CS231n 2025 Lecture Notes

Deep Learning for Computer Vision — Stanford, Spring 2025

Murat Raimbekov

Table of contents

1. CS231n 2025 Lecture Notes **1**
- 1.1. Lectures . . . 1
- 1.2. What these notes are . . . 2
- 1.3. Roadmap . . . 2
- 1.4. Contributing . . . 2
- 1.5. Disclaimer . . . 2

I. Fundamentals **3**

2. Introduction **5**
- 2.1. Vision as the hard part of intelligence . . . 5
- 2.2. Eyes before brains . . . 6
- 2.3. What neuroscience handed over . . . 7
- 2.4. Thirty years of geometry . . . 8
- 2.5. Why vision is ill-posed . . . 10
- 2.6. What the human system tells us about the target . . . 11
- 2.7. Features, benchmarks, and the turn to learning . . . 12
- 2.8. The neural network thread . . . 12
- 2.9. 2012 . . . 13
- 2.10. What followed, and what did not . . . 15
- 2.11. The shape of this course . . . 16
- References . . . 19

3. Image Classification with Linear Classifiers **21**
- 3.1. The semantic gap . . . 21
- 3.2. Nearest neighbors: let the dataset do the work . . . 22
- 3.3. Validation is part of the method . . . 23
- 3.4. A parametric model: one matrix, one score per class . . . 23
- 3.5. The loss turns “good” into a number . . . 24
- 3.6. Softmax: scores as a probability distribution . . . 24
- 3.7. Multiclass SVM loss: enforce a margin . . . 25
- 3.8. Softmax and SVM are choices above the same model . . . 26
- References . . . 26

4. Regularization and Optimization **27**
- 4.1. Two questions the loss leaves open . . . 27
- 4.2. Regularization . . . 27
- 4.3. What L_1 and L_2 each prefer . . . 29
- 4.4. Optimization as search . . . 30
- 4.5. Following the slope . . . 30
- 4.6. Gradient descent, and why it becomes stochastic . . . 31
- 4.7. Three ways SGD struggles . . . 32
- 4.8. Momentum . . . 33

4.9. Per-parameter learning rates . . . 34
4.10. Why AdamW is not Adam plus L_2 . . . 35
4.11. Learning rate schedules . . . 36
4.12. Second-order methods . . . 38
4.13. Where the linear model stops . . . 39
References . . . 39

5. Neural Networks and Backpropagation **41**
5.1. What the hidden layer buys . . . 41
5.2. What the max is doing . . . 42
5.3. Choosing an activation . . . 43
5.4. Depth, width and the forward pass . . . 46
5.5. Size is not a regularizer . . . 47
5.6. The brain analogy and its limits . . . 47
5.7. Why gradients need a system . . . 48
5.8. The chain rule on a graph . . . 48
5.9. A longer example, and a choice about granularity . . . 50
5.10. Four patterns worth recognising . . . 51
5.11. Forward and backward as an interface . . . 51
5.12. Vectors, and the Jacobian you do not build . . . 52
5.13. Matrices, and the same trick at scale . . . 53
5.14. What this makes possible . . . 54
References . . . 54

6. Convolutional Networks **57**
6.1. Two places to put the design . . . 57
6.2. What flattening throws away . . . 58
6.3. A bit of history . . . 59
6.4. The convolution layer . . . 60
6.5. Stacking convolutions . . . 61
6.6. What the filters learn . . . 62
6.7. Padding, stride, and the shape arithmetic . . . 64
6.8. Receptive fields . . . 64
6.9. Counting a layer . . . 65
6.10. Pooling . . . 67
6.11. Equivariance . . . 68
6.12. What this leaves . . . 69
References . . . 70

7. CNN Architectures **71**
7.1. The rest of the kit . . . 71
7.2. Normalization layers . . . 71
7.3. Dropout . . . 73
7.4. Activations, settled . . . 74
7.5. The revolution of depth . . . 75
7.6. VGG: design by rule . . . 76
7.7. Where depth stopped working . . . 78
7.8. Residual networks . . . 79
7.9. Initialization . . . 80
7.10. Preprocessing . . . 81
7.11. Noise you add and then average away . . . 82
7.12. Transfer learning . . . 83

7.13. Choosing hyperparameters . . . 84
7.14. What this leaves . . . 85
References . . . 85

8. Recurrent Neural Networks **87**
8.1. Problems with a shape . . . 87
8.2. A state that persists . . . 88
8.3. A network built by hand . . . 89
8.4. Unrolling . . . 90
8.5. Truncated backpropagation . . . 92
8.6. One character at a time . . . 93
8.7. What the state ends up holding . . . 94
8.8. One network's output as another's input . . . 95
8.9. Depth in the other direction . . . 96
8.10. Why the gradient dies . . . 97
8.11. LSTM . . . 98
8.12. What recurrence costs . . . 100
8.13. What this leaves . . . 101
References . . . 101

9. Attention and Transformers **103**
9.1. One vector too few . . . 103
9.2. Looking back . . . 104
9.3. What the weights show . . . 106
9.4. Attention as a layer in its own right . . . 107
9.5. Keys and values . . . 108
9.6. One set, attending to itself . . . 109
9.6.1. The layer does not know what order the inputs came in . . . 110
9.6.2. Blocking some of the connections . . . 111
9.7. Several heads . . . 111
9.8. What it costs . . . 112
9.9. Three ways to process a sequence . . . 113
9.10. The transformer block . . . 114
9.11. Stacking blocks into a language model . . . 114
9.12. The same block, on images . . . 116
9.13. What changed after 2017 . . . 116
9.14. What this leaves . . . 117
References . . . 118

II. Vision Tasks **119**

10. Object Detection and Segmentation **121**
10.1. Past a single label . . . 121
10.2. Labelling every pixel . . . 122
10.3. Putting the resolution back . . . 123
10.4. The shape that keeps coming back . . . 125
10.5. Detection is not a regression problem . . . 126
10.6. Proposing regions . . . 127
10.7. Learning the proposals . . . 128
10.8. One pass . . . 130
10.9. Detection as set prediction . . . 131

10.10 Instance segmentation for almost free . . . 132
10.11 What has the network learned . . . 133
10.12 Which pixels mattered . . . 133
10.13 Where the evidence is . . . 134
10.14 What this leaves . . . 135
References . . . 136

11. Video Understanding **139**
11.1. The axis that was missing . . . 139
11.1.1. Videos are big . . . 139
11.1.2. Which frames . . . 140
11.2. The baseline that refuses to model time . . . 140
11.3. Where to fuse . . . 141
11.3.1. Late fusion . . . 141
11.3.2. Early fusion . . . 141
11.4. Slow fusion: the 3D convolution . . . 142
11.4.1. The three architectures side by side . . . 142
11.4.2. Why not just stack the frames . . . 143
11.4.3. What the filters learn . . . 144
11.5. What the comparison actually showed . . . 144
11.5.1. C3D . . . 145
11.6. Motion as its own signal . . . 146
11.6.1. Optical flow . . . 146
11.6.2. Two-stream networks . . . 147
11.7. Reaching past the clip . . . 148
11.7.1. The two ways of covering time . . . 148
11.7.2. Recurrent convolutional networks . . . 148
11.8. Attention over space and time . . . 150
11.9. Recycling the image networks . . . 151
11.9.1. Transformers, and where the numbers went . . . 152
11.10 What a video model sees . . . 153
11.11 Past the clip label . . . 153
11.12 The modality that was left out . . . 154
11.13 Where the field is going . . . 154
11.13.1 Efficiency . . . 155
11.13.2 Egocentric video . . . 155
11.13.3 Video and language models . . . 155
11.14 What this leaves . . . 155

III. Systems **157**

12. Large-Scale Distributed Training **159**
12.1. Twenty-five thousand computers, one model . . . 159
12.2. Inside one GPU . . . 160
12.2.1. Where the throughput actually comes from . . . 160
12.2.2. A thousandfold, with an asterisk . . . 161
12.3. From a chip to a cluster . . . 162
12.4. Four axes and a budget . . . 163
12.5. Data parallelism . . . 164
12.5.1. Hiding the all-reduce . . . 164
12.5.2. The ceiling . . . 165

12.6. Fully sharded data parallelism . . . 166
12.7. Hybrid sharding . . . 167
12.7.1. A recipe, and where it stops . . . 168
12.8. Activations, and paying compute to store fewer of them . . . 168
12.9. The metric that settles the arguments . . . 169
12.10Tensor parallelism . . . 171
12.11Context parallelism . . . 173
12.12Pipeline parallelism . . . 173
12.13Four dimensions at once . . . 174
12.14Closing . . . 175

13. Self-Supervised Learning **177**
13.1. What a representation costs . . . 177
13.2. An invented task, and a real one . . . 178
13.3. Tasks made from transformations . . . 180
13.4. Tasks made from reconstruction . . . 182
13.5. Tracking, from nobody's tracking labels . . . 183
13.6. Masked autoencoders . . . 185
13.7. What all the pretext tasks were approximating . . . 187
13.8. The contrastive objective . . . 188
13.9. SimCLR . . . 189
13.10MoCo . . . 190
13.11Contrasting sequences instead of instances . . . 191
13.12DINO, and doing without negatives . . . 192
13.13Where this leaves things . . . 194

IV. Generative Models **195**

14. Generative Models I **197**
14.1. Two axes, not one . . . 197
14.2. Probability mass is a fixed budget . . . 198
14.3. Bayes' rule connects all three . . . 200
14.4. What a density is actually for . . . 200
14.5. A taxonomy by what you can compute . . . 201
14.6. Maximum likelihood, and the direction of the divergence . . . 202
14.7. The chain rule needs no assumptions . . . 204
14.8. Images do not come as sequences . . . 205
14.9. Reconstruction without a distribution . . . 207
14.10One latent variable, two intractable integrals . . . 209
14.11A second network for the posterior . . . 209
14.12The evidence lower bound . . . 211
14.13Getting a gradient through a sample . . . 212
14.14Two terms that want opposite things . . . 213
14.15What each family bought . . . 215

15. Generative Models II **217**
15.1. A model defined only by its sampler . . . 217
15.2. The discriminator is the loss function . . . 218
15.3. The gradient the obvious objective does not deliver . . . 219
15.4. What the game would converge to . . . 220
15.5. What GANs turned out to give, and what they cost . . . 221

15.6. Trading one hard step for many easy ones . . . 222
15.7. Straight lines between two distributions . . . 223
15.8. Sampling is an ODE solve . . . 225
15.9. What the network can actually predict . . . 225
15.10 Conditioning, and buying obedience with a second forward pass . . . 228
15.11 Diffusing somewhere cheaper . . . 228
15.12 The backbone, and where the conditioning attaches . . . 231
15.13 Video, and the bill for many steps . . . 231
15.14 One object, several languages . . . 232
15.15 Autoregression comes back . . . 234
15.16 What each family bought . . . 235

V. Beyond 2D **237**

16. 3D Vision **239**
16.1. The question images never ask . . . 239
16.2. Points, and what they cannot tell you . . . 240
16.3. Surfaces as maps . . . 241
16.4. The asymmetry that organises everything . . . 242
16.5. Building shapes out of functions . . . 244
16.6. What there was to learn from . . . 246
16.7. Sidestepping the problem: render it first . . . 247
16.8. Voxels, and the exponent they pay . . . 248
16.9. Learning on sets: PointNet . . . 250
16.9.1. Comparing two sets . . . 250
16.10 Predicting a surface instead of a cloud . . . 252
16.11 The network is the field . . . 253
16.12 Geometry and appearance in one function . . . 254
16.13 Back to explicit, for speed . . . 256
16.14 What geometry leaves out . . . 257
16.15 The map is the result . . . 258

VI. Multimodal **261**

17. Vision and Language **263**
17.1. One model, or one model per task . . . 263
17.2. Text as the label that already exists . . . 264
17.3. A similarity model is already a classifier . . . 266
17.4. What the zero-shot number was hiding . . . 267
17.5. The failure the objective permits . . . 269
17.6. Fusion by concatenation . . . 270
17.7. Fusion by cross-attention . . . 271
17.8. Open weights are not open data . . . 273
17.9. Promptability as the pretraining task . . . 274
17.10 Chaining, and the case for not training . . . 276
17.11 What the chapter adds up to . . . 277

VII. Embodied AI **279**

18. Robot Learning **281**
18.1. Closing the loop . . . 281
18.2. What makes robot perception a different problem . . . 282
18.3. Four things the reward has to pay for . . . 284
18.4. Where scale worked, and what it cost . . . 286
18.5. Learning the model instead of the policy . . . 288
18.6. Imitation, and two ways it goes wrong . . . 290
18.6.1. The distribution the policy makes for itself . . . 292
18.6.2. Averaging two good answers . . . 293
18.7. Robotic foundation models . . . 294
18.8. What is not solved . . . 297

VIIIImpact **299**

19. Human-Centered AI **301**
19.1. What the field was trying to match . . . 301
19.2. Seeing what no person could . . . 301
19.3. Where human vision fails . . . 302
19.4. What "unfair" would have to mean . . . 303
19.5. Designing for what must not be seen . . . 308
19.6. The clinic and the base rate . . . 309
19.7. Letting people choose the task . . . 312
19.8. Augmentation . . . 312

1. CS231n 2025 Lecture Notes

Deep Learning for Computer Vision — Stanford University, Spring 2025

> **How to read these notes**
>
> Each lecture is a standalone page written to be read straight through rather than skimmed. Sections are numbered and individually linkable, so you can share a link to exactly the part you mean, and equations and figures are cross-referenced so a later lecture can point back at the thing it depends on.

