. What should PCA find? The answer is vertical variation. Raw SVD instead spends its first direction on the large mean; centering restores the intended question.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Make the PCA centering failure visible.
-

CODE

```

# [1]
observations = torch.tensor([
    [10.0, -2.0], [10.0, -1.0], [10.0, 0.0],
    [10.0, 1.0], [10.0, 2.0],
])
training_mean = observations.mean(dim=0)

_, raw_s, raw_vh = torch.linalg.svd(observations, full_matrices=False)
centered = observations - training_mean
_, centered_s, centered_vh = torch.linalg.svd(centered, full_matrices=False)

scores = centered @ centered_vh[:1].T
pca_reconstruction = scores @ centered_vh[:1] + training_mean
explained_variance = centered_s.square() / (observations.shape[0] - 1)
raw_values = [round(value, 6) for value in raw_s.tolist()]
centered_values = [round(value, 6) for value in centered_s.tolist()]

assert torch.allclose(raw_vh[0, 0].abs(), torch.tensor(1.0))
assert torch.allclose(centered_vh[0, 1].abs(), torch.tensor(1.0))
assert torch.allclose(observations, pca_reconstruction)

# [2]
print("training mean:", training_mean.tolist())
print("uncentered singular values:", raw_values)
print("centered singular values:", centered_values)
print(f"uncentered top-axis |x alignment|: {raw_vh[0, 0].abs():.6f}")
print(f"centered top-axis |y alignment|: {centered_vh[0, 1].abs():.6f}")
print(
    f"centered rank-one max error: "
    f"{(observations - pca_reconstruction).abs().max():.3e}"
)
print("explained variance:", explained_variance.tolist())

fig, axes = plt.subplots(1, 2, figsize=(8.4, 3.6), sharex=True, sharey=True)
for axis in axes:
    axis.scatter(observations[:, 0], observations[:, 1], s=46,
                color="#232D4B", zorder=3)
    axis.scatter(training_mean[0], training_mean[1], marker="x", s=90,
                color="#E57200", lw=2.5, zorder=4)
    axis.axhline(0.0, color="#D6DCE5", lw=0.8)
    axis.axvline(0.0, color="#D6DCE5", lw=0.8)
    axis.set_xlim(-1.0, 12.0)
    axis.set_ylim(-3.0, 3.0)
    axis.set_aspect("equal")
    axis.set_xlabel("feature 1")

axes[0].plot([-1.0, 12.0], [0.0, 0.0], color="#722F37", lw=2.5)
axes[0].set_title("without centering: mean wins")

```

```

axes[0].set_ylabel("feature 2")
axes[1].plot([10.0, 10.0], [-3.0, 3.0], color="#2E7D32", lw=2.5)
axes[1].set_title("after centering: variation wins")
plt.tight_layout()
plt.show()

```

```

training mean: [10.0, 0.0]
uncentered singular values: [22.36068, 3.162278]
centered singular values: [3.162278, 0.0]
uncentered top-axis |x alignment|: 1.000000
centered top-axis |y alignment|: 1.000000
centered rank-one max error: 0.000e+00
explained variance: [2.5000000000000004, 0.0]

```

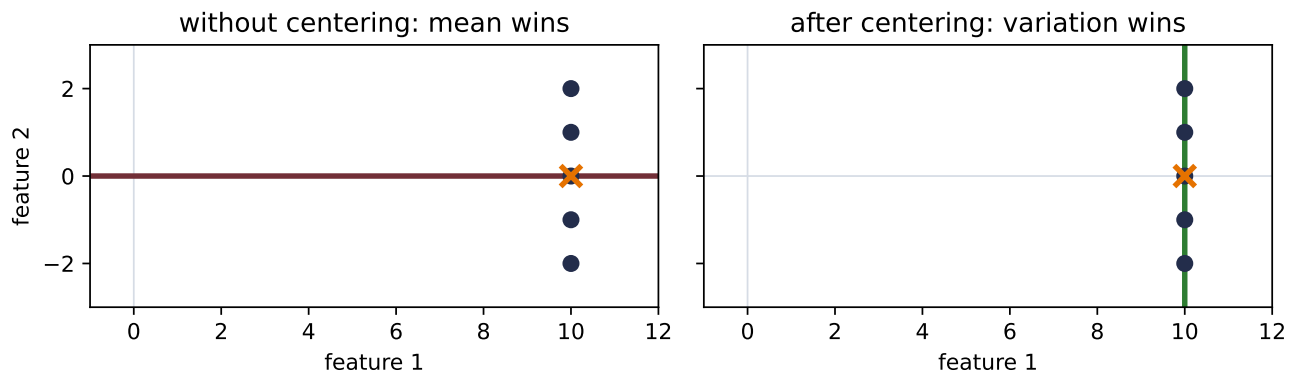


Figure A.3.: PCA asks about variation around the training mean. These five points vary only vertically, but raw SVD first follows their large horizontal offset from the origin: its singular values are 22.360680 and 3.162278. After centering at (10, 0), the vertical singular value is 3.162278, the other is zero, and one component reconstructs every point exactly.

In a train/validation/test workflow, estimate μ and the principal directions on the training split only. Reuse that training mean for every later split; re-centering validation or test data separately changes the map and leaks information. [The autoencoder interlude](#) adds the next link: a tied linear autoencoder can recover the same principal reconstruction subspace under its stated centered, squared-loss, undercomplete, global-optimum conditions. It does not guarantee the same basis or that gradient descent reaches the optimum.

A.8. Practical torch.linalg checklist

When the algebra moves into code, ask these questions in order:

1. **What are the last two dimensions?** For operands with rank at least two, they define the matrix and leading dimensions are batches. Rank-one operands use `matmul`'s vector-specific insertion and removal rules; [Appendix B](#) works through those cases. Write the expected output shape before executing.

2. **Is the task a product, a solve, or a least-squares problem?** Use `@`, `solve`, or `lstsq` directly. Do not compute an inverse or pseudoinverse merely to multiply it by a right-hand side.
3. **Does structure buy a better routine?** Use `eigh` for real symmetric matrices and `svdvals` when only singular values are needed.
4. **Are the values finite and the problem identifiable?** Check finiteness, rank, conditioning, and residuals. A returned tensor is not evidence that the answer is trustworthy.
5. **What is actually invariant?** Compare reconstructions, losses, residuals, or projectors. Singular-vector signs and bases within repeated subspaces are not fixed.
6. **Does autograd need the vectors?** Gradients involving `U` or `Vh` can become unstable when singular values coincide or nearly coincide. If the objective needs only singular values, compute only those; otherwise test the gradient regime rather than assuming the forward decomposition settles it.

Okay, so—the whole appendix can be reduced to one habit: read the geometry and the shapes before trusting the operation. Matrices map one space to another; least squares projects onto what is reachable; SVD names the independent input and output directions; truncation has a precise norm guarantee; PCA adds the indispensable centering step. PyTorch will perform each calculation quickly, but it cannot decide whether you asked the right question.

Sources and further reading

- [Grant Sanderson, *Essence of Linear Algebra*](#): the visual interpretation of vectors, bases, and matrices as transformations. The figures here are independently constructed rather than adapted from the series.
- [Gilbert Strang, *A Vision of Linear Algebra*](#): input/output spaces, singular vectors, and the geometry of SVD in MIT OpenCourseWare.
- [Gilbert Strang, *Linear Algebra 18.06SC*](#): column spaces, projections, least squares, eigenvectors, and SVD.
- [Eckart and Young, *The Approximation of One Matrix by Another of Lower Rank*](#): the original low-rank approximation result; Mirsky later extended the norm statement.
- [Baldi and Hornik, *Neural Networks and Principal Component Analysis: Learning from Examples without Local Minima*](#): the qualified relationship between undercomplete linear autoencoders and principal subspaces used in [the autoencoder interlude](#).
- [PyTorch, *Linear algebra \(torch.linalg\)*](#): current interfaces for products, solves, decompositions, rank, and norms.
- [PyTorch, *Numerical accuracy*](#): conditioning, finite inputs, batched-operation differences, and spectral-decomposition caveats.

Exercises

1. **(Pencil.)** Let $\mathbf{W} \in \mathbb{R}^{5 \times 3}$ be the weight of an `nn.Linear(3, 5)`. Write the shapes of the output for inputs shaped $(3,)$, $(7, 3)$, and $(4, 7, 3)$. Then explain why PyTorch evaluates a row batch with $\mathbf{XW}^\top$ even though the single-vector formula is $\mathbf{W}\mathbf{x}$.
2. **(Pencil.)** (a) From Equation A.3, derive $\mathbf{X}^\top \mathbf{r} = \mathbf{0}$. **(Code.)** (b) Construct increasingly collinear design matrices, compare `torch.linalg.lstsq(X, y)` with a solve on the normal equations, and

report residual and parameter error in float32 and float64. Do not assume which method first shows visible error.

3. **(Pencil.)** Using Equation A.6, prove Equation A.7. For a repeated singular value, show why rotating the corresponding columns of $\mathbf{U}$ and $\mathbf{V}$ together leaves $\mathbf{A}$ unchanged.
4. **(Code.)** Generate a deterministic batch with shape $(6, 8, 3)$ and reconstruct all six matrices from reduced SVDs. Build rank-one and rank-two approximations, verify Equation A.10 for every batch member, and state a tolerance justified by the dtype and matrix scale.
5. **(Code.)** Add a large offset to two-dimensional data and compare raw SVD with centered PCA. Fit the mean and basis on a training split; apply both unchanged to validation data. Compare projectors rather than signed basis vectors, and explain the leakage from recentering validation data.

B. Tensors in Practice

Here is a PyTorch bug that did not begin with an error message. In the Chapter 1 Chapter 1’s coding session, a prediction had shape $(N,)$ while a noise column had shape $(N, 1)$. Adding them did not produce $(N, 1)$. PyTorch compared the axes from the right, expanded both inputs, and legally returned (N, N) . The calculation ran—the meaning was wrong.

That debugging pause contains the whole lesson of this appendix. A tensor is not just a box of numbers—and a shape is not enough unless we know what every axis means. Chapters 1–20 have already used dense batches, images, sequences, attention heads, masks, and parameter tensors. Here we gather the habits that make those objects predictable: state the contract, name the axes, and make every operation account for the axes it changes.

B.1. The six-part tensor contract

Before operating on a tensor `x`, you should be able to answer six questions.

Part of the contract	Question to ask	PyTorch evidence
Values	What does each number represent?	Inspect a small slice; check its range and invariants
Shape and axes	What does each position in the shape mean?	<code>x.shape</code> , <code>x.ndim</code> , plus names in prose or comments
Data type	Are these real values, integer indices, or Boolean decisions?	<code>x.dtype</code>
Device	Where does the storage live, and do interacting tensors agree?	<code>x.device</code>
Layout	How does an index step through storage?	<code>x.stride()</code> , <code>x.is_contiguous()</code>
Gradient role	Is autograd recording how this value was produced?	<code>x.requires_grad</code> , <code>x.grad_fn</code> , <code>x.is_leaf</code>

The values and their axis meanings are the *modeling* contract—dtype, device, and layout are the *representation* contract. Gradient status is the *learning* contract. One PyTorch object carries all three—changing any one can change which operations are valid.

i Dimension, order, and rank are not synonyms

A scalar tensor has shape $()$ and zero axes. A vector-shaped tensor such as $(7,)$ has one axis; a matrix-shaped tensor such as $(5, 7)$ has two. In code, call this `ndim` rather than “rank.” **Matrix rank** means the number of linearly independent directions and is computed by `torch.linalg.matrix_rank`. In geometry and physics, *tensor* has a further coordinate-

transformation meaning. A PyTorch tensor can store the coordinates of such an object, but an arbitrary multidimensional array is not automatically that mathematical tensor.

Autograd deserves the same precision. A tensor does not track history merely because it is a tensor. Operations are recorded when gradient mode is active and at least one relevant input requires gradients; a scalar loss then seeds the reverse pass. Chapter 5 develops that machinery in [Chapter 5](#). Here, `requires_grad` is one field to audit rather than a promise that `backward()` will always be meaningful.

B.2. The book’s shape dictionary

We use an **examples-as-rows batch convention**: the first axis indexes examples, while the final axis usually carries features. Spatial tensors are the important exception because they retain named channel, height, and width axes.

Table B.1.: The book’s recurring tensor shapes and axis meanings.