1.1. Lectures

The course runs to 18 lectures, and all of them are written. The roadmap groups them by theme.

Table 1.1.: Lectures published so far.

#	Topic	Status
1	Introduction	Complete
2	Image Classification with Linear Classifiers	Complete
3	Regularization and Optimization	Complete
4	Neural Networks and Backpropagation	Complete
5	Convolutional Networks	Complete
6	CNN Architectures	Complete
7	Recurrent Neural Networks	Complete
8	Attention and Transformers	Complete
9	Object Detection and Segmentation	Complete
10	Video Understanding	Complete
11	Large-Scale Distributed Training	Complete
12	Self-Supervised Learning	Complete
13	Generative Models I	Complete
14	Generative Models II	Complete
15	3D Vision	Complete
16	Vision and Language	Complete
17	Robot Learning	Complete
18	Human-Centered AI	Complete

1.2. What these notes are

The official cs231n.github.io notes are excellent — and they stopped being updated in 2017. Everything that has since reshaped computer vision is missing from them.

These notes are written from the Spring 2025 lecture recordings and slides, not summarised from the 2017 notes. That means they follow what the lecturers actually emphasised in 2025, including the parts that changed.

Each page aims to be:

- **Self-contained** — you should not need the video open alongside it.
- **Honest about depth** — core ideas get full derivations; peripheral ones get an intuition and a link to the paper.
- **Linked to sources** — every result points at the original paper, not at a secondary summary.

1.3. Roadmap

Lectures are published as they are written, in course order, following the 2025 syllabus.

1. **Fundamentals** — introduction, image classification, optimisation, backpropagation
2. **Convolutional networks** — CNNs, architectures from AlexNet to ConvNeXt
3. **Sequence models** — RNNs, attention, Transformers, ViT
4. **Vision tasks** — detection, segmentation, video, 3D
5. **Generative models** — VAEs, GANs, diffusion
6. **Advanced topics** — NeRF, vision-language models, robot learning, ethics

1.4. Contributing

Found an error? Corrections are genuinely welcome — there is an **Edit this page** link at the bottom of every lecture, which opens a pull request in your browser. For anything larger, open an issue.

1.5. Disclaimer

This is an **unofficial** resource, not affiliated with Stanford University. For official course materials see cs231n.stanford.edu.

The notes are released under the MIT licence, which covers the writing and the diagrams made for this project. Figures reproduced from research papers and lecture slides remain the property of their original authors and are credited in their captions — see NOTICE.

Part I.

Fundamentals

2. Introduction

Lecture 1

Based on the lecture by Fei-Fei Li and Ehsan Adeli — CS231n, Stanford University, Spring 2025.

2.1. Vision as the hard part of intelligence

Artificial intelligence is usually introduced as a single field with a single goal, and computer vision as one application among many that sit downstream of it. The ordering is convenient and slightly misleading. For most of the history of life, vision is what made intelligence necessary in the first place, and in the history of this particular field it is the problem on which our best learning methods were first shown to work at all.

The course sits at a specific intersection, and it is worth being precise about which one. Machine learning supplies the mathematics: the idea that a system's behaviour can be fitted to data rather than specified by hand. Within machine learning, the last decade or so belongs to deep learning, a family of methods built around neural networks, which we will define properly once we have the machinery to do so. Computer vision supplies the problems — what is in this image, where is it, what is it doing, and what is about to happen. Neither field is covered here in its entirety, and neither could be in ten weeks. What is covered is the overlap, which is also where both fields have moved fastest.

That overlap has turned out to be unusually portable. The convolutional networks, attention mechanisms and generative models developed to solve vision problems now underpin systems that never touch an image, and the techniques travel in the other direction too. A student who leaves this course with a working understanding of how these models are built and trained is not confined to vision, which is the main reason it is worth learning them properly rather than as recipes.

The claim that vision is central rather than peripheral has an anatomical version. The primate visual system is not a module but a substantial fraction of the cortex, and the most-cited mapping of it identifies thirty-two distinct cortical areas that are visual or visual-association in function, connected by several hundred pathways arranged in a rough hierarchy (Felleman and Van Essen, 1991). Whatever else the brain is doing, a large part of it is looking. This is the sense in which understanding visual intelligence is not a subproblem of understanding intelligence: it is most of the territory.

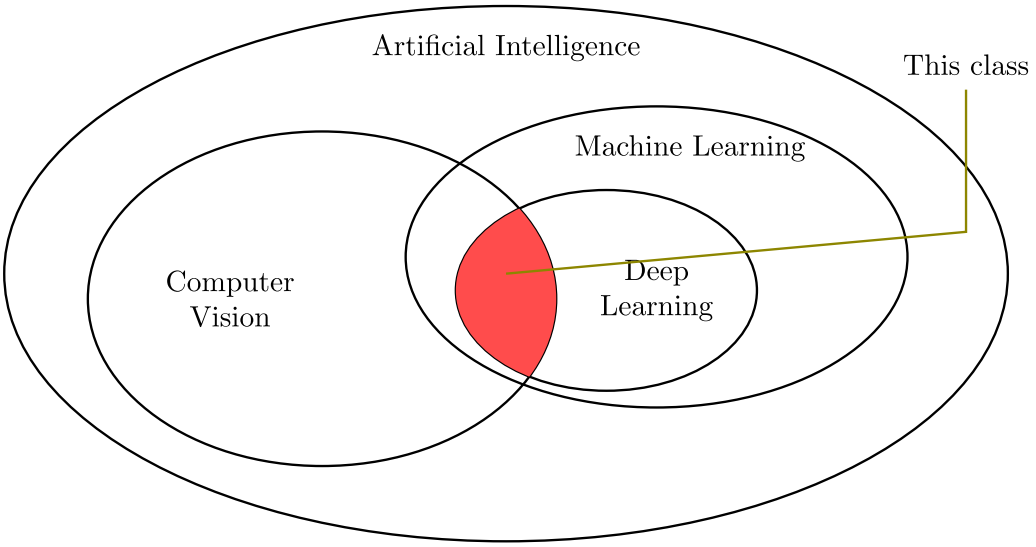


Figure 2.1.: The scope of this course. Machine learning supplies the methods, computer vision supplies the problems, and deep learning is the part of machine learning where the two have most productively met. *Drawn for these notes, after the CS231n 2025 Lecture 1 slides.*

2.2. Eyes before brains

The history of vision is considerably older than the history of anything that could be called a brain. For roughly three billion years, life on Earth was aquatic, sparse and passive. Animals drifted, absorbed what came to them, and did not do much else. Then, over a window of perhaps ten million years — an eyeblink on the evolutionary clock — the fossil record shows an abrupt proliferation of animal body plans that palaeontologists call the Cambrian explosion, dated to around 540 million years ago.

Several explanations have been offered for what triggered it, from ocean chemistry to climate. The one that matters here is the hypothesis that the trigger was the eye. The first light-sensitive structures were not lenses and retinas but something closer to a pinhole: a patch of photosensitive cells that could register the direction light was coming from. That is a modest piece of biological engineering with an immodest consequence. An animal that cannot sense at a distance is confined to metabolism; it takes what arrives and it is taken in turn. An animal that can see becomes a participant in its environment. It can pursue, and it can be pursued.

The evolutionary pressure this creates is the point. Once one animal can see, every other animal is in an arms race — to detect, to hide, to move faster, to predict. Vision does not merely benefit from a nervous system; it is one of the forces that made an elaborate nervous system worth paying for. The five hundred million years that follow are, on this reading, not the evolution of vision alongside intelligence but the evolution of vision as intelligence.

Humans sit at an extreme of that lineage. We are overwhelmingly visual animals, and the elaborate cortical machinery described in Section 2.1 is what that specialisation looks like from the inside.

Building machines that see is a much more recent ambition, though not as recent as one might assume. The optical principle arrived long before the technology: thinkers in ancient China and Greece described the projection of an image through a small aperture, and by the sixteenth century the camera obscura was well enough understood for Leonardo to sketch it. Photography

made the image permanent, and the intervening two centuries have made cameras so cheap and so numerous that most of the images now produced on Earth are never seen by a human being at all.

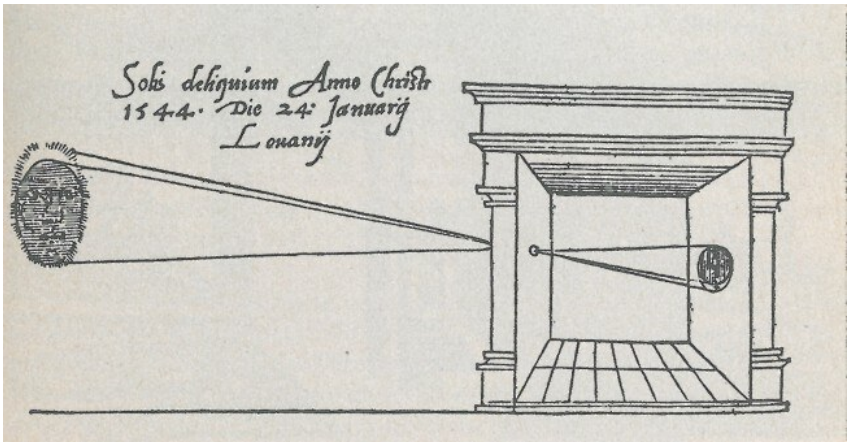


Figure 2.2.: The camera obscura as illustrated by Gemma Frisius in 1545, showing a solar eclipse projected through a small aperture onto the wall of a darkened room. The optical principle is the same one Equation 2.1 describes. *Public domain.*

None of which solves the problem. A camera is an apparatus for capturing light, in the same way that an eye is an apparatus for capturing light, and neither one sees. Evolution needed hundreds of millions of years of nervous system on top of the pinhole before anything was recognised. The gap between capturing an image and understanding it is the entire subject of this course, and the first thirty years of trying to close it are instructive mostly for the ways they failed.

2.3. What neuroscience handed over

The most consequential experiment in the prehistory of computer vision was not a computing experiment. In the late 1950s, David Hubel and Torsten Wiesel set out to characterise how the mammalian visual pathway represents what falls on the retina. They recorded from single neurons in the primary visual cortex of anaesthetised cats while presenting visual stimuli, looking for the pattern that would make a given cell fire (Hubel and Wiesel, 1959).

The result that mattered was partly an accident of apparatus. The stimuli were slides, and the cells were largely uninterested in them; what reliably provoked a response was the edge of the slide sweeping across the projector as it was changed. Neurons in the primary visual cortex were not responding to objects, or to complex patterns, or to anything a person would name. They were responding to oriented bars of light at particular positions, and each one had a preferred orientation it responded to far more strongly than any other.

Two properties of that population turned out to be foundational. The cells Hubel and Wiesel called *simple* respond to an edge at a specific orientation in a specific location, and stop responding when the stimulus is moved or rotated away from that preference. The cells they called *complex* respond to the same oriented structure but tolerate some displacement, firing across a range of positions rather than a single one. Between them, these two populations do something a designer would recognise: they detect a local feature, and then they build a slightly more forgiving version of the same detection that no longer depends on exactly where the feature fell.

The architectural claim implicit in this is that visual representation is hierarchical and that it starts small. The cortex does not attempt to match an incoming image against a stored template of a cat. It extracts oriented edges, combines them into configurations that tolerate small changes in position, and repeats — with each stage responding to something more specific and less tied to the retinal coordinates it came from than the stage below.

Every convolutional network in this course is a restatement of that idea. The convolution operator is a bank of local feature detectors applied at every position; pooling is the tolerance to displacement that distinguishes complex cells from simple ones; depth is the hierarchy. When we come to look at what the early layers of a trained network have learned, they will turn out to be oriented edge detectors, **which nobody put there**. It took thirty years and a great deal of intervening failure for the field to arrive at an architecture that took Hubel and Wiesel seriously, and the failure is worth understanding before the success.

2.4. Thirty years of geometry

Computer vision as an academic field is conventionally dated to the early 1960s, and to a doctoral thesis. Larry Roberts, working at MIT, asked whether a program could take a photograph of a scene built from wooden blocks and recover the three-dimensional shapes that produced it (Roberts, 1963). The pipeline extracted intensity discontinuities, selected feature points, and fitted polyhedra to them. It worked, on blocks, under controlled lighting, which is a considerably narrower claim than it sounds like but was a genuine result at the time.

The ambition of the period is captured better by what happened three years later, when a summer project was set up at MIT with the intention of using a handful of undergraduates over a single vacation to construct a substantial part of a vision system. Vision was not solved that summer. The episode is usually told as a joke about optimism, and it is one, but the more useful reading is that the problem's difficulty was *invisible from the outside*. Nothing about looking at a photograph tells you that recovering its contents is hard, because your own visual system does it without reporting any effort. Sixty years later the annual computer vision conferences draw over ten thousand people, and the summer project has become a field.

The intellectual centre of the following decade was David Marr, whose account of vision proposed that the problem be decomposed into stages of increasingly abstract representation. An input image is first reduced to a *primal sketch* — zero crossings, blobs, bars, edges and their groupings, which is very nearly the vocabulary Hubel and Wiesel had found in the cortex. The primal sketch is then elaborated into a **2½-D sketch**, which assigns local surface orientation and marks discontinuities in depth: enough to say that the ball is in front and the ground is behind, but still tied to the viewer's position. Only at the final stage does the system arrive at a full three-dimensional model, organised hierarchically in terms of surfaces and volumetric primitives and independent of where the observer happens to be standing.

Marr's framework was principled, neurally motivated, and it set the direction for a generation. It also placed the hardest thing in vision at the end of the pipeline as the definition of success. Recovering the complete three-dimensional structure of a scene is the *holy grail* on this account, and everything before it is preparation.

The recognition work of the period inherited that geometric commitment. If objects are three-dimensional shapes, then an object category should be describable as an arrangement of simpler shapes, and recognition becomes the problem of fitting that arrangement to an image. Generalized cylinders, developed at Stanford by Brooks and Binford, modelled objects as assemblies of tubular primitives; pictorial structures, from Fischler and Elschlager, modelled

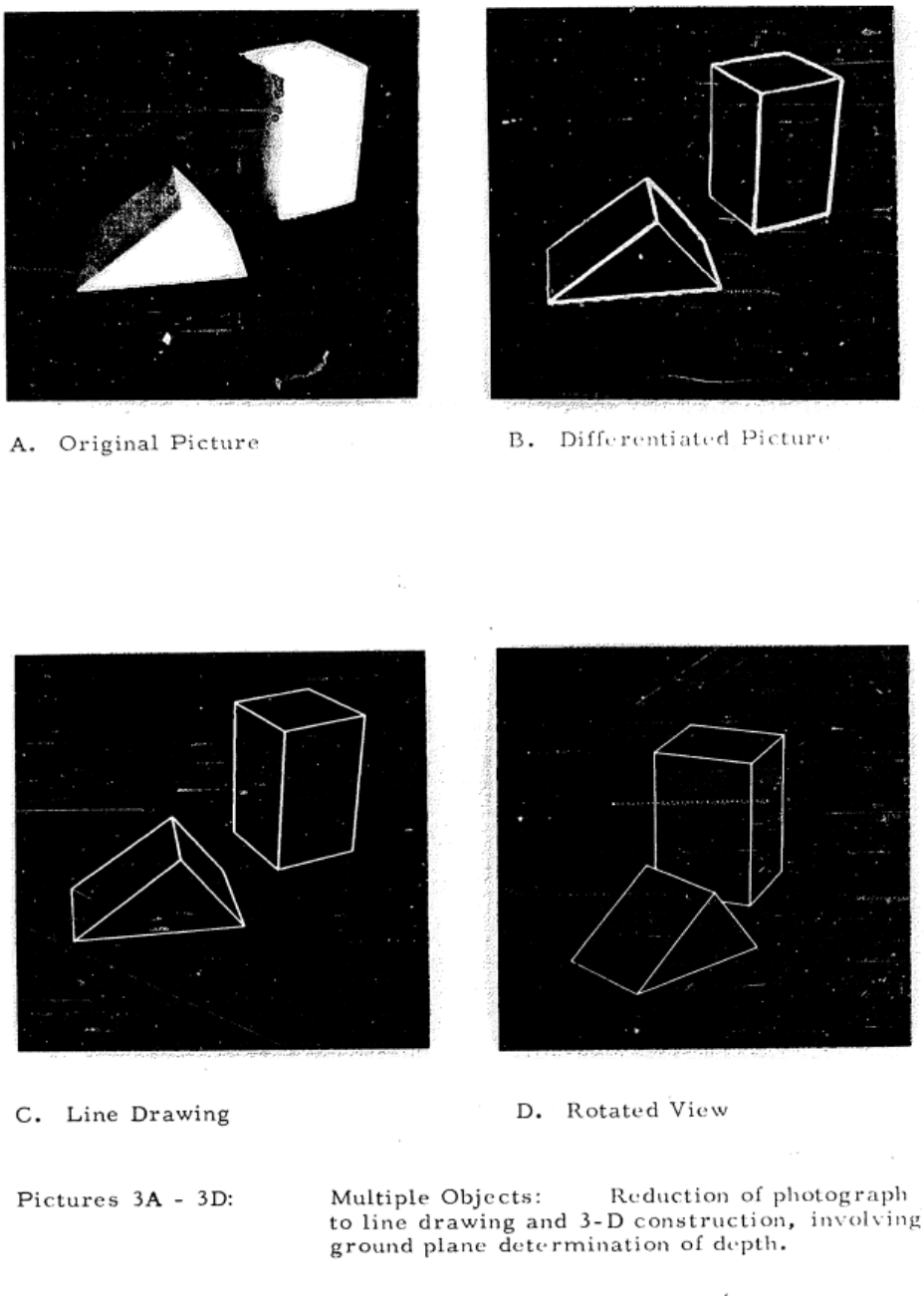


Figure 2.3.: Stages from Roberts' 1963 thesis: the original photograph, its differentiated form, and the feature points selected from it. The pipeline recovers polyhedral structure — reliably, for scenes made of blocks. *After L. G. Roberts, MIT, 1963.*

them as parts connected by spring-like constraints on their relative positions. Both ideas are genuinely good — the second one, in particular, keeps reappearing in the literature for the next forty years — and neither one scaled beyond the handful of categories it was demonstrated on.

By the 1980s, with images becoming digital, a great deal of effort went into the front end of the pipeline instead. Edge detection was formalised and made robust, most durably by Canny, whose detector is still in use. But it is worth being blunt about how this looked from outside the field: after two decades of work, the visible output of computer vision was a picture with its outlines traced. The disappointment was reasonable, the funding reflected it, and the field entered the period known as the **AI winter** alongside expert systems and robotics, all of which had promised more than they delivered.

2.5. Why vision is ill-posed

The reason the geometric programme stalled is not that its practitioners were insufficiently clever. It is that the problem they had defined as the goal is, in the strict sense, *ill-posed* — the data do not determine the answer.

Consider what a camera or an eye actually does. A point (X, Y, Z) in the world, expressed in coordinates centred on the sensor, lands on the image plane at

$$(u, v) = \left(f \frac{X}{Z},\ f \frac{Y}{Z} \right) \tag{2.1}$$

where f is the focal length and (u, v) are image coordinates. The projection Equation 2.1 throws away one number per point, and it throws it away in a specific and unrecoverable way: for any $t > 0$, the point (tX, tY, tZ) maps to exactly the same (u, v). Every point along a ray through the optical centre produces an identical measurement. **A single image is consistent with infinitely many three-dimensional scenes**, and no amount of processing applied to that image can distinguish between them, because the information was destroyed at the sensor rather than lost somewhere downstream.

Figure 2.4 is the whole difficulty in one picture. This is the fundamental problem nature had to solve, and it solved it partly by refusing to solve it exactly. Two eyes give two projections from slightly different centres, and depth can be triangulated from the disparity between them — but only once the correspondence problem is settled, that is, only once you know which point in the left image is the same world point as which point in the right. Correspondence is itself hard, and stereo does not recover geometry to anything like the precision the Marr pipeline assumes as its goal. Human three-dimensional perception is approximate, and it is approximate in ways that are easy to demonstrate. We know roughly what shape things are, which turns out to be sufficient for catching, grasping and not walking into walls.

There is a second difference, more philosophical than mathematical, that is worth carrying through the course. Language does not exist in nature; it is generated by brains, it is one-dimensional, and it is sequential. That structure is precisely what makes it tractable to model as a probability distribution over the next symbol, which is the whole basis of the current generation of language models. Vision is not generated. There is a physical world, obeying the laws of physics and optics and material science, and an image is a measurement of it. The consequence is that visual problems do not decompose the way linguistic ones do, and methods that transfer between the two — as attention has — do so on the strength of their generality rather than because the domains are alike.

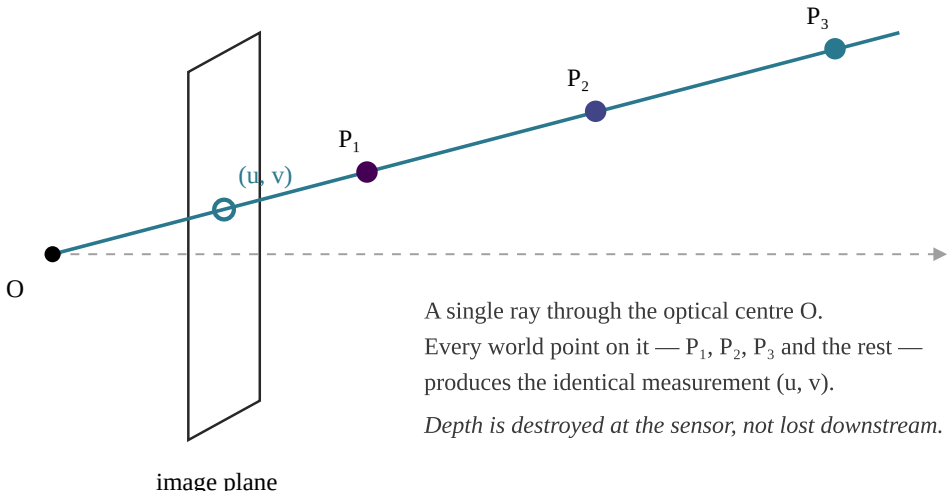


Figure 2.4.: Why a single image cannot determine the scene that produced it. Every world point along one ray through the optical centre maps to the same image coordinates, so the depth of P_1, P_2 and P_3 is not merely difficult to recover — it is absent from the measurement. *Drawn for these notes.*

The lesson the field eventually took from all this is that reconstructing the world in full is the wrong target. Most useful visual tasks do not require it. Knowing that there is a pedestrian ahead is worth more than a precise depth map of the pedestrian, and it is a far better-posed question.