Object	Shape used in this book	Axis meaning
Dense batch	(B, D)	batch, features
Dense weight	(D_out, D_in)	output features, input features
Image batch	(B, C, H, W)	batch, channels, height, width
Convolution weight	(C_out, C_in, K_h, K_w)	output channels, input channels, kernel height, kernel width
Sequence batch	(B, T, D)	batch, time or tokens, features
Multi-head states	(B, N_h, T, D_h)	batch, heads, positions, width per head
Attention scores	(B, N_h, T, S)	batch, heads, query positions, key positions
Token logits	(B, T, V)	batch, positions, vocabulary classes

The entries in Table [B.1](#) are contracts—not universal laws. An external library or dataset may choose a different order. Convert once at the boundary—assert the result, and then follow one convention internally. Chapter 8 establishes NCHW for images ([Chapter 8](#)); Chapter 10 establishes batch-first sequences ([Chapter 10](#)); Chapters 13–16 extend that sequence convention to source positions, heads, masks, and image patches.

Keep the batch axis explicit even when $B=1$. For a token-level loss, logits (B, T, V) become $(B * T, V)$ and labels (B, T) become $(B * T,)$ only at the loss boundary; `reshape(-1, V)` and `reshape(-1)` must flatten the same batch-time order. Until then, preserving B and T keeps examples and positions recoverable.

B.2.1. The `nn.Linear` storage convention

Let one example be a column vector $\mathbf{x} \in \mathbb{R}^{D_{\text{in}}}$. A dense layer stores $\mathbf{W} \in \mathbb{R}^{D_{\text{out}} \times D_{\text{in}}}$ and $\mathbf{b} \in \mathbb{R}^{D_{\text{out}}}$, so

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}.$$

PyTorch batches examples as rows: $\mathbf{X} \in \mathbb{R}^{B \times D_{\text{in}}}$. The same stored weight therefore appears transposed in the batch expression. Let $\mathbf{1}_B \in \mathbb{R}^B$ be an all-ones vector. Then

$$\mathbf{Y} = \mathbf{X}\mathbf{W}^\top + \mathbf{1}_B\mathbf{b}^\top, \quad \mathbf{Y} \in \mathbb{R}^{B \times D_{\text{out}}}.$$

PyTorch writes the same broadcast compactly as `X @ W.T + b`. The bias matches the trailing output-feature axis—it is reused across the batch. Let us pin that convention against `nn.Linear` rather than trusting memory.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify the `nn.Linear` weight and bias shape contract.
 3. Report or visualize the measured result.
-

CODE

```
import torch
from torch import nn

# [1]
_ = torch.manual_seed(6050)

B, D_IN, D_OUT = 3, 4, 2
X = torch.arange(B * D_IN, dtype=torch.float64).reshape(B, D_IN) / 10
W = torch.tensor(
    [[1.0, -1.0, 0.5, 2.0], [-0.5, 1.5, 1.0, 0.0]],
    dtype=torch.float64,
)
b = torch.tensor([0.25, -0.75], dtype=torch.float64)

manual = X @ W.T + b
layer = nn.Linear(D_IN, D_OUT, dtype=torch.float64)
with torch.no_grad():
    layer.weight.copy_(W)
    layer.bias.copy_(b)
output = layer(X)

# [2]
assert output.shape == (B, D_OUT)
```

```

assert torch.allclose(output, manual)
gap = (output - manual).detach().abs().max().item()
# [3]
print(f"X {tuple(X.shape)}, W {tuple(W.shape)}, b {tuple(b.shape)}")
print(f"output {tuple(output.shape)}, max gap {gap:.1f}")

```

```

X (3, 4), W (2, 4), b (2,)
output (3, 2), max gap 0.0

```

The output is (3, 2), and `allclose` confirms agreement at its default tolerances; this run's maximum gap is 0.0. Notice where the transpose lives: `layer.weight` itself is (D_out, D_in). We transpose it only when writing the row-batch calculation by hand.

B.3. Broadcasting: convenient, but not semantic

Broadcasting lets an elementwise operation combine different shapes without manually copying the smaller tensor. Compare shapes from the trailing axis toward the left. At each position, two sizes are compatible when they are equal, one is 1, or one axis is missing. The result takes the larger compatible size.

For example, (B, D_out) + (D_out,) is exactly the dense-layer bias operation. The bias's only axis aligns with the final feature axis. PyTorch knows sizes—it does not know that one axis means channels and another means image width.

Let us build the more dangerous version of that bug. This NCHW image batch has three channels *and* width three. A bare (3,) channel mean can therefore attach to the wrong axis and still return the expected overall shape.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Expose a silent channel-versus-width broadcasting error.
 3. Report or visualize the measured result.
-

CODE

```

# [1]
images = torch.tensor(
    [
        [[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]],
        [[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]],
        [[3.0, 3.0, 3.0], [3.0, 3.0, 3.0]]
    ]
) # (B=1, C=3, H=2, W=3)

channel_mean = images.mean(dim=(0, 2, 3), keepdim=True)
centered = images - channel_mean
wrong = images - channel_mean.reshape(3)

```

```

correct_check = centered.mean(dim=(0, 2, 3))
wrong_check = wrong.mean(dim=(0, 2, 3))
# [2]
assert torch.equal(correct_check, torch.zeros(3))
assert not torch.equal(wrong_check, torch.zeros(3))

# [3]
print(f"images {tuple(images.shape)}, mean {tuple(channel_mean.shape)}")
print("correct channel means:", correct_check.tolist())
print("wrong channel means:  ", wrong_check.tolist())

```

```

images (1, 3, 2, 3), mean (1, 3, 1, 1)
correct channel means: [0.0, 0.0, 0.0]
wrong channel means:  [-1.0, 0.0, 1.0]

```

Both subtractions return shape (1, 3, 2, 3)—only the invariant reveals the bug: the correct result has channel means [0, 0, 0], while the accidental width subtraction leaves [-1, 0, 1]. The mistake ran because $C=W=3$ —a symmetric test shape hid the axis meaning.

Broadcasting hygiene

1. Write reductions with named `dim` values.
2. Use `keepdim=True` when the result will broadcast back to the original tensor.
3. Test with deliberately unequal axis sizes: $C=3, H=5, W=7$ rather than three convenient equal numbers.
4. Check a semantic invariant after the operation. Shape equality alone is weak evidence.

B.3.1. `expand` and `repeat` are different promises

Ordinary broadcasting is usually enough. When you need an explicitly expanded shape, `expand` returns a **view** and can enlarge singleton axes (or prepend new axes). It sets the expanded axis's stride to zero—several logical positions refer to the same stored value. `repeat`, in contrast, materializes a tiled result with separate stored entries.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Contrast an expanded view with repeated storage.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
base = torch.tensor([[10.0], [20.0]])      # (2, 1)
expanded = base.expand(2, 3)               # view: stride 0 on the new width
repeated = base.repeat(1, 3)              # materialized tiled values

base[0, 0] = 99.0

# [2]
assert expanded[0].tolist() == [99.0, 99.0, 99.0]
assert repeated[0].tolist() == [10.0, 10.0, 10.0]

# [3]
print("expanded stride:", expanded.stride())
print("repeated stride:", repeated.stride())
print("after editing base:", expanded[0].tolist(), repeated[0].tolist())
```

```
expanded stride: (1, 0)
repeated stride: (3, 1)
after editing base: [99.0, 99.0, 99.0] [10.0, 10.0, 10.0]
```

The expanded view has stride (1, 0): moving along its second axis does not move in storage. That aliasing is why an expanded view should not receive in-place writes. Clone it first if independent writable entries are required. Use **repeat** only when you genuinely intend to materialize copies—using it merely to make elementwise arithmetic work wastes the abstraction that broadcasting already provides.

 **Trap:** shape is not layout

A tensor can keep the same shape and values while its strides and storage layout change. **transpose** and **permute** normally return views, so a result can be non-contiguous even when it looks ordinary. Audit the strides before assuming that **view** is legal; use **reshape** or **contiguous** according to the consumer's contract.

B.4. Shape, stride, and contiguity

For the layout question, a dense strided tensor interprets storage through shape, stride, and an offset; dtype determines how the stored bits are read. The stride for an axis says how many storage positions to move when that index increases by one. A contiguous (2, 3, 4) tensor has stride (12, 4, 1): move twelve entries for the first axis, four for the second, and one for the last.

transpose and **permute** reorder axes by changing metadata; they normally return views rather than moving values. The result can be non-contiguous. That is where **view**, **reshape**, and **contiguous** part company.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Follow shape and stride through a permutation.
3. Report or visualize the measured result.

CODE

```
# [1]
original = torch.arange(24).reshape(2, 3, 4)
permuted = original.permute(0, 2, 1)

try:
    permuted.view(2, -1)
except RuntimeError:
    view_result = "view rejected the incompatible strides"
else:
    raise AssertionError("view unexpectedly succeeded")

reshaped = permuted.reshape(2, -1)
materialized = permuted.contiguous().view(2, -1)
# [2]
assert torch.equal(reshaped, materialized)
reshape_copied = (
    reshaped.untyped_storage().data_ptr()
    != permuted.untyped_storage().data_ptr()
)

# [3]
print("original:", tuple(original.shape), original.stride())
print("permuted:", tuple(permuted.shape), permuted.stride(),
      permuted.is_contiguous())
print(view_result)
print("reshape copied in this case:", reshape_copied)
```

```
original: (2, 3, 4) (12, 4, 1)
permuted: (2, 4, 3) (12, 1, 4) False
view rejected the incompatible strides
reshape copied in this case: True
```

The permutation changes (2, 3, 4) with stride (12, 4, 1) into (2, 4, 3) with stride (12, 1, 4). No values moved—the desired flattening no longer follows one compatible run through storage—`view` refuses to pretend otherwise. `reshape` may return a view when strides permit it or a copy when they do not; code should never depend on which one it chooses. `contiguous` makes the copy explicit when a contiguous consumer requires one.

The pointer comparison above is a diagnostic for this fixed example, not a control flow technique. Program logic should depend on values and documented contracts—not on whether one particular `reshape` happened to share storage.

Operation	Axis effect	Storage promise
<code>unsqueeze(dim)</code>	insert a size-one axis	view
<code>squeeze(dim)</code>	remove that axis if its size is one	view
<code>transpose(a, b)</code>	swap two axes	view
<code>permute(order)</code>	reorder all axes	view
<code>flatten(start_dim=...)</code>	merge a consecutive axis range	view or copy
<code>view(shape)</code>	reinterpret compatible strides	view or error
<code>reshape(shape)</code>	request a shape	view or copy
<code>contiguous()</code>	preserve shape and values	original if already contiguous, otherwise copy

Specify the axis in `squeeze(dim)`. An unrestricted `squeeze()` can silently remove the batch axis when `B=1`. Likewise, reserve `.T` for ordinary 2-D matrices. For a batch of matrices, use `transpose(-2, -1)` to say that only the final two axes swap. That is the pattern used by attention throughout the book.

B.5. Products: which axes meet?

Elementwise multiplication and matrix multiplication answer different questions. `A * B` multiplies corresponding broadcastable entries. `A @ B` contracts axes:

- two vectors $(D,)$ `@` $(D,)$ produce a scalar;
- (M, D) `@` (D, N) produces (M, N) ;
- when both operands have at least two axes, `@` treats the final two as matrix axes and broadcasts compatible leading batch axes.

Appendix A develops the geometry of these products in Appendix A. Here our concern is executable bookkeeping. Consider the multi-head attention product from Chapter 14. Let

$$\mathbf{Q} \in \mathbb{R}^{B \times N_h \times T \times D_h}, \quad \mathbf{K} \in \mathbb{R}^{B \times N_h \times S \times D_h}.$$

For each batch, head, query position, and key position, we contract only the shared feature axis:

$$R_{bhts} = \sum_{d=1}^{D_h} Q_{bhtd} K_{bhsd}.$$

The direct matrix expression and Einstein notation should agree.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Verify batched matmul against an indexed contraction.
3. Report or visualize the measured result.

CODE

```
# [1]
generator = torch.Generator().manual_seed(6050)
B, N_HEADS, T, S, D_HEAD = 2, 3, 4, 5, 6
queries = torch.randn(
    B, N_HEADS, T, D_HEAD, generator=generator, dtype=torch.float64
)
keys = torch.randn(
    B, N_HEADS, S, D_HEAD, generator=generator, dtype=torch.float64
)

by_matmul = queries @ keys.transpose(-2, -1)
by_indices = torch.einsum("bhtd,bhsd->bhts", queries, keys)

# [2]
assert by_matmul.shape == (B, N_HEADS, T, S)
assert torch.allclose(by_matmul, by_indices, rtol=0, atol=1e-12)
gap = (by_matmul - by_indices).abs().max().item()
# [3]
print(
    f"Q {tuple(queries.shape)}, K {tuple(keys.shape)}, "
    f"scores {tuple(by_matmul.shape)}"
)
print(f"matmul/einsum max gap: {gap:.1f}")
```