2.6. What the human system tells us about the target

While computer vision was working through geometry, cognitive science and neuroscience were producing results that suggested a different set of problems worth attacking, and gave some indication of what a solution would have to look like.

The first is that recognition is *contextual*. Biederman showed that identifying a cued object in a briefly presented photograph is measurably harder when the scene has been jumbled, even though the target object itself is unchanged and the subject knows both what to look for and where it will appear (Biederman, 1972). From the sensor's point of view nothing has changed — the same photons land in the same place — so whatever is degraded must belong to the interpretation rather than the measurement. Objects are not recognised in isolation; the scene they sit in is part of the evidence.

The second is that recognition is *fast*. Rapid serial visual presentation experiments flash a sequence of unrelated images at around ten per second and ask the subject to detect a target that was never described to them in advance — not its appearance, not its position, not which frame it will occur in. Subjects do this reliably at a hundred milliseconds per frame. Thorpe and colleagues pinned the timing down directly by recording EEG while subjects decided whether a previously unseen photograph contained an animal, and found a differential signal in the brain roughly 150 ms after the image appeared (Thorpe, Fize and Marlot, 1996).

That number is worth sitting with, because its significance is easy to miss. Compared with a modern processor 150 ms is enormous. Compared with the hardware actually doing the work it is almost nothing: neurons are slow, and 150 ms buys only a small number of successive

processing stages. Whatever the visual system is doing to categorise a novel natural image, **it is doing it in a few steps, largely in one direction, without time for extended iterative refinement**. A pipeline that reconstructs three-dimensional structure before it will commit to naming anything is not a plausible model of that.

The third result is that the system is specialised. Work in the 1990s identified cortical regions responding selectively to particular categories — faces, places, body parts — which says that the mature visual system is not a uniform mechanism applied to everything but has developed dedicated machinery for the categories that matter.

Taken together these findings redirected the field's attention. If human vision is fast, context-sensitive and organised around categories, then the problem to work on is not sketching the outline of a wooden block. It is **recognising objects in natural scenes**, at the scale and variety at which they actually occur, and that reframing is what the following two decades of computer vision are largely about.

2.7. Features, benchmarks, and the turn to learning

Through the 1990s — nominally still the AI winter, though the research did not stop — computer vision reorganised itself around problems it could actually make progress on. Rather than reconstructing scenes, the field asked how an image could be partitioned into coherent regions, treating segmentation as a graph problem and separating foreground from background by cutting the graph where the affinities are weakest. Rather than matching whole objects, it asked what could be computed at a point in an image that would survive the object being moved, rotated, rescaled or relit. The answer, in the form of local descriptors of which SIFT is the best known, made it possible to match parts of one image to parts of another with some robustness, and matching turned out to support a great deal.

The result that convinced people outside the field arrived in 2001, when Viola and Jones demonstrated face detection that ran in real time on the hardware of the day. It matters here for two reasons beyond the application. It was one of the first genuinely successful uses of machine learning in vision — the detector's cascade of weak classifiers was *trained*, not designed — and within a few years it was shipping inside consumer digital cameras as autofocus that found faces. The gap between a research result and a product in a shop had become short.

The other thing that changed in this period was that the field acquired data. Digital cameras and the internet arrived together, and images stopped being scarce. Caltech 101 and the PASCAL Visual Object Classes challenge assembled labelled collections in the thousands to tens of thousands of images and, more importantly, established the practice of evaluating competing methods on a common held-out set. A benchmark converts a matter of opinion into a number, and once the number exists, progress on it can be demanded.

This is where the story pauses, because the thing that resolved it was not developing inside computer vision at all.

2.8. The neural network thread

Running in parallel with everything described so far, and largely ignored by it, was a line of work on learning systems built from simple interconnected units.

It begins with the perceptron, Rosenblatt's late-1950s model of a single artificial neuron that computes a weighted sum of its inputs and thresholds it. The perceptron could be trained, which

was the striking part, and the enthusiasm it generated ran well ahead of what it could do. In 1969 Minsky and Papert established the limits precisely: a single-layer perceptron computes only linearly separable functions, and *XOR* — true when exactly one of two binary inputs is true — is not one of them. Their analysis was correct. The reception of it, which took the result as an indictment of neural networks generally rather than of one-layer networks specifically, set the area back by years.

Work continued regardless. In 1980 Fukushima described the **Neocognitron**, a computational model of the visual system built directly on Hubel and Wiesel's hierarchy: layers of simple cells performing what we would now call convolution, interleaved with layers of complex cells pooling their outputs, stacked five or six deep. Architecturally this is a convolutional network, and it is worth registering how early that is. What it lacked was any way to learn its parameters. Every one of its several hundred weights was set by hand, which is an engineering achievement and a dead end — the approach cannot be scaled by anyone, including its author.

The missing piece arrived in 1986, when Rumelhart, Hinton and Williams described **backpropagation** as a practical method for training networks with multiple layers. The idea is to define an objective measuring the discrepancy between what the network output and what it should have output, then propagate that error backwards through the network to obtain the gradient of the objective with respect to every parameter. Mechanically it is the chain rule of calculus applied systematically to a composition of functions, and we will derive it properly in a later lecture. Conceptually it replaced hand-tuning with optimisation, and that is what made networks of any depth trainable at all.

The two ideas were combined by LeCun and colleagues through the 1990s, culminating in a convolutional network of around seven layers trained by backpropagation to read handwritten digits. It worked, and it was deployed commercially to process cheques. Compared with a modern architecture the differences are matters of detail.

And then, for over a decade, the approach did not go much further. Networks that read digits did not generalise to photographs of cats, chairs and flowers. Deeper networks were attempted through the 2000s without becoming a mainstream research topic. **The reason was not architectural, and this is the point of the whole section.** These models have very high capacity — many parameters, and correspondingly many functions they can represent — and a high-capacity model fitted to a small dataset will reproduce that dataset without learning anything that transfers to new examples. The relationship between capacity, data and generalisation is a mathematical one, which we will treat properly when we get to overfitting and regularisation. In the 1990s it was largely invisible, because the field's attention was on architectures, and data was regarded as a practical inconvenience rather than a component of the method.

2.9. 2012

The response to that diagnosis was ImageNet: an attempt to build a dataset at a scale nobody had previously thought necessary. The collection eventually held roughly fourteen million images labelled across some twenty-two thousand categories, a number chosen with reference to the psychological literature on how many object categories a child acquires in early life. Building it required organising annotation on an industrial scale, and its authors' bet was explicit — that the constraint on visual recognition was not the model but the data.

The dataset alone would not have settled anything. What made it decisive was the competition built on top of it. The ImageNet Large Scale Visual Recognition Challenge distributed a curated

subset of roughly 1.4 million images across 1,000 object classes and invited anyone to submit a system, evaluated on a held-out set by a single number, with no constraint whatsoever on method (Russakovsky et al., 2015). In the first years the best systems produced top-5 error rates near 30%, and progress between one year and the next was incremental.

In 2012 an entry from Krizhevsky, Sutskever and Hinton, a convolutional network trained on GPUs with about 60 million parameters, recorded a top-5 error of 15.3% against 26.2% for the next-best submission (Krizhevsky, Sutskever and Hinton, 2012). **The error rate was cut nearly in half in a single year**, by a method the rest of the field was not using.

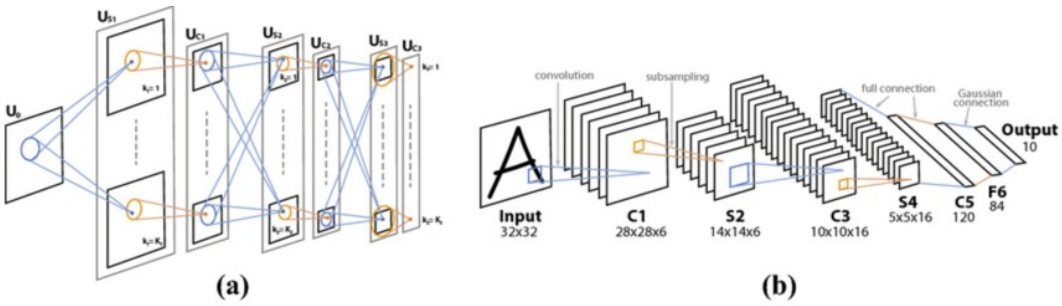


Figure 2.5.: The Neocognitron of 1980 beside a LeNet-style convolutional network of 1998. Alternating convolution and pooling, stacked with depth, is present in both; what separates them is that one was trained and the other was tuned by hand. *After Fukushima (1980) and LeCun et al. (1998).*

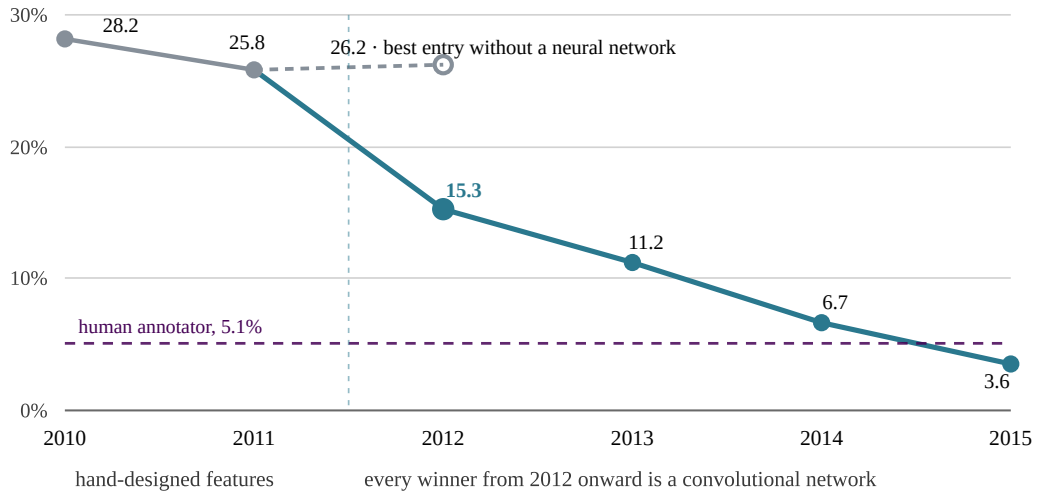


Figure 2.6.: Winning top-5 error on the ImageNet challenge. The 2010 and 2011 winners used hand-designed features and improved by about two points between them; the best 2012 entry that did not use a neural network scored 26.2%, essentially where the field had been the year before. *Plotted from the official ILSVRC results; the 5.1% human figure is from Russakovsky et al., 2015.*

Figure 2.6 shows why the year is remembered. The gap between the two 2012 points matters more than the drop from 2011: both were produced under identical conditions on identical data, and the difference between them is the method.

The comparison that makes the result legible is with Fukushima's Neocognitron, thirty-two years earlier. The architectures are recognisably the same idea — convolution and pooling, stacked.

Almost everything that separates them lies outside the architecture. Backpropagation supplied a principled way to fit the parameters instead of setting them by hand. ImageNet supplied enough data for a model of that capacity to generalise rather than memorise. Graphics hardware supplied enough arithmetic to train it in a feasible amount of time.

The honest summary of 2012 is therefore not that someone invented a better network. It is that **three independent curves — algorithms, data, and compute — reached the point where a thirty-year-old idea started working**, and the field noticed simultaneously. That is why the date is treated as the beginning of the modern era, and why the rest of this course spends comparatively little time on architecture as such and a great deal on training, data and optimisation.

2.10. What followed, and what did not

The decade after 2012 is easiest to characterise by what stopped being difficult. Classifying an image into one of a thousand categories went from a research problem to a solved component, and attention moved to questions that assume it: locating every object in a scene rather than naming the dominant one, assigning a category to every pixel rather than to the image, and doing either of these on video, where the temporal dimension carries most of the information about what is actually happening. Each of these is a lecture later in the course.

Some of what followed was less predictable. Networks were trained to produce a sentence describing a photograph, which requires a representation shared between vision and language rather than either separately. Others were trained to describe the *relationships* between the objects they found — that a person is riding a horse rather than merely that both are present — which is a step from perception towards something more like understanding. Style transfer showed that the intermediate representations of a network trained for classification implicitly separate content from appearance, which nobody had asked it to do. Generative models progressed from producing recognisable faces to producing plausible images from a text description.

The compute curve underneath all of this is worth looking at directly, because it explains the pace.

The two curves are connected. The GPU marked in Figure 2.7 is the one AlexNet was trained on, a pair of them, in 2012; by 2020 the same money bought more than seven times the arithmetic. Once deep learning became the thing these chips were bought for, the hardware began to be designed for the workload, with dedicated units for the matrix operations that dominate training. The relationship runs in both directions: better hardware made larger models trainable, and the demand for larger models redirected hardware design.

By the end of the decade the field's standing had changed outside it as well. The 2018 Turing Award went jointly to Bengio, Hinton and LeCun for the work on deep learning, and in 2024 Hinton shared the Nobel Prize in Physics with Hopfield for the foundational discoveries enabling machine learning with neural networks. It is unusual for a computing result to be recognised by a physics committee, and the fact that it was says something about how far outside its origins the technique has travelled.

None of which means vision is solved, and the failures are not evenly distributed. Because these systems are fitted to data, and the data is a record of human activity, they reproduce the patterns in that record including the ones we would not endorse. Face analysis systems have been shown to perform unequally across demographic groups, and the same techniques are deployed in settings — hiring, lending — where an error is not an inconvenience but a decision about someone's life. This is not a separate ethical appendix to the technical material; it is a

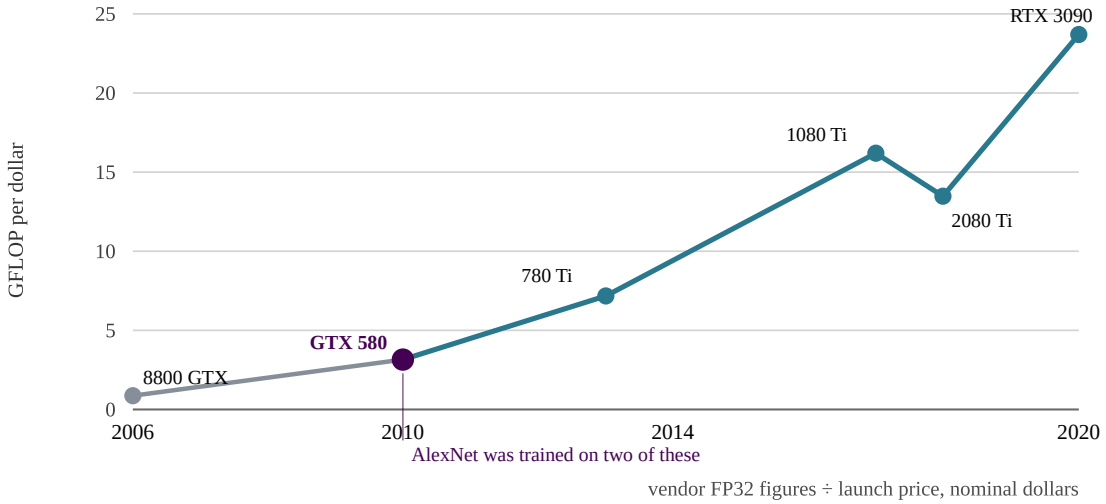


Figure 2.7.: Single-precision arithmetic per dollar across successive generations of consumer NVIDIA hardware. The trend is not smooth — the 2018 generation was worse value in FP32 terms than the one before it — but the direction after 2012 is unmistakable. *Computed from vendor FP32 specifications and launch prices in nominal dollars; not adjusted for inflation.*

direct consequence of the fact that the model's behaviour comes from the data and that nobody wrote down what it learned.

The same properties that make this technology risky also make it useful, and medical imaging is the clearest case: radiology and pathology are deeply visual disciplines where a system that reliably attends to the whole image extends what a clinician can do.

The last thing worth saying about the state of the art is how much of ordinary human vision remains out of reach. We do not have systems that read a scene the way a person does — that see the joke in a photograph, or infer what someone is about to do from their posture, or notice that a situation is unusual without having been told what unusual means. **Recognition is largely solved; understanding is not**, and the gap between them is most of what makes the field still worth working in.

2.11. The shape of this course

What follows is organised as a sequence of increasing capability, and it is worth seeing the whole arc now so that each piece arrives as an answer to a difficulty raised by the one before it.

Everything starts from **image classification**: given an image, produce a label. The task is deliberately impoverished, and it is the right place to begin because it isolates the difficulty. Whatever makes vision hard is already present in deciding whether a photograph contains a cat, and none of it is hidden behind the complexity of a richer output.

The simplest thing that could work is a *linear classifier*. If an image is flattened into a vector $x \in \mathbb{R}^d$ — for a 32×32 colour image, $d = 3072$ — then a score for each of C candidate classes can be computed as

$$f(x, W) = Wx + b \tag{2.2}$$

with a weight matrix $W \in \mathbb{R}^{C \times d}$ and a bias vector $b \in \mathbb{R}^C$. The predicted class is the one with the highest score. Geometrically, each row of W defines a hyperplane in $\mathbb{R}^d$, and the classifier carves the input space into regions with flat boundaries, as in Figure 2.8.

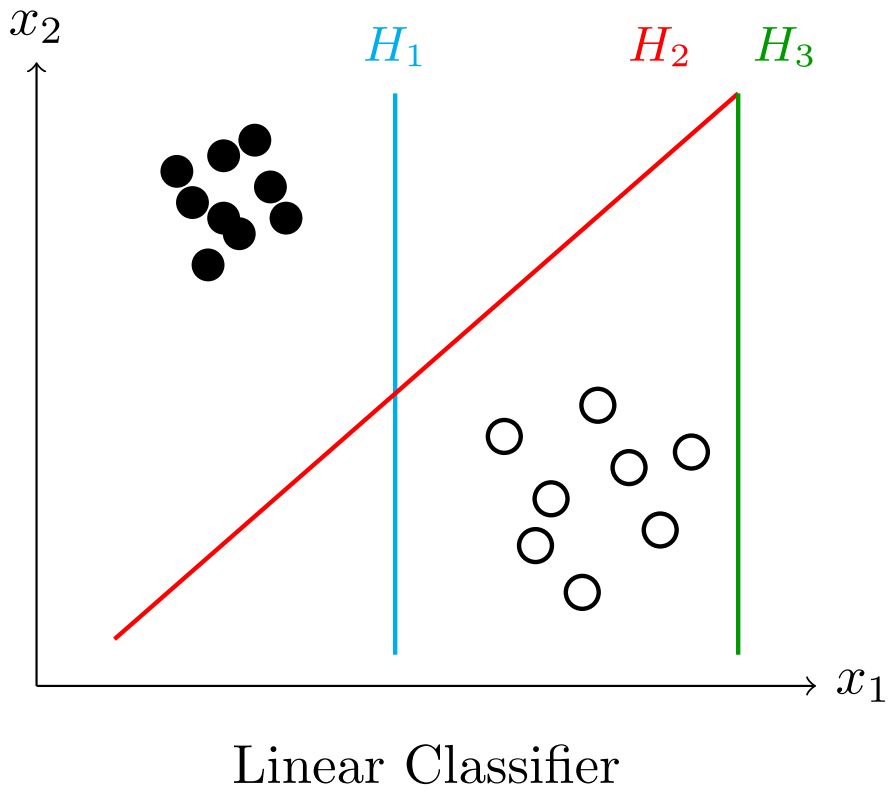


Figure 2.8.: A linear classifier in a two-dimensional feature space. The decision boundary is a straight line, which is sufficient when the classes separate cleanly and is the source of the difficulty when they do not. *Drawn for these notes.*

This immediately raises the two questions the next several lectures answer: how do we measure how wrong a particular W is, which is the subject of *loss functions*, and how do we search for a better one, which is *optimisation*. Alongside them sits the tension introduced in Section 2.8 — a model flexible enough to fit the training data can fit it too well, and *regularisation* is how that is controlled.

Flat decision boundaries are not enough, because the classes we care about are not linearly separable in pixel space. Stacking layers of linear maps separated by non-linear functions produces a *neural network*, which can represent far richer decision boundaries, and backpropagation makes such a network trainable. Applying the same idea with weights shared across spatial positions gives the **convolutional neural network**, and the reason to share weights is the observation from Section 2.3: a feature worth detecting in one part of an image is worth detecting everywhere.

With classification established, the course widens the definition of the task. *Semantic segmentation* asks for a label at every pixel — grass here, cat there, sky above — without distinguishing one cat from another. *Object detection* asks for the location as well as the identity, drawing a box around each instance. *Instance segmentation* combines them, producing a separate pixel mask for each individual object, and is the most demanding of the three.

Figure 2.9 makes the progression concrete. Each task asks more of the same input, and each requires the representation to preserve something the previous one could discard — position,

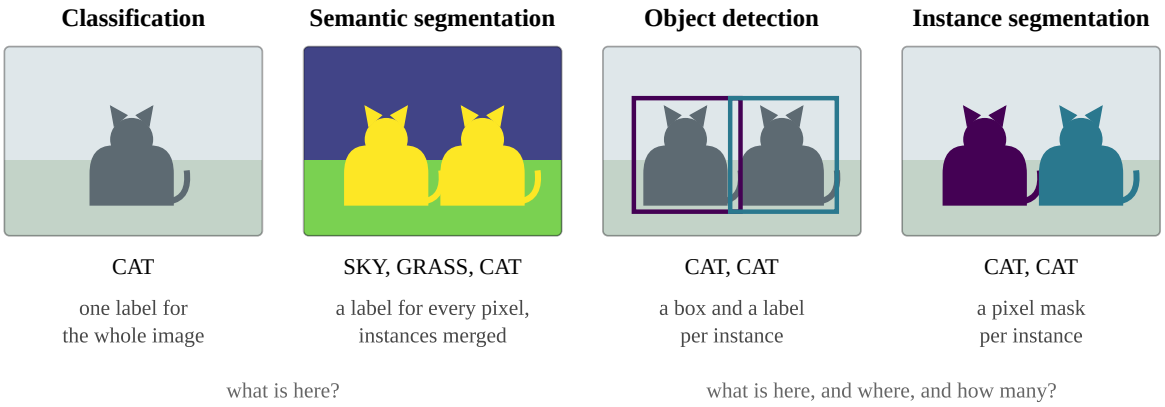


Figure 2.9.: The same scene under four recognition tasks. The distinction that matters most is between the second panel and the fourth: semantic segmentation labels every pixel but has no notion of how many cats there are, while instance segmentation separates them. *Drawn for these notes.*

extent, and finally the identity of individual objects. Video adds time, which introduces both the tasks of recognising actions rather than objects and the problem of combining modalities, since what is happening in a video is often carried as much by its audio as by its frames. Running alongside these is the question of *visualisation and interpretability*: what has a trained network actually learned, and what in an image is it responding to when it makes a decision.

The models then broaden in the same way. Recurrent networks handle sequential data; attention mechanisms and **transformers** replace recurrence with a mechanism that relates every element of a sequence to every other, and have since become the dominant architecture in vision as well as language. As models and datasets grow, training stops fitting on one device, and distributed training becomes a topic in its own right — splitting the data across workers that each hold a copy of the model, or splitting the model itself across devices, and paying the coordination costs that either choice implies.