```
Q (2, 3, 4, 6), K (2, 3, 5, 6), scores (2, 3, 4, 5)
matmul/einsum max gap: 0.0
```

The unscaled result shape is (B, N_h, T, S) . After scaling and masking, attention softmax belongs on $\text{dim}=-1$: each query distributes weight over its S keys. A key-validity mask (B, S) becomes $(B, 1, 1, S)$ so it broadcasts across heads and queries. A causal mask (T, S) broadcasts across batch and heads—changing either mask axis changes which information the model may use. Chapters 13–15 derive those modeling choices in [Chapter 13](#), [Chapter 14](#), and [Chapter 15](#).

Read the `einsum` string as an axis audit. The letters `b`, `h`, `t`, and `s` appear in the output—the axes survive. The feature letter `d` appears in both inputs but not the output, so it is multiplied and summed away. This is often the clearest way to check a new contraction—even when the final implementation uses the more familiar `@`.

💡 Translate the operation into one sentence

Before typing a product, say: “For every batch and head, compare every query with every key by summing over features.” That sentence predicts (B, N_h, T, S). If the code produces a different order, fix the contract before adding another `reshape`.

B.6. Indexing and masks

Indexing can either preserve a coordinate system or collapse it. Basic slicing such as `x[:, 1:3]` normally returns a view. Advanced indexing with integer tensors or Boolean masks returns selected values in a new tensor. Both are useful—the mistake is expecting one shape behavior while asking for the other.

Suppose token states have shape (B, T, D) and a visibility mask has shape (B, T). Direct Boolean selection collects the visible rows and drops their original batch-time grid. `masked_fill` keeps the grid—its mask needs a singleton feature axis so it can broadcast across D.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Contrast Boolean selection with shape-preserving masking.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
tokens = torch.arange(2 * 4 * 3).reshape(2, 4, 3)
visible = torch.tensor(
    [[True, True, False, False], [True, False, True, False]]
)

selected = tokens[visible]
masked = tokens.masked_fill(~visible[..., None], -1)

# [2]
assert selected.shape == (4, 3)
assert masked.shape == tokens.shape
assert torch.equal(masked[visible], selected)
# [3]
print(f"tokens {tuple(tokens.shape)}, mask {tuple(visible.shape)}")
print(f"selection {tuple(selected.shape)}, masked tensor {tuple(masked.shape)}")
```

```
tokens (2, 4, 3), mask (2, 4)
selection (4, 3), masked tensor (2, 4, 3)
```

Use selection when the desired object really is “all valid rows,” as when flattening only supervised positions for a loss. Use `masked_fill` or `torch.where` when later operations still need the original axes. Attention masks in Chapters 13–15 preserve the score grid; padding-aware token losses often select or flatten supervised rows. The operation should follow the question, not habit.

B.7. Dtype- and device-aware construction

A factory call creates another contract. `torch.zeros(shape)` chooses defaults; those defaults may not match the tensor already moving through a model. When a new tensor should follow a reference, use a `*_like` factory or a `new_*` method:

- `torch.zeros_like(x)` inherits shape, dtype, layout, and device;
- `x.new_full(shape, value)` inherits dtype and device while choosing a new shape;
- `torch.arange(..., device=x.device)` makes placement explicit for indices.

Gradient status is deliberately separate—a fresh `zeros_like(x)` does not require gradients by default even when `x` does. And not every tensor *should* inherit the reference dtype: masks are Boolean, while class and position indices are normally `torch.int64`.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Construct values, masks, and indices from one reference.
 3. Report or visualize the measured result.
-

CODE

```
# [1]
reference = torch.linspace(-1, 1, 6, dtype=torch.float64).reshape(2, 3)
reference.requires_grad_(True)

accumulator = torch.zeros_like(reference)
thresholds = reference.new_full((1, 3), 0.25)
valid = torch.ones(
    reference.shape[0], dtype=torch.bool, device=reference.device
)
indices = torch.arange(reference.shape[0], device=reference.device)

# [2]
assert accumulator.dtype == thresholds.dtype == reference.dtype
assert accumulator.device == thresholds.device == reference.device
assert not accumulator.requires_grad
assert valid.dtype == torch.bool and indices.dtype == torch.int64

# [3]
print("reference:", reference.dtype, reference.device, reference.requires_grad)
```

```
print("accumulator:", accumulator.dtype, accumulator.device,
      accumulator.requires_grad)
print("semantic dtypes:", valid.dtype, indices.dtype)
```

```
reference: torch.float64 cpu True
accumulator: torch.float64 cpu False
semantic dtypes: torch.bool torch.int64
```

The code prints matching floating-point dtypes and devices for the numerical values, then `bool` and `int64` for the mask and indices. This is the rule—inherit the representation when the new tensor plays the same numerical role; choose explicitly when its semantics differ. Chapter 17 applies that distinction to stored and computed weights (Chapter 17), while Appendix C explains what a floating-point dtype can actually represent.

B.8. A debugging routine

When a shape error—or worse, a suspicious result—appears, resist the urge to add `squeeze` and `reshape` until the exception disappears. Walk the contract instead.

The impossible downstream shape was a clue, not the place to patch. The diagnosis traced upstream, stated the expected matrix product, applied the right-alignment rules, reshaped the intended column explicitly, and asserted its shape. That order matters—the downstream failure was only the symptom.

💡 Six questions before the next edit

1. **Name the axes.** Write # (B, T, D), not merely # 3-D tensor.
2. **Inspect the full representation.** Check `shape`, `dtype`, `device`, `stride()`, `is_contiguous()`, and `requires_grad`.
3. **State the operation in words.** Which axes survive, broadcast, reduce, contract, or reorder?
4. **Check the boundary.** Assert the input and output shapes at module interfaces.
5. **Use awkward test sizes.** Include B=1, make unrelated axes unequal, and use T\ne S for attention.
6. **Check meaning, not only shape.** Probabilities sum to one over the intended axis; centered channels have zero channel means; masks hide exactly the forbidden positions.

The original $(N,) + (N,1)$ bug becomes straightforward under this routine. Name the prediction axis, decide whether the target is $(N,)$ or $(N,1)$, make every term follow that choice, and assert the result before computing a loss. Do not ask PyTorch to infer meaning from matching integers—it cannot.

💡 Practice bridge (non-examinable): from samples to batches

A map-style `Dataset` owns the sample contract: what one index means and which tensors or labels it returns; an `IterableDataset` instead owns a sample stream. A `DataLoader` owns iteration around either contract: sampling order where applicable, batching, collation, and optionally worker processes. That separation turns Chapter 4’s minibatch symbol B into an explicit software boundary (Chapter 4).

Start with `num_workers=0`. The dataset then runs in the main process, so exceptions and random-state mistakes are easiest to diagnose. A positive value asks subprocesses to prepare samples.

PyTorch assigns each worker a seed, but other random libraries and iterable streams still need worker-aware initialization or sharding; every dataset or collation object must also work under the platform’s process-start rules. `pin_memory=True` asks the loader to return tensors in pinned host memory. It does not move a batch to an accelerator, and the flag alone is not evidence of a faster pipeline. Treat worker count and pinning as workload settings, then apply the measurement contract in Appendix C before making a performance claim.

Okay, so the practical bridge is now complete. Shape tells you how many positions exist—axis names tell you what they mean; strides tell you how those positions map to storage. Broadcasting, indexing, and matrix products are then not mysterious rules. Each is an explicit decision about which axes remain and which disappear.

B.9. Where the version-fragile details live

This appendix, like the chapters, prints only **stable semantics**: shapes, masks, reductions, modes, and stability choices that survive framework releases. Everything tied to a particular PyTorch version, backend, or device — kernel selection for fused attention, determinism-flag coverage, autocast dtype policies, thread-count choices, the tested-environment table — lives in the repository’s living compatibility note (`docs/compatibility.md`), which the weekly Execution Audit keeps honest. If a printed number ever disagrees with a fresh run, start there: the semantics in these pages should not have moved, but the environment may have.

Sources and further reading

- [PyTorch broadcasting semantics](#): the exact trailing-axis compatibility rules.
- [PyTorch tensor views](#): views, strides, contiguity, basic versus advanced indexing, and the conditional copy behavior of `reshape`.
- [PyTorch data loading](#): the `Dataset` and `DataLoader` contracts, worker processes, collation, and memory pinning.
- [`torch.nn.Linear`](#): the official input, output, weight, and bias shape contract.
- [`torch.einsum`](#): executable indexed contractions.
- [PyTorch tensor attributes](#): dtype, device, and layout references.

Exercises

1. (**Pencil.**) Starting from the token-logit convention in Table B.1, let `scores` have shape (B, T, V). Determine the result shape of each operation and name every remaining axis: `scores.mean(dim=1)`, `scores.mean(dim=1, keepdim=True)`, `scores.argmax(dim=-1)`, and `scores.reshape(-1, V)`. Which operations lose the original time coordinates, and how do reduction and axis merging lose them differently?
2. (**Code.**) Write `standardize_channels(x)` for an NCHW tensor. Compute one mean and standard deviation per channel with `keepdim=True`, return the standardized tensor, clamp the standard deviation away from zero, and assert that its per-channel means are zero. Test (B,C,H,W)=(2,3,5,7) and the deceptive case (1,3,2,3).

3. **(Code.)** Start with a sequence tensor of shape (B, T, N_h, D_h) . Reshape and permute it into (B, N_h, T, D_h) , then reverse the operation. Assert exact equality with the input for $B=2$, $T=5$, $N\_h=3$, and $D\_h=4$. Print the stride after every step and identify where a copy may occur.
4. **(Pencil.)** (a) For x shaped $(4, 1)$, compare $x.expand(4, 1000)$ and $x.repeat(1, 1000)$. Predict their strides and how editing the base changes each result. **(Code.)** (b) Verify those predictions, then explain why an in-place write through the expanded view is unsafe.
5. **(Code.)** Let logits have shape (B, T, V) and a validity mask have shape (B, T) . Produce (a) selected logits shaped (N_valid, V) and (b) shape-preserving logits in which invalid rows are all zero. Test with $B=2$, $T=4$, and $V=5$, and assert that both forms agree on the valid rows.

C. Numerical Precision and Hardware Efficiency

Suppose a training step computes a gradient of 10^{-8} . The gradient is nonzero—the learning rate is nonzero, and their product is nonzero. Yet a stored weight can remain exactly unchanged. Now suppose an attention implementation performs the same mathematical operation as another one—with essentially the same arithmetic count—but finishes much sooner because it moves fewer bytes. Neither story is visible from the algebra alone—it omits both rounding and movement.

That is why numerical precision and hardware efficiency belong together. A dtype says which values a tensor can store—it does not, by itself, say how every operation is evaluated. A FLOP count says how much arithmetic an algorithm requests—it does not say how often data crosses a slow memory boundary. We need contracts for both.

This appendix owns the first-course mechanics. The companion volume *Deep Learning: Making It Trainable* turns the same objects into graduate diagnostic instruments: begin with [count versus cost](#), then make the [precision contract quantitative](#).

Appendix B established that shape, dtype, device, and layout are separate parts of a tensor’s representation (Appendix B). Here we open the dtype field—and follow its consequences through rounding, stable computation, mixed precision, the Roofline model, and one especially useful case study: FlashAttention.

C.1. One operation, several precision contracts

The phrase “this model uses FP16” is too compressed to audit. For one operation, ask four questions instead.

Contract	What to record	Why it can differ
Operand storage	dtype used to store the inputs	Parameters and activations may occupy fewer bytes
Evaluation or product	precision used for products or nonlinear functions	A kernel may round inputs or choose an operation-specific dtype
Accumulation	dtype used for a running sum or reduction	Dot products may accumulate low-precision products into a wider dtype
Output storage	dtype used to store the result	A wider accumulator may be rounded back to a narrower tensor

A matrix multiply can therefore read two FP16 tensors—then form products from those inputs, accumulate the dot products in FP32, and store an FP16 output. An optimizer can update an FP32 parameter even when selected forward operations use BF16. These are not edge cases—they are the point of mixed precision.

This extends the distinction in [Chapter 17](#): **storage precision is not compute precision, and neither one identifies the trainable state**. It also prevents a performance shortcut. A two-byte tensor is smaller than a four-byte tensor, but that alone does not prove a twofold speedup. The device needs an efficient kernel for the operation, and the operation needs to be limited by something those saved bytes actually relieve.

C.2. What a binary float can represent

A normal binary floating-point value has the form

$$x = (-1)^s (1.f)_2 \times 2^{e-\text{bias}}. \quad (\text{C.1})$$

The sign bit s chooses the sign. The exponent field e buys **range**—access to very large and very small scales. The fraction field f buys **resolution**—more representable points within one scale. “Mantissa” is common informal vocabulary for the significand, but fraction bits is the more precise name for the stored trailing field. Normal values receive an implicit leading 1, so a format with 23 stored fraction bits has 24 bits of significand precision.