The final part of the course leaves supervised recognition behind. *Self-supervised learning* derives a training signal from the structure of unlabelled data itself, which is what makes it possible to use the enormous quantity of images that nobody has annotated. *Generative models* invert the problem: rather than mapping an image to a label, they produce images, whether by recombining the content of one image with the appearance of another, or — in the case of *diffusion models*, which learn to reverse a gradual noising process — by generating a novel image from a text description. *Vision–language models* place images and text in a shared representation space, so that either can be used to retrieve or generate the other. *3D vision* returns to the problem of Section 2.5 with modern tools, and *embodied intelligence* connects perception to action, which is where vision started in Section 2.2 and is arguably where it belongs.

The through-line is that each of these is a different answer to the same question: what representation of an image supports the thing we want to do with it. Keeping that question in view is more useful than memorising any particular architecture, because the architectures will change and the question will not.

References

Works referred to above, in the order they arise. Where no link is given, the citation is complete enough to locate the paper directly.

- L. G. Roberts, *Machine Perception of Three-Dimensional Solids*, PhD thesis, MIT, 1963. MIT DSpace
- D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurones in the cat's striate cortex", *Journal of Physiology* 148:574–591, 1959. doi:10.1113/jphysiol.1959.sp006308
- D. J. Felleman and D. C. Van Essen, "Distributed hierarchical processing in the primate cerebral cortex", *Cerebral Cortex* 1(1):1–47, 1991. doi:10.1093/cercor/1.1.1
- I. Biederman, "Perceiving real-world scenes", *Science* 177(4043):77–80, 1972. doi:10.1126/science.177.4043.77
- S. Thorpe, D. Fize and C. Marlot, "Speed of processing in the human visual system", *Nature* 381:520–522, 1996. doi:10.1038/381520a0
- D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*, W. H. Freeman, 1982.
- J. Canny, "A computational approach to edge detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8(6):679–698, 1986.
- J. Shi and J. Malik, "Normalized cuts and image segmentation", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22(8):888–905, 2000.
- D. G. Lowe, "Distinctive image features from scale-invariant keypoints", *International Journal of Computer Vision* 60(2):91–110, 2004.
- P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features", *CVPR*, 2001.
- F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain", *Psychological Review* 65(6):386–408, 1958.
- M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, 1969.
- K. Fukushima, "Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", *Biological Cybernetics* 36:193–202, 1980.
- D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors", *Nature* 323:533–536, 1986. doi:10.1038/323533a0
- Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE* 86(11):2278–2324, 1998.
- J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "ImageNet: a large-scale hierarchical image database", *CVPR*, 2009.
- A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet classification with deep convolutional neural networks", *NeurIPS*, 2012. Proceedings
- O. Russakovsky et al., "ImageNet Large Scale Visual Recognition Challenge", *International Journal of Computer Vision* 115:211–252, 2015. doi:10.1007/s11263-015-0816-y

3. Image Classification with Linear Classifiers

Lecture 2

Based on the lecture by Fei-Fei Li and Ehsan Adeli — CS231n, Stanford University, Spring 2025.

3.1. The semantic gap

Image classification asks for a function that maps an image to one label from a fixed set: cat, dog, truck, plane, and so on. The request sounds like ordinary sorting—inspect an object, identify its category—but the input available to a computer is not an object. It is a tensor of numbers. An RGB image of width W and height H is an array in $\mathbb{R}^{H\times W\times 3}$, with each entry usually representing an intensity between 0 and 255. The category is not written anywhere in that array. It has to be inferred.

The difference between the representation and the meaning is the semantic gap. A small change in the scene can change nearly every pixel while leaving the label intact: moving the camera changes the viewpoint, a cloud changes the illumination, and a different background changes many more pixels than the object itself. Objects can be partly hidden, deformed, or surrounded by clutter. A category also contains large *intra-class variation*: two dogs can differ in pose, breed, scale, and colour more than a dog resembles another dog in a particular image. Context can help or mislead as well. A small object may be hard to recognise in isolation but obvious in a familiar scene.

There is no short collection of rules that solves all of these cases. One can write an edge detector—Canny's formulation is a canonical example—and detect corners or contours, but the step from local structure to "this is a cat" requires choices about which structures matter and how they combine. The data-driven approach changes the question. Instead of programming a recognition procedure directly, collect labelled examples, fit a classifier to them, and measure its predictions on images it has not seen.

That separation introduces three objects that will recur throughout the course. A model maps an input to a prediction. A loss function says how unsatisfactory that prediction is for a labelled example. An optimization procedure changes the model's parameters to reduce the average loss on the training set. Figure 3.1 shows how these pieces connect, and the first two classifiers in this lecture make the contrast between the choices explicit.

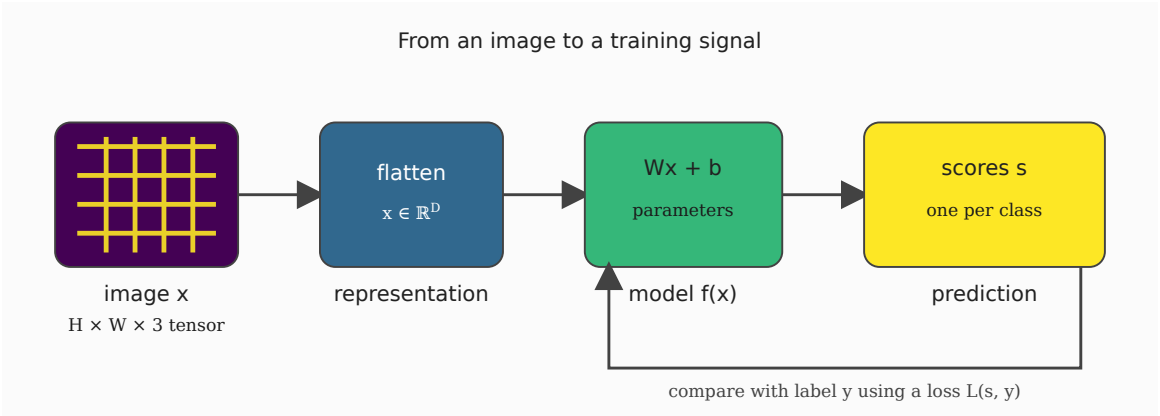


Figure 3.1.: Training separates the representation of an image, the model that produces class scores, and the loss that compares those scores with the label. The same separation will let us replace the linear model with a neural network without changing the basic training logic. *Drawn for these notes.*

3.2. Nearest neighbors: let the dataset do the work

The nearest-neighbor classifier stores every training image and its label. Given a new image x, it compares x with each stored image, finds the closest one under a chosen distance, and returns that image's label. With a training set $\{(x_i, y_i)\}_{i=1}^{N}$, the rule is

$$\hat{y}(x) = y_{\arg\min_i d(x, x_i)}. \tag{3.1}$$

For images, the simplest distance is the L_1 or Manhattan distance. If images are flattened into vectors, it is

$$d_1(x, z) = \sum_{p=1}^{D} |x_p - z_p|, \tag{3.2}$$

where $D = H \times W \times 3$ is the number of scalar pixels. The L_2 distance instead measures the Euclidean distance,

$$d_2(x, z) = \sqrt{\sum_{p=1}^{D} (x_p - z_p)^2}. \tag{3.3}$$

Nearest neighbors are useful because they reveal what a data-driven method means in its most literal form: the classifier does not learn a compact description of a class; it relies on the training set as a memory. That makes training almost free—there is nothing to fit—but makes prediction expensive. To classify one test image, the algorithm must compare it against all N training images, so prediction costs $O(ND)$ in a direct implementation. In machine learning it is usually acceptable for training to be slow if the resulting model predicts quickly; nearest neighbor has the opposite trade-off.

The k-nearest-neighbor variant reduces sensitivity to the one accidental match that Equation 3.1 rests on. It finds the k closest examples and predicts the majority label among them. The integer k and the distance metric are *hyperparameters*: choices about how the algorithm should

operate, rather than parameters inferred from the labelled examples. A larger k smooths over individual examples but can erase small classes or local structure.

This baseline also exposes why raw pixel distances are a poor notion of visual similarity. A one-pixel translation, a tint, or an occluder can produce a large numerical change even when a person would regard the image as the same object. Conversely, two images with similar low-level statistics may depict different categories. The problem is not that Equation 3.2 is badly implemented, and switching to Equation 3.3 does not repair it. The representation and metric have failed to express the invariances that recognition needs.

3.3. Validation is part of the method

Hyperparameters cannot be selected by minimizing training error. With $k = 1$, every training image is its own nearest neighbor, so the training accuracy is perfect even if the classifier generalizes badly. Choosing the setting that performs best on the test set is also invalid: the test set would then influence the method, and its final accuracy would no longer be an honest estimate of performance on new data.

The usual division is into training, validation, and test sets. Training examples define the fitted model; the validation set chooses among hyperparameters such as k and the distance; the test set is held back until the end. The validation score estimates how each choice generalizes while preserving the test set as an untouched measurement.

When data are scarce, cross-validation gives a less noisy estimate. Split the training data into F folds, then repeat the following F times: hold out one fold, fit on the remaining $F - 1$, and score the held-out fold. Averaging the F scores measures a hyperparameter setting on every training example in turn rather than on one arbitrary split. This costs F times the computation and is uncommon for large deep-learning training runs, but it is a sensible way to compare a few choices on a small dataset such as CIFAR-10. CIFAR-10 contains 60,000 colour images in ten classes, conventionally divided into 50,000 training and 10,000 test examples (Krizhevsky, 2009).

The workflow matters beyond nearest neighbors. A model can always be made to look better on data that helped choose it. Validation protects against that circularity, while the test set answers the narrower question we actually care about: after all choices have been made, how well does the finished procedure work on unseen examples?

3.4. A parametric model: one matrix, one score per class

Nearest neighbors retain all examples. A parametric classifier takes the opposite approach: it compresses the training set into a fixed collection of parameters. Flatten an image into $x \in \mathbb{R}^D$, let there be C classes, and define

$$f(x, W, b) = Wx + b, \tag{3.4}$$

where $W \in \mathbb{R}^{C \times D}$ is the weight matrix and $b \in \mathbb{R}^C$ is the bias vector. The output $s = f(x, W, b)$ contains one score s_j for each class j; prediction chooses the largest score, $\hat{y} = \arg\max_j s_j$.

For CIFAR-10, $D = 32 \cdot 32 \cdot 3 = 3072$ and $C = 10$, so the matrix in Equation 3.4 has shape 10×3072. Each row of W can be viewed as a template for one class: the class score is a weighted sum of image pixels, plus its bias. In this view, positive weights favour pixels or colour channels

at particular positions, while negative weights suppress them. The geometric view says the same thing differently. Each class wins in a region of pixel space separated from other regions by hyperplanes, because equality of two class scores gives a linear boundary.

This explains both the appeal and the limitation of the model. It is compact and fast: one matrix multiplication produces all class scores. But one linear boundary cannot express arbitrary arrangements of examples. A class occupying two disconnected regions, a ring surrounding another class, or several distinct visual modes cannot generally be separated from its competitors by a single set of linear score comparisons. Flattening the image also discards the spatial structure that makes nearby pixels and repeated patterns meaningful. Later architectures will address those limitations by composing many learned transformations; for now, the point is to understand how a simple model is trained.

3.5. The loss turns "good" into a number

The classifier is not defined by the score function alone. Many different matrices W and biases b produce different scores, and we need a criterion for preferring one parameter setting to another. Given labelled examples (x_i, y_i), define a per-example loss L_i and average it over the dataset:

$$L(W, b) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W, b), y_i). \tag{3.5}$$

The loss is deliberately separate from the model. The linear map says what scores can be represented; the loss says which score vectors are desirable. Everything that follows in this lecture fills in the single term L_i inside Equation 3.5, and training becomes the optimization problem of finding parameters with small $L(W, b)$. Regularization and optimization, which the next lectures develop, add further structure to this objective; the present lecture concentrates on the data term.