Format	Bytes	Exponent bits	Fraction bits	Spacing above 1	Smallest positive normal	Largest finite
FP64	8	11	52	2^{-52}	2^{-1022}	about 1.80×10^{308}
FP32	4	8	23	2^{-23}	2^{-126}	about 3.40×10^{38}
FP16	2	5	10	2^{-10}	2^{-14}	65,504
BF16	2	8	7	2^{-7}	2^{-126}	about 3.39×10^{38}

FP16 and BF16 spend the same 16 bits differently. FP16 gives three more fraction bits, so its grid is eight times finer near 1. BF16 keeps the eight-bit exponent field of FP32, so its range is in the same class as FP32, but its local grid is coarse. The right question is not “which 16-bit format is more precise?” It is “does this calculation need more range or more resolution, and where?”

`torch.info` lets us inspect the actual dtype contract. Its **eps** is the distance from 1 to the next larger representable value. It is not the smallest positive value; **tiny** is the smallest positive **normal** value.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Inspect range and spacing for four floating-point dtypes.
-

CODE

```

import math

import matplotlib.pyplot as plt
from matplotlib.patches import FancyArrowPatch, Rectangle
import torch
from torch import nn

# [1]
torch.manual_seed(6050)
torch.set_default_dtype(torch.float64)
torch.set_num_threads(1)

dtype_names = {
    torch.float16: "FP16",
    torch.bfloat16: "BF16",
    torch.float32: "FP32",
    torch.float64: "FP64",
}

# [2]
for dtype in (torch.float16, torch.bfloat16, torch.float32, torch.float64):
    info = torch.finfo(dtype)
    print(
        f"{dtype_names[dtype]:>4}: eps={info.eps:.8e}, "
        f"tiny={info.tiny:.8e}, max={info.max:.8e}"
    )

for dtype in (torch.float16, torch.bfloat16, torch.float32):
    one = torch.tensor(1.0, dtype=dtype)
    next_one = torch.nextafter(one, torch.tensor(float("inf"), dtype=dtype))
    assert next_one - one == torch.finfo(dtype).eps

wide = torch.tensor(70_000.0)
small = torch.tensor(2.0**-26)
print("70,000 stored as FP16 / BF16:",
      wide.to(torch.float16).item(), wide.to(torch.bfloat16).item())
print("2^-26 stored as FP16 / BF16:",
      small.to(torch.float16).item(), small.to(torch.bfloat16).item())

```

```

FP16: eps=9.76562500e-04, tiny=6.10351562e-05, max=6.55040000e+04
BF16: eps=7.81250000e-03, tiny=1.17549435e-38, max=3.38953139e+38
FP32: eps=1.19209290e-07, tiny=1.17549435e-38, max=3.40282347e+38
FP64: eps=2.22044605e-16, tiny=2.22507386e-308, max=1.79769313e+308
70,000 stored as FP16 / BF16: inf 70144.0
2^-26 stored as FP16 / BF16: 0.0 1.4901161193847656e-08

```

The printed spacings are 9.765625×10^{-4} for FP16, 7.8125×10^{-3} for BF16, and about 1.19×10^{-7} for FP32. The range-resolution trade is visible in the last two lines. FP16 sends 70,000 to infinity and 2^{-26}

to zero; BF16 stores them as 70,144 and 1.4901161×10^{-8} . BF16 reached both scales, but 70,000 moved to the nearest point on its coarse grid.

The spacing is not constant across the number line. Within one exponent bin it is fixed; when the exponent grows by one, the spacing doubles. Near 1, subtracting 10^{-8} from an FP32 value can therefore disappear—even though FP32 can represent 10^{-8} perfectly well near zero. Representability depends on the value’s neighborhood, not just on the isolated update.

i Normals, subnormals, infinities, and NaNs

The all-zero exponent field represents zero and **subnormal** values, which fill part of the gap below the smallest normal number with reduced precision. The all-one exponent field represents infinities and NaNs in the IEEE formats used here. Hardware and kernel behavior around subnormals can vary—some accelerator paths flush them to zero—so **tiny** is a portable normal-range landmark, not a promise about the smallest value every execution path preserves.

Two other names appear frequently in deep learning—**TF32** is a matrix-compute mode, not a tensor storage dtype: supported operations keep FP32 storage and range while using reduced input precision for multiplication and wider accumulation. **FP8** is a family rather than one universal format; E4M3 and E5M2 allocate their few exponent and fraction bits differently and are normally used with scales and wider accumulators. Chapter 17 treats those choices as quantization policies rather than magical calls to `tensor.to(...)` (Chapter 17).

C.3. Rounding is part of the algorithm

Real-number algebra lets us reassociate sums, cancel equal terms, and treat every nonzero magnitude as distinct. Floating-point arithmetic does not. After an operation, the result is rounded to an available grid point—overflow, underflow, and ordering can therefore change the computation.

The familiar $0.1 + 0.2$ example is the smallest witness. One tenth and one fifth have nonterminating binary expansions, so their stored approximations add to the FP64 value printed as `0.30000000000000004`, not the stored approximation printed as `0.3`. The real-number identity is intact—the finite encodings and rounding steps are different.

C.3.1. A nonzero update can become no update

Let a stored weight be $w = 1$. If the exact update Δw is less than roughly half the local spacing, rounding to nearest can return the original bit pattern:

$$\text{fl}(w - \Delta w) = w.$$

That is different from an update underflowing to zero by itself. The update can be representable near zero and still be too small to change a much larger stored value. The depth experiment in Chapter 5 and the norm diagnostic in Chapter 9 encountered these two failure modes from different directions.

PLAN

1. Separate rounded-away updates, accumulation loss, and cancellation.

CODE

```
# [1]
for dtype, update in (
    (torch.float16, 1e-4),
    (torch.bfloat16, 1e-3),
    (torch.float32, 1e-8),
):
    weight = torch.tensor(1.0, dtype=dtype)
    changed = weight - torch.tensor(update, dtype=dtype)
    print(f"{dtype_names[dtype]}: 1 - {update:g} -> {changed.item():g}")
    assert changed == weight

term = torch.tensor(0.125, dtype=torch.float16)
sum16 = torch.tensor(0.0, dtype=torch.float16)
sum32 = torch.tensor(0.0, dtype=torch.float32)
for _ in range(10_000):
    sum16 += term
    sum32 += term.float()

large = torch.tensor(1e8, dtype=torch.float32)
one = torch.tensor(1.0, dtype=torch.float32)
cancel_first = (large - large) + one
small_first = (large + one) - large

print("10,000 serial additions in FP16 / FP32:",
      sum16.item(), sum32.item())
print("cancellation orders:", cancel_first.item(), small_first.item())
```

```
FP16: 1 - 0.0001 -> 1
BF16: 1 - 0.001 -> 1
FP32: 1 - 1e-08 -> 1
10,000 serial additions in FP16 / FP32: 256.0 1250.0
cancellation orders: 1.0 0.0
```

All three stored weights remain 1. The deliberately serial FP16 accumulator stops at 256, while the FP32 accumulator reaches the exact sum 1,250. This loop isolates the accumulator's role—production reductions may use trees, fused instructions, or wider accumulators and need not reproduce 256. Finally, $(10^8 - 10^8) + 1$ returns 1 in FP32, but $(10^8 + 1) - 10^8$ returns 0 because the 1 was rounded away before subtraction. Algebraically equal expressions acquired different histories.

C.3.2. Stable algebra prevents avoidable overflow

For logits $\mathbf{z}$, direct softmax forms exponentials that may overflow:

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

Subtracting $m = \max_j z_j$ leaves the ratio unchanged and ensures that the largest exponent is 1:

$$p_i = \frac{e^{z_i - m}}{\sum_j e^{z_j - m}}. \quad (\text{C.2})$$

This is the log-sum-exp trick from [Chapter 2](#). Let us separate what it can repair from information already lost at storage time.

PLAN

1. Contrast stable softmax with overflow and prior rounding.
-

CODE

```
# [1]
for dtype in (torch.float16, torch.bfloat16, torch.float32):
    logits = torch.tensor([1000.0, 1001.0, 1002.0], dtype=dtype)
    exponentials = torch.exp(logits)
    naive = exponentials / exponentials.sum()
    stable = torch.softmax(logits, dim=0)
    print(f"{dtype_names[dtype]} stored logits: {logits.tolist()}")
    print(" naive finite:", bool(torch.isfinite(naive).all()),
          "stable:", [round(x, 4) for x in stable.float().tolist()])
```

```
FP16 stored logits: [1000.0, 1001.0, 1002.0]
 naive finite: False stable: [0.09, 0.2448, 0.665]
BF16 stored logits: [1000.0, 1000.0, 1000.0]
 naive finite: False stable: [0.334, 0.334, 0.334]
FP32 stored logits: [1000.0, 1001.0, 1002.0]
 naive finite: False stable: [0.09, 0.2447, 0.6652]
```

The naïve calculation is nonfinite in all three dtypes. Stable softmax prints `[0.0900, 0.2448, 0.6650]` in FP16 and `[0.0900, 0.2447, 0.6652]` in FP32—the latter is the expected vector to four decimals. BF16 returns an approximately uniform vector—not because stable softmax failed, but because 1000, 1001, and 1002 were all stored as 1000 **before** softmax began. Stable algebra can avoid unnecessary overflow; it cannot reconstruct distinctions that the input representation erased.

The same principle appears across the book:

- Kernel regression in [Chapter 12](#) subtracts a maximum before exponentiating so a tiny bandwidth does not turn every weight into zero and produce 0/0.
- Chapter 10’s default forget-gate factor compounds to $0.5^{80} = 8.27 \times 10^{-25}$ ([Chapter 10](#)). That value is still nonzero in FP32—the architectural signal can be unusably attenuated before representation turns it into zero.
- Long products such as the diffusion schedule in [Chapter 19](#) are often more legible as sums of logarithms. A coefficient can be tiny without being zero; inspect the chosen dtype before replacing mathematics with a special case.
- Linear systems should be solved directly rather than by explicitly forming an inverse ([Appendix A](#)). Higher precision can reduce rounding error, but it does not cure an ill-conditioned question.

 “Use more precision” is not a stability proof

A wider dtype postpones some failures. It does not repair a poor formulation, make subtraction well conditioned, or guarantee finite intermediate values. First choose a stable algorithm—log-sum-exp, a direct solve, a protected denominator—then choose a precision contract that preserves the scales the algorithm must carry.

C.4. Mixed precision is a policy

Mixed precision assigns different numerical jobs to different dtypes. The original FP16 training recipe used low-precision working tensors, FP32 accumulation where needed, FP32 master weights for small optimizer updates, and **loss scaling** for tiny gradients. Current frameworks choose operation-specific policies—but the reason for the pieces remains useful.

BF16 changes the balance. Its FP32-like exponent range makes gradient underflow much less common than in FP16, so BF16 training often does not need loss scaling. Its seven fraction bits still give coarse local spacing—wide range does not protect a small update applied to a much larger BF16 value.

For a positive scale S and loss L ,

$$\nabla_{\theta}(SL) = S\nabla_{\theta}L.$$

Backward propagation on SL moves small gradients into FP16’s representable range. Before the optimizer update, division by S restores their intended scale. Dynamic loss scaling lowers S after detected overflow and may grow it after successful steps. It cannot restore a forward activation that already overflowed or an optimizer update that was rounded away in narrow master state.

The next check uses no accelerator timing—it isolates the arithmetic, then asks CPU autocast which dtypes it chose for one linear layer in this environment.

PLAN

1. Prepare the inputs and fixed settings for the example.
2. Expose loss scaling and operation-specific CPU autocast dtypes.

CODE

```
# [1]
tiny_gradient = torch.tensor(2.0**-26, dtype=torch.float32)
scale = 2.0**12
direct = tiny_gradient.to(torch.float16).to(torch.float32)
recovered = (
    (tiny_gradient * scale).to(torch.float16).to(torch.float32) / scale
)
assert direct == 0 and recovered == tiny_gradient
print("gradient direct / scaled-then-unscaled:",
      direct.item(), recovered.item())

torch.manual_seed(6050)
layer = nn.Linear(4, 2, dtype=torch.float32)
inputs = torch.randn(3, 4, dtype=torch.float32)
with torch.autocast("cpu", dtype=torch.bfloat16):
    outputs = layer(inputs)
# [2]
outputs.float().square().mean().backward()

print("input, parameter, output, parameter-gradient dtypes:")
print(inputs.dtype, layer.weight.dtype, outputs.dtype, layer.weight.grad.dtype)
```

```
gradient direct / scaled-then-unscaled: 0.0 1.4901161193847656e-08
input, parameter, output, parameter-gradient dtypes:
torch.float32 torch.float32 torch.bfloat16 torch.float32
```

Direct FP16 storage turns 2^{-26} into zero; scaling by 2^{12} , rounding to FP16, and unscaling recovers 2^{-26} in this controlled case. In the autocast check, the input and parameter remain FP32, the linear output is BF16, and the parameter gradient is FP32. That observed policy is evidence about this operation and build—not a promise that every operation, device, or future release makes the same choice.



Audit mixed precision by role

Record parameter dtype, activation dtype at important boundaries, reduction or accumulator dtype when the API exposes it, gradient dtype, optimizer-state dtype, and the loss-scaling policy. Do not use “FP16 training” as a substitute for that list. Also distinguish arithmetic accumulation inside a dot product from **gradient accumulation** across microbatches; they share a word, not a mechanism.

C.5. Work is not time: the Roofline model

Why can two mathematically equivalent programs have different speed? Imagine a chef whose arithmetic station is extremely fast but whose ingredients arrive through one narrow door. More cooks at the station do not help when every recipe waits at the door. Other recipes reuse each delivered ingredient many times and keep the station busy. Hardware has the same tension between computation and data movement.

Choose a memory boundary—often device memory to on-chip cache or SRAM—and count:

$$I = \frac{\text{useful floating-point operations}}{\text{bytes crossing the chosen boundary}} \quad \text{FLOP/byte.} \quad (\text{C.3})$$

This ratio is **operational intensity**. “Arithmetic intensity” is often used informally for the same idea, though the original Roofline terminology emphasizes measured traffic at a stated boundary. Let $P_{\max}$ be peak compute throughput and let β be sustainable bandwidth across that boundary. The basic Roofline bound is

$$P_{\text{attainable}}(I) \leq \min(P_{\max}, \beta I). \quad (\text{C.4})$$

The sloped roof βI is the bandwidth limit; the flat roof $P_{\max}$ is the compute limit. They meet at the **ridge point**

$$I_* = \frac{P_{\max}}{\beta}. \quad (\text{C.5})$$

A kernel left of I_* is bandwidth-bound relative to these two roofs—a kernel right of it is compute-bound relative to them. “Relative to” matters: another ceiling—an unsupported shape, launch overhead, synchronization, instruction mix, or a different memory level—may sit lower.

The following figure uses an invented machine with a 100-TFLOP/s compute roof and a 2-TB/s bandwidth roof. It is a worked coordinate system, not a benchmark of this computer or a claim about any product.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Build and interrogate a synthetic Roofline.
-

CODE


```

# [1]
peak_tflops = 100.0
bandwidth_tbps = 2.0
ridge = peak_tflops / bandwidth_tbps

intensities = torch.logspace(-1, 3, 500)
bandwidth_roof = bandwidth_tbps * intensities
combined_roof = torch.minimum(
    bandwidth_roof, torch.full_like(bandwidth_roof, peak_tflops)
)

examples = {
    "elementwise-style": 0.25,
    "reuse growing": 8.0,
    "compute roof": 80.0,
}

# [2]
for name, intensity in examples.items():
    bound = min(peak_tflops, bandwidth_tbps * intensity)
    print(f"{name:>17}: I={intensity:g}, bound={bound:g} TFLOP/s")
print(f"ridge point: {ridge:g} FLOP/byte")

fig, ax = plt.subplots(figsize=(7.6, 4.6))
ax.loglog(intensities, bandwidth_roof, color="#E57200", lw=2.2,
          ls="--", label="bandwidth roof: 2 TB/s")
ax.loglog(intensities, torch.full_like(intensities, peak_tflops),
          color="#232D4B", lw=2.2, ls=":", label="compute roof: 100 TFLOP/s")
ax.loglog(intensities, combined_roof, color="#2E7D32", lw=3.0,
          label="attainable upper bound")
ax.axvline(ridge, color="#722F37", lw=1.3, alpha=0.8)
ax.text(ridge * 1.12, 0.17, "ridge = 50 FLOP/byte",
        color="#722F37", rotation=90, va="bottom")

for index, (name, intensity) in enumerate(examples.items()):
    bound = min(peak_tflops, bandwidth_tbps * intensity)
    ax.scatter(intensity, bound, s=48, color="#232D4B", zorder=4)
    factor = (1.35, 0.72, 0.67)[index]
    ax.annotate(name, (intensity, bound), xytext=(7, 10 * factor),
                textcoords="offset points", fontsize=9)

ax.set_xlim(0.1, 1000)
ax.set_ylim(0.1, 220)
ax.set_xlabel("operational intensity (FLOP/byte)")
ax.set_ylabel("attainable throughput (TFLOP/s)")
ax.grid(True, which="both", color="#D6DCE5", lw=0.6, alpha=0.7)
ax.legend(loc="lower center", bbox_to_anchor=(0.5, 1.01),
          ncol=3, frameon=False, fontsize=8.5)
plt.tight_layout()
plt.show()

```

elementwise-style: $I=0.25$, bound= 0.5 TFLOP/s
 reuse growing: $I=8$, bound= 16 TFLOP/s
 compute roof: $I=80$, bound= 100 TFLOP/s
 ridge point: 50 FLOP/byte

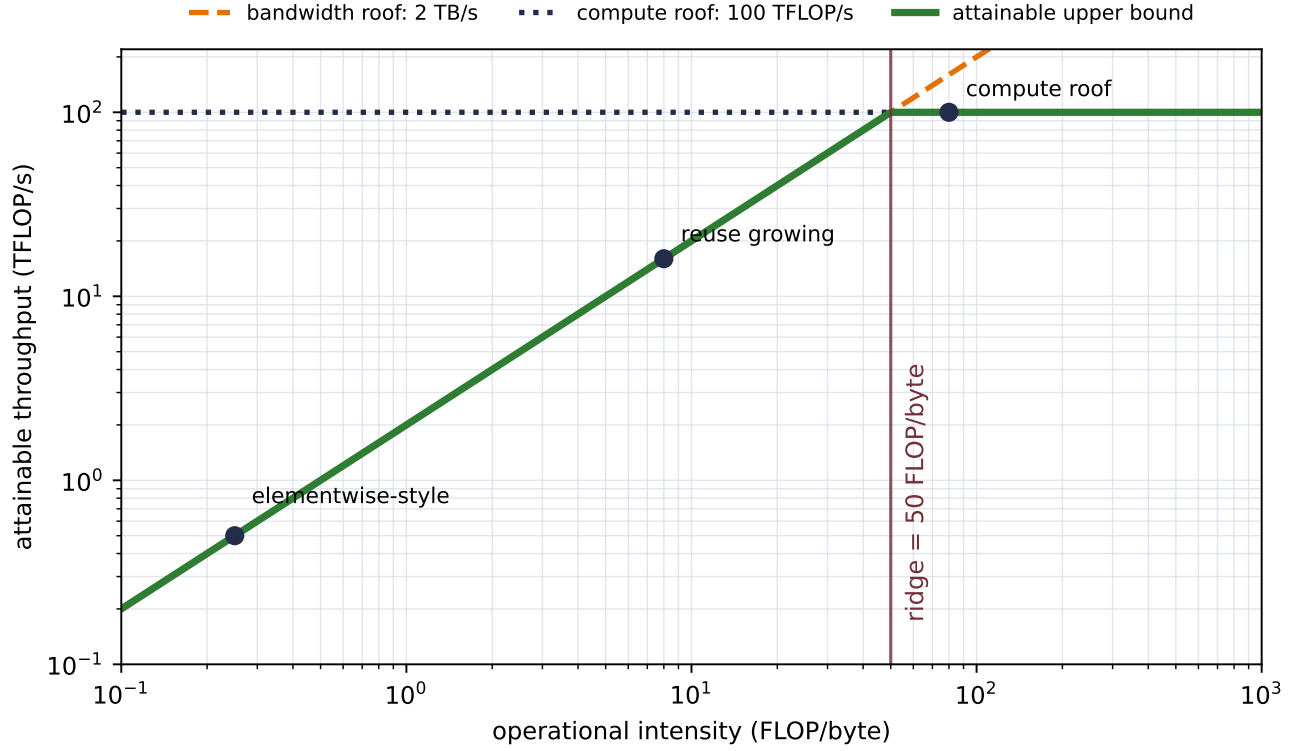


Figure C.1.: A synthetic Roofline with a 100-TFLOP/s compute roof and a 2-TB/s bandwidth roof. Their intersection is 50 FLOP/byte. The three labeled points illustrate a low-intensity elementwise-style kernel, a kernel still limited by the bandwidth roof, and a kernel at the compute roof; they are teaching coordinates, not measurements.

C.5.1. Matrix multiplication earns reuse

Consider $\mathbf{C} = \mathbf{AB}$ with shapes $m \times k$, $k \times n$, and $m \times n$. Count a multiply-add as two FLOPs. An optimistic compulsory-traffic estimate—read each input once and write the output once—gives

$$I_{\text{GEMM,ideal}} = \frac{2mkn}{s(mk + kn + mn)}, \quad (\text{C.6})$$

where s is bytes per stored element. For square $n \times n$ matrices this reduces to $2n/(3s)$. At $n = 128$, the estimate is 21.3 FLOP/byte for FP32 and 42.7 for a two-byte format. On the synthetic machine in Figure C.1, both remain left of the 50-FLOP/byte ridge. At $n = 2048$, the ideal estimates rise to 341.3 and 682.7; reuse is now high enough to reach the flat roof if the implementation realizes the ideal traffic.

This is why matrix multiplication is such a valuable computational primitive. Larger dense products can reuse values many times after paying to move them. A simple elementwise transform with about two

operations, one FP32 read, and one FP32 write has an intensity near $2/8 = 0.25$ FLOP/byte—little arithmetic is available to hide the doorway cost.

Lower precision can change both axes of this analysis. Fewer bytes can raise operational intensity, while specialized low-precision instructions can raise the compute roof. The ridge moves when either roof moves. Scales, casts, wider outputs, optimizer state, alignment, and unsupported shapes also consume resources. “Half the bytes” is therefore an input to a model, not a timing conclusion.

 A Roofline is a bound, not a stopwatch

State the memory boundary, dtype, operation family, shapes, and roofs. Prefer measured traffic and sustainable bandwidth when diagnosing a real run. A point below the roof does not prove why it is below; it tells you which ceilings are still possible and where to investigate next.

 Systems bridge (non-examinable): cache the past at generation time

For a causal Transformer with stable positions and no eviction, a new token cannot change earlier representations. A **key/value (KV) cache** therefore retains each layer’s earlier projected rows and appends one new pair. In Chapter 14’s regression language, this is not a separate mysterious memory: **the KV cache is the nonparametric estimator’s dataset**. The new query traverses those rows, while the old rows do not have to be projected again.

Let L be the layer count, B the batch size, h_{KV} the number of key/value heads, t the cached length, d_h the head width, and s the bytes per stored element. The cache ledger is

$$N_{KV} = 2LBh_{KV}td_h \text{ elements}, \quad M_{KV} = 2sLBh_{KV}td_h \text{ bytes}.$$

Standard multi-head attention commonly uses $h_{KV} = h_Q$: every query head has its own key and value head. **Grouped-query attention (GQA)** instead lets several query heads share one key/value head. If $h_{KV} < h_Q$, the cache becomes

$$\frac{M_{KV,GQA}}{M_{KV,MHA}} = \frac{h_{KV}}{h_Q}.$$

This changes the number of stored and projected key/value heads. It does not change causal visibility, make the prefix shorter, or eliminate the score traversal over t . The grouping is therefore a serving-cost choice with an empirical quality and throughput trade-off, not a different masking rule.

Thus caching spends storage that grows linearly with context length. For one new token, dense attention still reads a score row whose length grows with t ; across a generation, that work is quadratic in final length. Caching avoids re-projecting and re-attending old queries. That is why Chapter 14’s naive sampler, which rebuilds its current context on every step, is wasteful. Sliding-window eviction and position reindexing need an explicit cache policy, so that toy loop cannot simply reuse rows unchanged. Chapter 17 adds this linear-size cache to the total-memory ledger ([Chapter 17](#)).

C.6. FlashAttention through the Roofline lens

For one dense attention head, [Chapter 13](#) and [Chapter 14](#) compute

$$\mathbf{O} = \text{softmax}_{\text{row}} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} + \mathbf{M} \right) \mathbf{V}, \quad (\text{C.7})$$

where N is sequence length, d is head width, and $\mathbf{M}$ carries any mask. The two matrix products require roughly $4N^2d$ FLOPs under the same two-FLOPs-per-multiply-add convention, plus $O(N^2)$ work for masking and softmax.