3.6. Softmax: scores as a probability distribution

The softmax classifier interprets the linear scores as unnormalized log-probabilities, or logits. For score vector s, it defines

$$p_j = \frac{\exp(s_j)}{\sum_{k=1}^{C} \exp(s_k)}. \tag{3.6}$$

Every p_j produced by Equation 3.6 is nonnegative and the probabilities sum to one. The predicted class is still the class with the largest score, because exponentiation and normalization preserve the ordering. The extra interpretation is useful for training: if the correct class is y_i, the cross-entropy loss is the negative log probability assigned to it,

$$L_i = -\log p_{y_i} = -\log \left(\frac{\exp(s_{y_i})}{\sum_{j=1}^{C} \exp(s_j)} \right). \tag{3.7}$$

Equation 3.7 is small when the correct class receives nearly all the probability and grows without bound as its probability approaches zero. Minimizing the average cross-entropy is equivalent to

maximizing the likelihood of the observed labels under the model. It is also the cross-entropy between the one-hot target distribution and the predicted distribution.

At initialization, if all scores are approximately equal, each class receives probability $1/C$. The initial loss is therefore

$$L_i = -\log(1/C) = \log C. \tag{3.8}$$

For ten classes this is approximately 2.3. This is a useful sanity check: a randomly initialized ten-way classifier should begin near the loss of a uniform guess, not near zero. In numerical code, the softmax is evaluated with a stabilized form—subtracting the largest score before exponentiating—because the probabilities do not change when the same constant is subtracted from every score, while the exponentials become safer to compute.

3.7. Multiclass SVM loss: enforce a margin

The multiclass SVM loss does not ask for calibrated probabilities. It asks for a margin: the correct class should score at least Δ more than every incorrect class. For an example with score vector s and label y_i, define

$$L_i = \sum_{j \neq y_i} \max\left(0,\, s_j - s_{y_i} + \Delta\right). \tag{3.9}$$

The term for class j is zero when the correct score beats it by at least Δ. Otherwise, the term is the amount by which the margin is violated. The loss is therefore piecewise linear: once all constraints are satisfied, increasing the correct score further does not improve the data loss. Piecewise linear also means not differentiable everywhere—each hinge has a corner at the point where its margin is met exactly—so optimization uses a subgradient there. In practice one picks either side of the corner, and the choice rarely matters, because exact ties are unusual in floating-point scores.

A worked example makes the arithmetic concrete. Take three training images, three classes, and $\Delta = 1$, with the scores some particular W assigns. For the cat image the scores are 3.2 for cat, 5.1 for car, and -1.7 for frog. The correct class is cat, so the car term is $\max(0, 5.1 - 3.2 + 1) = 2.9$ and the frog term is $\max(0, -1.7 - 3.2 + 1) = 0$, giving $L_i = 2.9$—the car class outscores the correct one, and the loss is exactly the amount by which that margin is violated. For the car image the scores are 1.3, 4.9, and 2.0; the correct class leads both competitors by more than Δ, both terms vanish, and $L_i = 0$. For the frog image the scores are 2.2, 2.5, and -3.1, and the correct class comes last by a wide margin: the terms are $\max(0, 2.2 + 3.1 + 1) = 6.3$ and $\max(0, 2.5 + 3.1 + 1) = 6.6$, so $L_i = 12.9$. Averaging the three gives the dataset objective, $L = (2.9 + 0 + 12.9)/3 \approx 5.27$.

The same sanity check that Equation 3.8 provides for softmax is available here. If the weights are initialized small enough that every score is approximately zero, each of the $C - 1$ incorrect classes contributes $\max(0, 0 - 0 + \Delta) = \Delta$, so the loss at the first iteration should be close to

$$L_i \approx (C - 1)\,\Delta. \tag{3.10}$$

For ten classes and $\Delta = 1$ that is about 9. A first measurement far from this value points to a bug in the implementation rather than to an unlucky initialization.

The loss has a useful invariance. Adding the same constant to every score leaves each difference $s_j - s_{y_i}$ unchanged, so it leaves Equation 3.9 unchanged. Softmax probabilities have the same invariance: their common factor cancels during normalization. What matters in both cases is the relative arrangement of the class scores, not their absolute origin.

The two objectives differ in what happens after a prediction is already correct. SVM stops caring once the margin is met. Softmax continues to reward a larger probability for the correct class, pushing the distribution toward greater confidence. That does not make one universally correct and the other universally wrong; it means they encode different preferences about the scores.

3.8. Softmax and SVM are choices above the same model

Softmax and SVM can be applied to the same linear score function $Wx + b$. They differ in the loss layer, not in the basic model that converts pixels to scores. In practice they may produce similar predicted labels, especially when the data are easy, even though the numerical objectives behave differently. A score vector that already separates the correct class by a large margin can have zero SVM loss while still changing the softmax loss if its confidence changes.

This separation is the main conceptual handoff to the next lectures. A classifier is not just an architecture name. It is a chain of decisions: how the image is represented, how parameters map it to scores, how scores are judged against labels, and how parameters are optimized. Nearest neighbor made the training set itself the model. The linear classifier compresses the examples into W and b. Softmax and SVM then impose different notions of what those scores should mean. Once these layers are kept distinct, improving a vision system becomes a tractable question: which part of the chain is limiting the result, and what evidence would tell us that?

References

- J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6), 1986. doi:10.1109/TPAMI.1986.4767851
- A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Technical Report, University of Toronto, 2009. Report

4. Regularization and Optimization

Lecture 3

Based on Lecture 3 of CS231n, Stanford University, Spring 2025.

4.1. Two questions the loss leaves open

The objective built in the previous chapter, Equation 3.5, turns a classifier into a number, but it does not single out a classifier. Take the car image from that chapter's worked example. With $\Delta = 1$ the scores 1.3, 4.9 and 2.0 put the correct class more than a margin ahead of both competitors, so $L_i = 0$. Now double every entry of W. The scores become 2.6, 9.8 and 4.0; the margins double along with them; and the loss is unchanged, since $\max(0,\, 2.6 - 9.8 + 1) = 0$ and $\max(0,\, 4.0 - 9.8 + 1) = 0$. Any scaling by a factor greater than one preserves a zero loss, so the objective is minimized not by one W but by an unbounded family of them, and nothing written down so far prefers any member of that family over the others.

That is the first gap. The second is more elementary: Equation 3.5 says how to score a candidate W, and says nothing whatsoever about how to produce one. Both gaps have to be closed before a classifier can be trained, and each is answered by half of this chapter—the first by adding a term to the objective, the second by a procedure for descending it.

4.2. Regularization

The fix for the first gap is to stop asking the objective to measure only fit. Write it as a sum of two terms,

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W), \tag{4.1}$$

where the first term is the *data loss* of Equation 3.5 and the second is a *regularization* term that depends on the weights alone and never looks at the data. Because R ignores the training examples, it cannot express any opinion about accuracy; it can only express a preference among the weight settings that the data leaves tied.

Stated that way, regularization sounds like a tiebreaker, which undersells it. Its more important effect is to make the model fit the training data *less* well on purpose. Consider fitting a curve to a handful of noisy points, as in Figure 4.1. A high-degree polynomial can pass exactly through every one of them and drive the data loss to zero. A straight line cannot, and by the only measurement available at training time it is the worse model. But the wiggles that let the polynomial hit each point are reconstructions of the noise in those particular points, and there

is no reason for the noise in a fresh sample to lie the same way. The straight line, wrong on the training set, is the one that transfers.

Figure 4.1.: Nine noisy observations of a straight-line relationship, with two models fitted to them. The polynomial f_1 passes through every point and has zero training error; the line f_2 misses all of them. On new data drawn from the same source, f_2 is the better predictor. *Drawn for these notes.*

This is Occam's razor recast as an optimization problem: among competing hypotheses that explain the observations, prefer the simplest, and only reach for a more elaborate one when the simple hypothesis is shown to be wrong. The razor is a heuristic about the world rather than a theorem, and Equation 4.1 is a way of asserting it numerically.

The scalar λ sets how loudly that assertion is made. At $\lambda = 0$ the regularizer is switched off and Equation 4.1 collapses back to the data loss. As λ grows, the optimizer becomes progressively more willing to accept a worse fit in exchange for a smaller penalty, until at very large λ the weights are driven towards zero and the model stops fitting anything at all. Between those extremes lies a value that generalizes best, and it cannot be found by minimizing Equation 4.1—raising λ always makes the training objective's job harder, so training loss cannot vote on it. It is a hyperparameter, chosen the same way k was chosen for nearest neighbors in Section 3.3: by measuring on held-out data.

Three choices of R account for most of the classical use. The first, and by far the most common in vision, is L_2 regularization, also called weight decay:

$$R(W) = \sum_{k} \sum_{l} W_{k,l}^2. \tag{4.2}$$

The second replaces the square with an absolute value, giving L_1 regularization:

$$R(W) = \sum_{k} \sum_{l} |W_{k,l}|, \tag{4.3}$$

and the third, elastic net, is their weighted sum, $\sum_{k,l} \beta W_{k,l}^2 + |W_{k,l}|$, which trades one behaviour against the other. All three are penalties on the *size* of the weights, which is why they answer the question this section opened with: doubling W leaves the data loss at zero but multiplies Equation 4.2 by four, so the scaled copy is no longer tied with the original.

These are not the only regularizers, or even the ones that do the most work in a modern network. Dropout, batch normalization and stochastic depth all belong to the same family in intent—each makes training harder in order to make testing better—but they act on the layers of the model rather than on a penalty added to the loss, and they are taken up later in the course. What Equation 4.1 shares with them is the trade being made, not the mechanism.

Three separate motives are worth keeping distinct, because they justify regularization on different grounds. It expresses a preference over weights, when the shape of the problem gives a reason to prefer one kind of solution. It makes the model simpler, so that it depends on the training sample less and transfers better. And, for L_2 specifically, it improves the optimization itself: adding a quadratic term adds curvature to the objective, which is a property the second half of this chapter will show to be valuable in its own right.

4.3. What L_1 and L_2 each prefer

The two penalties are usually distinguished by asserting that L_1 produces sparse weights and L_2 produces small, spread-out ones. That is the right conclusion, but the standard example for it is more interesting than the slogan suggests.

Take an input $x = [1, 1, 1, 1]$ and two candidate weight vectors, $w_1 = [1, 0, 0, 0]$ and $w_2 = [0.25, 0.25, 0.25, 0.25]$. Both give the same score, since $w_1^\top x = w_2^\top x = 1$, so the data loss cannot distinguish them at all and the regularizer decides alone. Under Equation 4.2 the first costs $1^2 = 1$ and the second costs $4 \times 0.25^2 = 0.25$, four times less: L_2 prefers w_2, because squaring punishes a single large entry far more than it punishes several small ones summing to the same total. Spreading the weight out is exactly what it rewards, and there is a reading of that behaviour in terms of the model rather than the arithmetic—a classifier whose weight is concentrated on one input dimension is betting everything on one measurement, and L_2 pushes it to take account of all four.

Now ask the same question of Equation 4.3, and the answer is that there is no preference. The first vector costs 1 and the second costs $4 \times 0.25 = 1$; they tie exactly. This is not an artefact of the numbers chosen. The L_1 penalty of any vector is unchanged by moving mass between coordinates, so it is indifferent to how spread out the weights are and cares only about their total magnitude.

Which raises the obvious objection: if L_1 does not prefer the sparse vector here, where does its reputation for sparsity come from? Not from comparing endpoints, but from what happens along the way. The penalties differ in their derivatives. The gradient of w^2 is $2w$, which shrinks as w approaches zero, so under L_2 the pressure on a weight to get smaller weakens exactly as it becomes small—the last stretch to zero is pushed with almost no force, and weights settle at small non-zero values. The gradient of $|w|$ is $\text{sign}(w)$, which has the same magnitude no matter how small w is, so under L_1 a weight that is not earning its place is pushed towards zero with undiminished force until it arrives there. Sparsity is a consequence of the optimization dynamics, not of the penalty's value at the solution.