A straightforward sequence of kernels materializes a score matrix $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top/\sqrt{d}$, writes it to device memory, reads and normalizes it into a probability matrix $\mathbf{P}$, writes $\mathbf{P}$, then reads it again for $\mathbf{P}\mathbf{V}$. The $N \times N$ intermediates are the expensive part of the ledger.

For one head with $d = 64$ and a two-byte dtype, the tensor-shape counts below exclude any materialized mask. An implicit causal rule and a stored dense bias have different ledgers.

Sequence length N	Scores + probabilities	Q, K, V, O combined
512	1 MiB	0.25 MiB
2,048	16 MiB	1 MiB
4,096	64 MiB	2 MiB

At $N = 2048$, one $N \times N$ matrix is 8 MiB per head. Separate score and probability arrays are 16 MiB per head, or 512 MiB across 32 heads. These are logical tensor sizes—not allocator measurements, backward-pass totals, or exact traffic—and they make the scaling contrast visible: the state vectors grow linearly in N , while the materialized attention grid grows quadratically.

C.6.1. The quick recap: tile, normalize online, and fuse

FlashAttention computes exact dense attention without materializing the full quadratic intermediates in high-bandwidth memory. Exact here means that it does not replace attention with a sparse, low-rank, or kernel approximation. Floating-point operation order changes, so bitwise equality is not promised.

The algorithm tiles Q , K , and V to fit blocks in faster on-chip memory. For one query row, it can process score blocks while maintaining three pieces of state: a running maximum m , a running exponential sum ℓ , and an unnormalized output vector $\mathbf{o}$. Given a new score block $\mathbf{s}_b$ and matching value block $\mathbf{V}_b$,

$$\begin{aligned} m' &= \max \left(m, \max_j s_{b,j} \right), \\ \ell' &= e^{m-m'} \ell + \sum_j e^{s_{b,j}-m'}, \\ \mathbf{o}' &= e^{m-m'} \mathbf{o} + \sum_j e^{s_{b,j}-m'} \mathbf{V}_{b,j}. \end{aligned} \quad (\text{C.8})$$

Initialize $m = -\infty$, $\ell = 0$, and $\mathbf{o} = \mathbf{0}$; after all blocks, return $\mathbf{o}/\ell$. When a later block introduces a larger maximum, the factor $e^{m-m'}$ rescales the old state into the new coordinate system. It is the stable softmax trick carried across tiles. The scores $\mathbf{s}_b$ include any mask; a fully masked tile must be skipped or handled specially before taking its maximum. The recurrence also assumes every query has at least

one visible key—an all-masked row needs an explicit policy rather than arithmetic on an all- $-\infty$ score row.

The next cell is deliberately a one-row-at-a-time teaching implementation—not a FlashAttention kernel. It verifies the online identity on CPU; the following figure then diagrams the I/O change independently.

PLAN

1. Prepare the inputs and fixed settings for the example.
 2. Verify materialized attention against online blockwise attention.
 3. Check the claimed identities, shapes, or invariants.
 4. Report or visualize the measured result.
-

CODE

```
# [1]
generator = torch.Generator().manual_seed(6050)
N, D_HEAD, D_VALUE, BLOCK = 7, 4, 3, 3
queries = torch.randn(N, D_HEAD, generator=generator)
keys = torch.randn(N, D_HEAD, generator=generator)
values = torch.randn(N, D_VALUE, generator=generator)

scores = queries @ keys.T / math.sqrt(D_HEAD)
dense_output = torch.softmax(scores, dim=-1) @ values

streamed_rows = []
# [2]
for query in queries:
    running_max = torch.tensor(float("-inf"))
    running_sum = torch.tensor(0.0)
    running_output = torch.zeros(D_VALUE)
    for start in range(0, N, BLOCK):
        stop = min(start + BLOCK, N)
        block_scores = query @ keys[start:stop].T / math.sqrt(D_HEAD)
        new_max = torch.maximum(running_max, block_scores.max())
        old_scale = torch.exp(running_max - new_max)
        block_weights = torch.exp(block_scores - new_max)
        running_sum = old_scale * running_sum + block_weights.sum()
        running_output = (
            old_scale * running_output + block_weights @ values[start:stop]
        )
        running_max = new_max
    streamed_rows.append(running_output / running_sum)

streamed_output = torch.stack(streamed_rows)
gap = (dense_output - streamed_output).abs().max().item()
# [3]
assert torch.allclose(dense_output, streamed_output, rtol=0, atol=1e-12)
```

```
# [4]
print(f"dense/online maximum gap: {gap:.3e}")
print(f"full score grid: {N*N} entries; one-row block: at most {BLOCK}")
```

```
dense/online maximum gap: 4.441e-16
full score grid: 49 entries; one-row block: at most 3
```

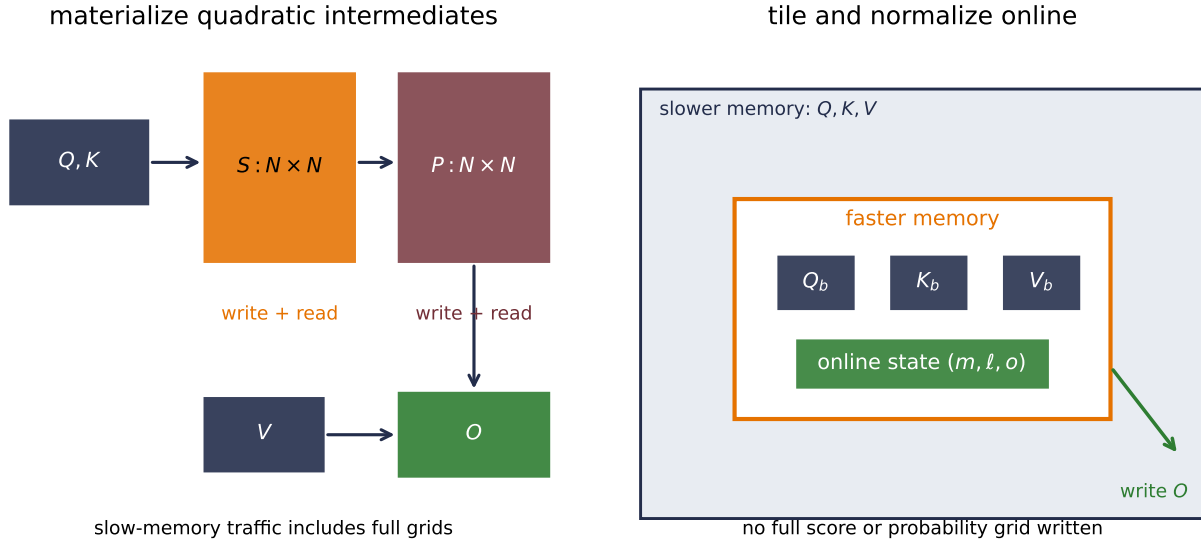


Figure C.2.: Materialized attention and tiled online attention evaluate the same dense rule but schedule its intermediates differently. The left path combines the probability grid with V after writing and rereading full score and probability grids. The right path keeps Q, K , and V tiles plus the online-softmax state close to the arithmetic, then writes the linear-size output. The diagram is conceptual; it does not claim measured traffic for a particular kernel.

The maximum difference is 4.441×10^{-16} in FP64. That is agreement to the expected scale, not bitwise identity. A full 7×7 score grid contains 49 entries; the pedagogical loop holds at most three scores for one query at a time. Real FlashAttention tiles multiple query rows and is engineered around a target memory hierarchy, parallel work partition, masks, and backward propagation.

The Roofline connection is now concrete. FlashAttention does not make dense attention linear-time: its dominant arithmetic remains $O(N^2d)$. It changes the **schedule**. Tiling, fusion, online softmax, and backward recomputation reduce reads and writes of quadratic intermediates across the expensive memory boundary. Some extra arithmetic can be a good trade when it avoids much more costly movement.

FlashAttention-2 keeps that I/O-aware idea and improves work partitioning, sequence- dimension parallelism, and the balance between matrix and non-matrix operations. Its paper’s device-specific speedups are measurements under stated conditions—not portable constants. Likewise, PyTorch’s `scaled_dot_product_attention` may select a FlashAttention backend, another fused backend, or a mathematical fallback depending on device, dtype, dimensions, masks, and software build. Calling the function alone does not prove which kernel ran.

i What FlashAttention does not change

The attention rule, causal or padding mask semantics, and dense quadratic arithmetic remain. “Exact” means no modeling approximation; floating-point order still changes. Longer context is not free, and an implementation that is faster for one shape or device may not be faster for another. In the **test-time-regression interlude**’s terms, FlashAttention reorganizes the **same nonparametric solve** to reduce traffic: it changes the schedule, not the statistics, and the KV dataset still grows. Linear-attention and delta-style solvers make a different bargain. They change the statistical contract by compressing history into a fixed-size state, accepting loss or restriction for constant storage in the prefix length.

C.7. A measurement contract

Precision and Roofline analysis help us ask sharper questions, but each performance claim still needs evidence matched to the quantity.

Claim	Evidence to collect
“The checkpoint is smaller”	Serialized bytes under a named format
“The live tensors use less memory”	Peak allocated/resident memory under a named workload
“The operation does less arithmetic”	An explicit operation-count convention or profiler
“The kernel moves fewer bytes”	A boundary-specific I/O model or hardware counters
“The kernel is faster”	Synchronized, warmed, repeated timings on named hardware and software
“The system has higher throughput”	End-to-end examples or tokens per second at a stated batch and sequence shape

Record dtype, tensor shapes, backend and kernel selection, hardware, warmup, synchronization, compile state, and summary statistic. Separate latency from throughput and kernel time from end-to-end time. A smaller checkpoint is not automatically faster, just as the dot-product proxy in **Chapter 16** was not a wall-clock measurement.

The studies in this appendix intentionally establish representation limits, analytical bounds, and CPU equality checks. They do not establish a hardware speedup. That honesty line is part of the result—not a footnote to add after an attractive number appears.

Okay, so—we can now read a performance story in two passes. First audit the numerical contract: what is stored, how products and reductions are evaluated, and where rounding can erase signal. Then audit the movement contract: how many useful operations are performed, how many bytes cross which boundary, and which roof can limit the result. FlashAttention is memorable because it joins the two passes. Its online softmax preserves a stable exact rule while its tiled schedule refuses to pay for full quadratic intermediates in slow memory.

Sources and further reading

- [IEEE 754-2019, *Standard for Floating-Point Arithmetic*](#): binary floating-point formats, exceptional values, rounding, and arithmetic semantics.
- [PyTorch, `torch.finfo`](#): dtype-specific bit counts, spacing, normal range, and finite range used in the executable audit.
- [Kalamkar et al., *A Study of BFLOAT16 for Deep Learning Training*](#): BF16’s range-resolution design and mixed-precision training study.
- [NVIDIA, *Accelerating AI Training with NVIDIA TF32 Tensor Cores*](#): TF32 as a compute path over FP32 operands rather than a storage dtype; the product-specific speed claims are not reused here.
- [Micikevicius et al., *FP8 Formats for Deep Learning*](#): the E4M3/E5M2 range-resolution trade and the need for wider roles around eight-bit inputs.
- [Micikevicius et al., *Mixed Precision Training*](#): the FP16 master-weight, wider-accumulation, and loss-scaling recipe.
- [PyTorch, *Automatic Mixed Precision*](#) and [AMP examples](#): current operation-specific autocast and gradient-scaling contracts.
- [PyTorch, *Numerical accuracy*](#): finite precision, extremal values, reduced-precision reductions, and backend-dependent behavior.
- [Williams, Waterman, and Patterson, *Roofline: An Insightful Visual Performance Model for Multicore Architectures*](#): operational intensity, compute and bandwidth roofs, and the ridge point.
- [Milakov and Gimelshein, *Online Normalizer Calculation for Softmax*](#): the online maximum and normalizer recurrence used in the blockwise derivation.
- [Dao et al., *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*](#): tiled exact attention and its I/O analysis.
- [Dao, *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*](#): improved partitioning and parallelism around the same I/O-aware spine.
- [Ainslie et al., *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*](#): grouped query heads with fewer shared key/value heads and the resulting quality-throughput trade-off.
- [PyTorch, `scaled\_dot\_product\_attention`](#): the public attention interface and backend-selection qualifications.

Exercises

1. **(Pencil.)** (a) For FP16, BF16, and FP32, predict how the representable gap above 2^{-10} , 1, and 2^{10} changes with the exponent. **(Code.)** (b) Use `torch.nextafter` to verify the prediction and identify one update that is representable near zero but rounds away when subtracted from each base value.
2. **(Code.)** Compare direct softmax and max-shifted softmax for [10, 11, 12], [1000, 1001, 1002], and [10000, 10001, 10002] in FP16, BF16, FP32, and FP64. Record the **stored logits** before interpreting the probabilities. Separate overflow prevented by stable algebra from distinctions erased by rounding.
3. **(Pencil.)** (a) Predict how FP16 and FP32 accumulators will differ for serial sums of 0.125 and 0.1. **(Code.)** (b) Reproduce the experiment, then implement the 2^{-26} loss-scaling probe with several powers-of-two scales. Explain why loss scaling can preserve a tiny gradient but cannot make a rounded-away FP16 master-weight update effective.

4. **(Pencil.)** Consider a hypothetical machine with a 60-TFLOP/s FP32 roof, a 120-TFLOP/s 16-bit matrix roof, and 1.5-TB/s memory bandwidth. Draw both Rooflines and compute both ridge points. Place an FP32 elementwise transform at 0.25 FLOP/byte, then place both the FP32 and two-byte ideal square GEMMs from Equation C.6 at $n = 256$. State every traffic assumption, then explain why the bounds do not predict a runtime.
5. **(Code.)** Extend the online-attention check to batches, multiple query rows per tile, and a causal mask. Compare it with materialized attention in FP64 and FP32; report maximum absolute error rather than demanding bitwise equality. For $N \in \{512, 2048, 4096\}$ and $d = 64$, compute the logical bytes of Q, K, V, O and two $N \times N$ intermediates for FP32 and a two-byte dtype. Explain which counts are tensor sizes and which would require a profiler to turn into measured traffic.

D. Notation

Notation should shorten an argument, not turn it into a guessing game. This book therefore treats a symbol’s declared shape and local definition as its contract. The typeface gives a useful first clue—the dimensions settle the question.

This appendix is a decoding key for Chapters 1–20—not a replacement for definitions at first use. Appendix A develops the linear algebra behind the symbols (Appendix A), while Appendix B develops their tensor and PyTorch contracts (Appendix B).

D.1. Typeface tells you what kind of object to expect

The book uses the following visual conventions.

Object	Typical notation	How to read it
Scalar	x, x_i, b, η	One number; italic unless it names a standard operator
Vector	$\mathbf{x}, \mathbf{w}, \boldsymbol{\mu}$	A one-axis array, normally a column vector in mathematical expressions
Matrix or batched array	$\mathbf{X}, \mathbf{W}, \boldsymbol{\Sigma}$	A two-axis linear map or a named multi-axis array; its declared shape identifies which
Function	f, g_θ, p_θ	A map, model, probability mass function, or density, as defined locally
Set or collection	$\mathcal{B}, \mathcal{D}, \mathbb{R}^d$	A minibatch, dataset, or ambient space
Objective	$\mathcal{L}, \ell_i$	An aggregate loss or one local/per-example loss term

Lowercase bold symbols usually denote vectors, and uppercase bold symbols usually denote matrices or batched arrays. Both are bold—the source helpers `\vect{}` and `\matr{}` intentionally share one visual style. Boldface alone does not distinguish a matrix from a rank-four image tensor—the shape does that work.

Plain uppercase letters can still be scalars. For example, B is a batch size, T is a sequence length, and V is a vocabulary size—all three are single numbers. By contrast, a bold uppercase symbol such as $\mathbf{V}$ is an array. The identity matrix and zero vector carry their size when ambiguity would matter: $\mathbf{I}_d \in \mathbb{R}^{d \times d}$ and $\mathbf{0} \in \mathbb{R}^d$.

i One sample on paper, rows of samples in code

A mathematical example is normally a column vector $\mathbf{x} \in \mathbb{R}^{D_{\text{in}}}$, so a dense layer is $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$. PyTorch stores a batch as rows: $\mathbf{X} \in \mathbb{R}^{B \times D_{\text{in}}}$. With $\mathbf{W} \in \mathbb{R}^{D_{\text{out}} \times D_{\text{in}}}$, the same layer becomes $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top + \mathbf{1}\mathbf{b}^\top$. The transpose is a consequence of the row-batch convention—it is not a different model.

D.2. Decorations carry a second message

Once the base object is clear, decorations tell us what happened to it.

Decoration	Example	Meaning in this book
Subscript	x_i, h_t, W_ℓ	A coordinate, example, time step, or layer; the surrounding text names the axis
Hat	$\hat{y}, \hat{\mathbf{x}}$	A prediction, estimate, or reconstruction
Star	$\mathbf{w}^*, D^*$	An optimizer or theoretically optimal object
Tilde	$\tilde{\mathbf{x}}$	A perturbed, sampled, or alternative version of an object
Bar	$\bar{\alpha}_t$	An average or accumulated quantity; in diffusion it is a cumulative product
Transpose	$\mathbf{W}^\top$	Rows and columns exchange roles
Inverse	$\mathbf{A}^{-1}$	The inverse map, when it exists; it does not prescribe how to compute a solution
Parenthesized superscript	$\mathbf{w}^{(t)}$	An optimizer iterate or other numbered stage, not exponentiation

The operators follow standard roles:

- $\|\mathbf{x}\|_2$ is the Euclidean norm, while a subscript such as F names a different norm—for example, the Frobenius norm of a matrix.
- $\mathbb{E}[Z]$, $\text{Var}(Z)$, and $\text{Cov}(X, Y)$ are expectation, variance, and covariance.
- $\arg \min_x f(x)$ and $\arg \max_x f(x)$ return an optimizing argument; $\min_x f(x)$ and $\max_x f(x)$ return the attained value.
- $\nabla_{\mathbf{w}} \mathcal{L}$ is a gradient with respect to a vector or parameter collection. $\partial \mathcal{L} / \partial w_j$ selects one scalar derivative.
- dx is a differential inside an integral, and $\mathbb{R}$ denotes the real numbers.

Equality, definition, and update are also distinct. The sign $=$ asserts equality; $:=$ introduces a definition; $\approx$ marks an approximation; and $\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{g}$ describes an algorithmic update—an instruction rather

than an equality. In prose, an arrow may instead summarize a deduction or a flow, so read its immediate context.

D.3. Indices and dimensions: a compact dictionary

An index answers “which entry?” A dimension answers “how many entries?” We define both near first use—the same letter can legitimately change roles across topics.

Symbol	Usual role
i, j	Example, coordinate, class, query position, or key position
b	Batch item; bold $\mathbf{b}$ instead denotes a bias vector
ℓ	Layer or feature coordinate
t	Optimizer iteration, sequence position, or diffusion step
h	Attention-head index; bold $\mathbf{h}_t$ is a hidden state
c	Channel index or class label, according to the section
a, b	Spatial offsets inside a convolution sum

Dimension	Usual role
n, N	Number of examples or another locally counted collection
d, D	Feature dimension or model width
B	Minibatch size
$D_{\text{in}}, D_{\text{out}}$	Input and output widths of a dense map
C, H, W	Channels, image height, and image width
K_h, K_w	Kernel height and kernel width
T, S	Query/target length and source/key length when both are needed
N_h, D_h	Number of attention heads and width per head
V	Vocabulary size; K often counts classes when declared that way
k, r	Bottleneck dimension, retained rank, or low-rank update size

Equations use one-based sums unless stated otherwise, for example $\sum_{i=1}^n$. Python and PyTorch index from zero—translating an equation into code changes labels, not the number of terms.

Local definitions outrank the dictionary

There is no global semantic difference between d and D , or between n and N . Likewise, t can mean an optimization step in Chapter 4 and a diffusion step in Chapter 19. The dictionary records recurring habits—it does not license skipping a section’s definitions.

D.4. The book’s recurring shape contracts

These shapes are the fastest way to audit a derivation or a code cell. Code comments write them with parentheses; equations write the same contract as a product of dimensions—the axis order is identical.

Object	Mathematical shape	PyTorch axis order
Dense batch X	$B \times D$	(B, D)
Dense weight W	$D_{\text{out}} \times D_{\text{in}}$	(D_out, D_in)
Image batch	$B \times C \times H \times W$	(B, C, H, W)
Convolution weight	$C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w$	(C_out, C_in, K_h, K_w)
Sequence states	$B \times T \times D$	(B, T, D)
Multi-head states	$B \times N_h \times T \times D_h$	(B, N_h, T, D_h)
Attention scores	$B \times N_h \times T \times S$	(B, N_h, T, S)
Token logits	$B \times T \times V$	(B, T, V)

In attention, **Q** carries T query positions and **K**, **V** carry S source positions. The score product contracts only the shared head width D_h , leaving (B, N_h, T, S) . Notice the typography: plain V is vocabulary size, while bold **V** is the value array. Plain H is image height, while a bold **H** can be a hidden-state array.

An external library may choose another order—time-first sequences or NHWC images, for example. Convert once at the boundary, state the permutation, and keep the internal contract stable. Table B.1 gives Appendix B’s fuller shape dictionary; the rest of that appendix supplies the broadcasting, stride, masking, and contraction audit.

D.5. Probability and optimization symbols are role labels

Probability sections commonly use uppercase X, Y for random variables and lowercase x, y for realized values, but the distinction is not imposed where boldface and shape already make the object clear. A conditional such as $p(y | x)$ reads “the distribution of y given x ,” and $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is a Gaussian with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$.

Subscripts on a distribution identify its parameters or role. Thus p_θ is a model parameterized by θ . The symbol q often denotes an auxiliary, variational, or forward process—its precise role is still local. A sampling statement such as $\mathbf{z} \sim p(\mathbf{z})$ means that $\mathbf{z}$ is drawn from that distribution.

Parameter letters are similarly local. The collections θ and ϕ can name two networks or two sides of one objective. Several recurring scalar knobs are deliberately overloaded, so their local declaration remains part of the contract:

Symbol	Recurring roles in this book
η, α	Learning rate or another update scale; α can also be an attention weight when locally indexed
β	Momentum coefficient, diffusion schedule, or β -VAE objective weight
λ	Regularization strength or another objective weight

Symbol	Recurring roles in this book
τ	Temperature: kernel bandwidth through $\tau = 2h^2$ in Chapter 12, attention sharpness, or contrastive-logit scale with $\tau = e^{-\gamma}$ in Chapter 20

The letter is a role label, not a global definition. Each chapter declares the role before using the symbol in a calculation.

For losses, ℓ_i usually names a term associated with example i , while $\mathcal{L}$ names the aggregate objective. A reported metric is not automatically the training loss: accuracy, reconstruction error, perplexity, and distributional diagnostics answer different questions even when they are computed from the same predictions.

D.6. A quick notation audit

When an expression feels opaque, ask four questions in order:

1. What does each symbol represent here?
2. What are its axes and dimensions?
3. Which indices are free, and which are summed or otherwise reduced?
4. Does the claimed output retain exactly the free axes?

That audit catches more than illegal products. It also catches legal calculations that contract the wrong axis—the notation equivalent of code that runs while solving the wrong problem.

For contributors, the canonical source helpers are `\vect`, `\matr`, `\E`, `\Ex`, `\var`, `\cov`, `\norm`, `\argmin`, `\argmax`, `\imp`, `\R`, `\dd`, and `\loss`. The helpers `\Ex` and `\imp` remain available for compatibility; current chapters prefer $\mathbb{E}$ for expectation and choose the arrow appropriate to the local statement. Every helper is defined for both the PDF and HTML renderers.

E. Statistical Learning Contracts

A training loss is a number, but the claim we want to make is usually about a population, a distribution, or a future use. Those are not the same object. This appendix gathers the statistical contracts used throughout the book so we can retrieve them without interrupting a chapter’s mechanism.

It is a reference, not a prerequisite. Chapters 1, 4, 6, 18, and 19 state the pieces they need where they need them. Come here when you want to audit the whole chain:

sample $\rightarrow$ estimator $\rightarrow$ population $\rightarrow$ claim.

E.1. One dataset, two risks

Let the observed dataset be $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, drawn from a distribution P . For a model f_θ and per-example loss ℓ , training commonly minimizes the **empirical risk**

$$\widehat{R}_{\mathcal{D}}(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f_\theta(\mathbf{x}_i), y_i). \quad (\text{E.1})$$

The corresponding **population risk** is

$$R_P(\theta) = \mathbb{E}_{(X,Y) \sim P} [\ell(f_\theta(X), Y)]. \quad (\text{E.2})$$

The first is computed from the sample in hand. The second is an expectation over the declared population. Even if Equation E.1 is an unbiased estimate of Equation E.2 for one fixed θ , the fitted $\widehat{\theta}_{\mathcal{D}}$ was selected using the same data. Its training risk is therefore not an independent estimate of its population risk.

This gives the data splits different jobs:

Split	Permitted use	Quantity it can estimate
Training	Fit parameters	The empirical objective used by the optimizer
Validation	Select models, stopping rules, or hyperparameters	Risk under its sampling distribution for the selection procedure being compared
Test or sealed endpoint	Evaluate after analysis choices are fixed	Risk under its declared sampling distribution for the selected procedure

An independent validation mean estimates $R_P(\widehat{\theta}_{\mathcal{D}})$ conditional on the fitted model when the validation examples are drawn from P . Once we repeatedly inspect that validation set and adapt choices to it, it

becomes part of the selection procedure. The estimate can become optimistic. “Never used for gradient updates” is not the same as “never used to make a decision.”

⚠ Trap: independent data is not automatically the right data

A perfectly independent test set drawn from P estimates performance under P . It does not estimate deployment risk under a different distribution Q . Independence protects against reuse; it does not repair a mismatched population.

E.2. Name the distribution that carries the claim

If deployment follows Q rather than P , then the target quantity is

$$R_Q(\theta) = \mathbb{E}_{(X,Y) \sim Q} [\ell(f_\theta(X), Y)] . \quad (\text{E.3})$$

Chapter 6 uses a controlled transformation to make this distinction visible. For a label-preserving input map T , the shifted risk is

$$R_{T_\# P}(\theta) = \mathbb{E}_{(X,Y) \sim P} [\ell(f_\theta(T(X)), Y)] , \quad (\text{E.4})$$

where $T_\# P$ denotes the distribution produced by applying T to inputs drawn from P . Keeping Y unchanged is part of the assumption. A horizontal shift may preserve a garment label; the same statement would be false for a transformation that changes the semantic target.

When T is itself drawn from a declared augmentation distribution $\mathcal{A}$, the population objective becomes

$$R_{\mathcal{A}}(\theta) = \mathbb{E}_{(X,Y) \sim P} \mathbb{E}_{T \sim \mathcal{A}} [\ell(f_\theta(T(X)), Y)] . \quad (\text{E.5})$$

Sampling one fresh transformation per example is a Monte Carlo estimator of the inner expectation. It optimizes average behavior under $\mathcal{A}$; it does not prove exact invariance to every transformation, including transformations outside the declared support.

The reusable audit is short: name the training distribution, the validation and test distributions, the deployment distribution, and every transformation relating them. If two of those happen to be equal, say so rather than leaving the equality implicit.

E.3. A loss is a conditional probability contract

Suppose a model specifies a conditional distribution $p_\theta(y \mid \mathbf{x})$. Under conditionally independent observations, its likelihood and mean negative log-likelihood are

$$\mathcal{L}(\theta) = \prod_{i=1}^n p_\theta(y_i \mid \mathbf{x}_i), \quad \text{NLL}(\theta) = -\frac{1}{n} \sum_{i=1}^n \log p_\theta(y_i \mid \mathbf{x}_i). \quad (\text{E.6})$$

Maximizing the likelihood is equivalent to minimizing NLL. The target space and conditional distribution determine the familiar loss:

Target contract	Model parameterization	NLL, up to constants or scale	Familiar loss
$Y \mid X \sim \mathcal{N}(\mu_\theta(X), \sigma^2)$ with fixed σ	$\mu_\theta \in \mathbb{R}$	$(y - \mu_\theta)^2$	Squared error
$Y \mid X \sim \text{Bernoulli}(p_\theta(X))$	$p_\theta = \text{sigmoid}(o_\theta)$	$-y \log p_\theta - (1 - y) \log(1 - p_\theta)$	Binary cross-entropy
$Y \mid X \sim \text{Categorical}(\mathbf{p}_\theta(X))$	$\mathbf{p}_\theta = \text{softmax}(\mathbf{o}_\theta)$	$-\log p_{\theta,y}$	Categorical cross-entropy
$Y \mid X \sim \text{Laplace}(\mu_\theta(X), b)$ with fixed b	$\mu_\theta \in \mathbb{R}$	$ y - \mu_\theta $	Absolute error

This table is not a command to force every problem into four rows. It is a way to ask what kind of target is being modeled, which parameters must stay in a valid domain, and what deviations the loss treats as plausible. Counts, angles, intervals, censored values, and structured outputs may require different conditional contracts.

E.3.1. Why the Gaussian is plausible, and where it is not

The Gaussian assumption is mathematically convenient, but convenience is not its only motivation. If a residual is the sum of many small, weakly dependent perturbations with finite variance, a central-limit argument can make a Gaussian approximation plausible. Among distributions on $\mathbb{R}$ with a fixed mean and variance, the Gaussian also has maximum differential entropy: it adds no further shape constraint beyond those two moments.

Neither argument proves that a particular residual is Gaussian. A few dominant causes, heavy tails, changing variance, dependence, truncation, or multimodality can break the approximation. The honest sequence is:

1. state the conditional residual assumption;
2. derive the loss it implies;
3. inspect residuals and relevant slices;
4. revise the likelihood or estimator when the mismatch changes the claim.

The maximum-entropy statement is also conditional on knowing the mean and variance and using support $\mathbb{R}$. Change the support or known constraints and the maximum-entropy distribution changes.

E.4. Three ways to compare distributions

Chapters 18 and 19 compare distributions for different reasons. The symbol “distance” can hide very different geometry, so start with the question rather than the formula.

For densities or mass functions p and q , the Kullback–Leibler divergence is

$$D_{\text{KL}}(P \parallel Q) = \mathbb{E}_{X \sim P} \left[\log \frac{p(X)}{q(X)} \right]. \quad (\text{E.7})$$

It is nonnegative and equals zero only when the distributions agree almost everywhere under the usual conditions. It is not symmetric, does not satisfy the triangle inequality, and becomes infinite when P puts positive mass where Q puts none. Its direction matters: $D_{\text{KL}}(P\|Q)$ averages a log density ratio under P .

Let $M = \frac{1}{2}(P + Q)$. The Jensen–Shannon divergence is

$$D_{\text{JS}}(P\|Q) = \frac{1}{2}D_{\text{KL}}(P\|M) + \frac{1}{2}D_{\text{KL}}(Q\|M). \quad (\text{E.8})$$

It is symmetric and lies between 0 and $\log 2$ when natural logarithms and equal mixture weights are used. The mixture prevents the support failure that can make a KL term between P and Q infinite. JSD still compares probability mass, not the physical distance between locations.

Given a ground metric d on the sample space, the Wasserstein-1 distance is

$$W_1(P, Q) = \inf_{\gamma \in \Pi(P, Q)} \mathbb{E}_{(X, Y) \sim \gamma} [d(X, Y)], \quad (\text{E.9})$$

where $\Pi(P, Q)$ is the set of couplings with marginals P and Q . In one dimension,

$$W_1(P, Q) = \int_0^1 |F_P^{-1}(u) - F_Q^{-1}(u)| \, du. \quad (\text{E.10})$$

Wasserstein distance uses the geometry of the sample space and inherits the units of the ground metric. That can be valuable, but it also means the answer depends on whether the chosen ground metric represents meaningful movement.

Quantity	What it makes visible	Boundary to remember
$D_{\text{KL}}(P\ Q)$	Expected log-density mismatch in a declared direction	Asymmetric; support mismatch can make it infinite
$D_{\text{JS}}(P\ Q)$	Symmetric mass disagreement with a shared mixture	Does not know that two nearby points are near
$W_1(P, Q)$	Amount and distance of mass movement	Requires a meaningful ground metric; estimation can be difficult in high dimensions

No row certifies sample quality, fairness, or deployment safety by itself. The metric must be paired with a population, an estimator, and the substantive property the measurement is meant to probe.

E.5. What the code estimates

Monte Carlo turns an expectation into an average. If $Z_1, \dots, Z_B$ are independent draws and $g(Z)$ has finite variance, then

$$\hat{\mu}_B = \frac{1}{B} \sum_{b=1}^B g(Z_b), \quad \mathbb{E}[\hat{\mu}_B] = \mathbb{E}[g(Z)], \quad \text{Var}(\hat{\mu}_B) = \frac{\text{Var}(g(Z))}{B}. \quad (\text{E.11})$$

The $1/B$ variance law assumes independent draws. Correlated samples contain less information than the same number of independent samples, so the sampling mechanism is part of the estimator contract.

Chapter 4 distinguishes three cases that recur across the book:

1. **Decomposable per-example objective.** A batch mean estimates a population mean. Squared error and cross-entropy usually live here.
2. **Nonlinear functional of an aggregate.** If the target is $R(\mathbb{E}[g(Z)])$, the plug-in $R(\hat{\mu}_B)$ is generally biased. Jensen’s inequality gives a direction when R is convex or concave. More optimization steps do not remove fixed-batch plug-in bias; larger or restructured estimators may.
3. **Batch-defined objective.** The batch supplies candidates or negatives and is part of the objective itself. Changing batch size changes the target, not merely its Monte Carlo noise.

The code audit is therefore not “does this line contain `.mean()`?” It is:

Field	Question
Target	Which population quantity or protocol-defined objective do we want?
Estimator	Which random quantity does the code compute?
Reduction	Which axes and examples enter the sum, mean, or denominator?
Validity	Under what sampling and independence assumptions does it estimate the target?
Boundary	What bias, dependence, missing support, or protocol change remains?

E.6. Standard deviation is not standard error

Uncertainty must name what was repeated. Suppose a complete training protocol is run with K independently chosen seeds and produces metrics $M_1, \dots, M_K$.

- The **seed standard deviation** describes spread among those K optimization runs under the fixed dataset and protocol.
- The **standard error of the seed mean**, often estimated as $s/\sqrt{K}$, describes uncertainty in that mean under an idealized population of comparable independent runs. With very small K , both s and the resulting interval are unstable.
- Neither quantity automatically measures variation across new datasets, deployment populations, annotator groups, prompts, or future time periods.

When two methods use the same data split, initialization family, and minibatch schedule, analyze paired differences $D_k = M_{A,k} - M_{B,k}$. Pairing can remove shared variation and answers the comparison more directly than two unrelated marginal SDs. If several observations come from one prompt, person, subject, or source document, that cluster may be the sampling unit; treating its rows as independent understates uncertainty.

i A compact statistical audit

Before interpreting a reported number, retrieve five nouns:

1. **Population:** which P , Q , slice, or transformed distribution?
2. **Estimand:** risk, accuracy, divergence, calibration error, or another quantity?
3. **Estimator:** what exact calculation and denominator produced the number?
4. **Variation:** data, optimization, sampling, annotation, or deployment variation?
5. **Boundary:** which tempting conclusion does this evidence not establish?

Sources and further reading

- Fisher, *On the Mathematical Foundations of Theoretical Statistics*: likelihood.
- Geman, Bienenstock, and Doursat, *Neural Networks and the Bias/Variance Dilemma*: classical bias and variance.
- Kullback and Leibler, *On Information and Sufficiency*: KL divergence.
- Lin, *Divergence Measures Based on the Shannon Entropy*: Jensen–Shannon divergence.
- Jaynes, *Information Theory and Statistical Mechanics*: maximum entropy under declared constraints.
- Metropolis and Ulam, *The Monte Carlo Method*: Monte Carlo estimation.
- Villani, *Optimal Transport: Old and New*: transport geometry.
- Prince, *Understanding Deep Learning*: an external completeness check for probability and loss coverage.

Exercises

1. **(Pencil.)** At a fixed x , write $\hat{f}_{\mathcal{D}}(x) - Y$ as the sum of the mean-prediction error, the training-set fluctuation, and the fresh-outcome noise. Expand the square. State the zero-mean and conditional-independence assumptions that remove the cross terms and recover Equation 1.5.
2. **(Pencil.)** Derive the Gaussian and Laplace rows of the likelihood table by taking negative logarithms. Which constants may be dropped when their scale parameters are fixed, and what changes if those scales are learned?
3. **(Code.)** Let $Z \sim \mathcal{N}(0, 1)$ and estimate $\log \mathbb{E}[e^Z]$ with $\log(B^{-1} \sum_b e^{Z_b})$. Sweep B and many repetitions. Measure the plug-in bias, predict its sign from Jensen’s inequality, and distinguish it from the Monte Carlo variance within one B .
4. **(Audit.)** A team trains on P , tunes repeatedly on one validation set from P , and reports accuracy on an independent shifted set $T_{\#}P$ as “test accuracy.” Name the three separate questions hidden by that phrase and propose a split and reporting contract that answers each one.
5. **(Audit.)** Two scalar generators have similar JSD but different Wasserstein-1 distance under the usual Euclidean metric. Give one configuration where this is plausible. Explain which conclusion would change if the coordinate units were rescaled, and why neither number alone establishes perceptual fidelity.
6. **(Code.)** Compare two methods across five matched seeds. Report the five paired differences, their mean and SD, and the two unpaired marginal SDs. Then repeat after breaking the pairing. Explain which source of variation the paired summary removes and which sources remain outside the experiment.