Both stories reduce to the same underlying trade, which is worth stating in the general form because it governs every regularizer in the chapter. The optimizer is minimizing a sum. If a change to the weights lowers the regularization term without meaningfully raising the data loss, it lowers the total, and the optimizer will make it. A weight that contributes nothing to accuracy is therefore not merely permitted to vanish—it is actively removed, because keeping it costs something and buys nothing.

4.4. Optimization as search

The second gap is the harder one. Equation 4.1 is now a complete statement of what a good W would be, and completely silent on how to obtain one.

The standard picture is a landscape. Each setting of the weights is a location, the loss at that setting is the elevation, and training is the problem of walking downhill to the lowest point. The picture is worth keeping, provided one correction is made to it: the walker is blindfolded. A person standing on a real hillside can look across the valley and see where the bottom is, and no such information is available here. What is available is purely local—the elevation underfoot and the slope of the ground at that one spot. Every algorithm in the rest of this chapter is a rule for choosing a step from local information alone, and the differences between them are differences in how much local history they are willing to remember.

Before turning to slopes, it is worth seeing what happens without them. Random search is the crudest possible strategy: draw a thousand random weight matrices, evaluate Equation 4.1 on each, and keep whichever scored best. It is not a straw man—it needs no derivatives, works on any loss whatever, and is trivially parallel. On CIFAR-10 it reaches about 15.5% test accuracy.

That number is the argument for everything that follows. Ten classes means random guessing is already right 10% of the time, so a thousand samples of the weight space bought roughly five points over chance, while networks trained by the methods in this chapter exceed 99%. The weight space of even the linear classifier from Section 3.4 has 10×3072 dimensions, and sampling it blindly is hopeless at any budget one might realistically spend. The loss surface has structure, and the only way to reach a good W is to use it.

4.5. Following the slope

The structure worth using is the slope. In one dimension it is the derivative, defined as the limit of a ratio of differences,

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}, \tag{4.4}$$

which reports how much the function changes per unit of movement. In many dimensions the analogous object is the *gradient* $\nabla_W L$, the vector whose entries are the partial derivatives of L with respect to each weight in turn. It has the same shape as W itself, so a linear CIFAR-10 classifier has a gradient with 30,720 entries, one per parameter.

Two facts about the gradient make it the right object. The slope of the loss along any direction u is the dot product $u^\top \nabla_W L$, so a single gradient answers the question "what happens if I move this way?" for every direction at once. And that dot product is largest when u points along $\nabla_W L$, which means the gradient points in the direction of steepest ascent and its negative points in the direction of steepest descent. The blindfolded walker's best move is $-\nabla_W L$.

There are two ways to get it. The first is to take Equation 4.4 literally: pick a small h, perturb one weight by it, re-evaluate the loss, and divide the change by h. The deck's worked example makes the mechanics plain. With a particular W the loss is 1.25347. Adding $h = 0.0001$ to the first weight moves it to 1.25322, so that partial derivative is $(1.25322 - 1.25347)/0.0001 = -2.5$: increasing this weight lowers the loss. Perturbing the second weight instead gives 1.25353, a partial derivative of $+0.6$, so this weight should be decreased. Perturbing the third leaves the loss at 1.25347 to five decimal places, and the partial derivative is recorded as 0.

The procedure works and should not be used. It is approximate twice over—h is finite rather than infinitesimal, and subtracting two nearly equal floating-point numbers discards most of the significant digits before the division amplifies what is left. Worse, it needs one full evaluation of the loss over the dataset per weight. Filling in all 30,720 entries of that gradient means 30,721 passes over the training set to take a single step, and this for the smallest model in the course.

The second way is to notice that none of this is necessary. The loss is not a black box to be probed; it is an expression that was written down, and L as a function of W is a composition of a matrix multiply, some exponentials and logarithms, and a sum. Composition of differentiable functions is what the chain rule is for. Differentiating Equation 4.1 with respect to W, holding the data x_i and y_i fixed, yields a formula that computes the whole gradient in closed form, at a cost comparable to evaluating the loss once. This *analytic gradient* is exact and fast.

Its one weakness is that it is derived and implemented by a person, and both steps admit mistakes that do not announce themselves—a wrong sign or a dropped factor produces a gradient that still points roughly downhill, so training still appears to work, just worse than it should. The standard defence is to use both methods: compute the analytic gradient in the code that trains the model, and compare it against the numerical gradient at a few randomly chosen coordinates before trusting it. This comparison is called a *gradient check*, and it is the reason the slow method is worth knowing.

4.6. Gradient descent, and why it becomes stochastic

With a gradient in hand the algorithm is a single line, repeated:

$$W \leftarrow W - \alpha \nabla_W L(W). \tag{4.5}$$

Evaluate the gradient at the current weights, move against it, repeat. The scalar α is the *step size* or *learning rate*, and of all the hyperparameters in deep learning it is the one that most often decides whether a training run works at all.

Two details of Equation 4.5 are easy to misread. The step size is fixed, but the distance travelled per iteration is not: the update is α times a gradient whose magnitude varies, so the same α produces long strides on a steep slope and short ones as the surface flattens out. Descent naturally decelerates near a minimum without anything being scheduled. And the loop as written never terminates. In practice it is run for a fixed number of iterations, or until the loss stops improving by more than some tolerance—there is no condition internal to Equation 4.5 that signals arrival.

The real obstacle is cost. The gradient in Equation 4.5 is the gradient of Equation 4.1, which is an average over all N training examples, so one step requires touching the entire dataset. On a dataset of a million images that makes each individual step of a procedure that needs many thousands of steps prohibitively expensive.

The resolution is to accept a worse gradient in exchange for far more steps. Sample a *minibatch* of B examples at random and use it to estimate the average:

$$\nabla_W L \approx \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla_W L_i(f(x_i, W), y_i) + \lambda \nabla_W R(W). \tag{4.6}$$

This is *stochastic gradient descent*, and the word stochastic refers to the sampling: each step uses a different random subset, so the estimate is noisy and the direction is only correct on average.

Batch sizes of 32, 64, 128 and 256 are the usual choices. In implementations the batches are not drawn independently but by shuffling the dataset and cutting it into consecutive blocks, so that every example is used exactly once before any is reused; one such traversal is an *epoch*.

The trade is heavily in favour of the noisy estimate. A gradient computed on 256 examples is a poor approximation to the true one, but it costs a few thousandth of the full computation, and taking a thousand approximate steps makes far more progress than taking one exact step. SGD in this form, or one of the refinements of it developed below, trains essentially every model in this course.

4.7. Three ways SGD struggles

Plain SGD is enough to train the linear classifier of the previous chapter, and it is not enough for much beyond it. Three distinct failures account for most of the difficulty, and it is worth separating them, because a single modification fixes all three.

The first is *poor conditioning*. Suppose the loss falls steeply as one weight changes and only gently as another changes—the level sets are then long, narrow ellipses rather than circles, as in the left panel of Figure 4.2. The negative gradient at almost every point in such a valley aims mostly across it rather than along it, because the steep direction dominates the gradient's magnitude. Descent therefore bounces from wall to wall, making rapid progress in the direction where there is nothing to gain and creeping along the direction that actually leads to the minimum. If the step size is large enough relative to the steepness, the oscillation grows instead of decaying and the run diverges. The quantity that measures this is the *condition number* of the loss, the ratio of the largest to the smallest eigenvalue of its matrix of second derivatives; a high condition number means a valley whose walls are much steeper than its floor is sloped, and such valleys are the normal case rather than a pathology.

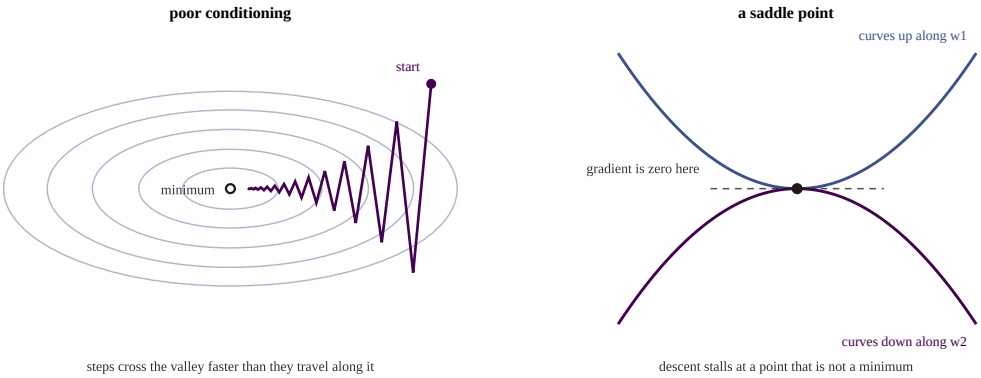


Figure 4.2.: Two of the three obstacles, drawn in the plane. Left: elongated level sets, with a descent path that oscillates across the narrow valley while advancing slowly along it. Right: the two principal cross-sections through a saddle point—the surface curves upwards along one weight and downwards along the other, and is flat along both at the centre, so the gradient there is zero. *Drawn for these notes.*

The second failure is that a zero gradient does not mean a minimum. At a local minimum the gradient vanishes and Equation 4.5 stops moving, which is the intended behaviour if the local minimum is good and a trap if it is not. More troubling is the *saddle point*, sketched in the right panel of Figure 4.2: a point where the surface curves upwards along some directions and downwards along others, so the gradient is zero while a large decrease lies just to one side.

Descent stalls there, and near it—where the gradient is small but not exactly zero—it crawls, which in practice costs more time than the exact saddles do.

Intuition from two dimensions badly understates how much this matters. In one dimension a stationary point is a minimum or a maximum; in two, a saddle is already possible. In n dimensions a stationary point is a minimum only if the surface curves upwards along *every* one of n directions, and the chance of that happening falls sharply as n grows. Empirically, in the high-dimensional losses that neural networks produce, the stationary points encountered during training are overwhelmingly saddles rather than local minima (Dauphin et al., 2014). The problem gets worse, not better, as models get larger.

The third failure is built into Equation 4.6. The gradient from a minibatch is an estimate, and estimates have error, so each step is aimed slightly wrong. The path to the minimum is not a smooth descent but a wandering one, and the wandering costs iterations. This one is a consequence of the very approximation that made SGD affordable, so it cannot be removed by computing more carefully—only by averaging.

4.8. Momentum

All three problems are complaints about a single quantity: the update depends only on the gradient at the current point, and throws away everything the walk has already learned. Momentum keeps some of it. Introduce a *velocity* v with the same shape as W, and let each gradient nudge the velocity rather than the weights directly:

$$v \leftarrow \rho v + \nabla_W L, \qquad W \leftarrow W - \alpha v. \tag{4.7}$$

The coefficient ρ, typically 0.9 or 0.99, controls how much of the past survives each step. Setting $\rho = 0$ recovers plain SGD. The physical reading is a ball rolling downhill instead of a walker being teleported: the gradient supplies acceleration rather than displacement, and ρ acts as friction. The mathematical reading is that v is an exponentially weighted running mean of the gradients seen so far, with $1/(1-\rho)$ setting roughly how many recent steps it averages over—ten steps at $\rho = 0.9$, a hundred at $\rho = 0.99$.

Both readings explain the same three repairs. In a narrow valley the across-valley components of successive gradients point in opposite directions and cancel in the running mean, while the along-valley components point the same way every time and accumulate; the zigzag is damped and the useful direction is amplified. At a saddle or a flat region the gradient is nearly zero, but v is not, so the accumulated velocity carries the iterate through instead of letting it stall. And minibatch noise is zero-mean by construction, so averaging over the last ten or hundred gradients suppresses it while leaving the signal they share.

None of this is free. A ball with momentum overshoots the bottom of a bowl and has to come back, so momentum can converge more slowly on an easy problem than plain SGD would. The bet is that overshooting a bad minimum in order to find a better one is worth the extra iterations, and empirically, for the loss surfaces neural networks produce, it usually is. It is not a law: there are models for which plain SGD wins, and choosing between them is done by trying both.

A refinement worth knowing is *Nesterov momentum*, which changes where the gradient is measured. Standard momentum evaluates the gradient at the current point and then adds it to the velocity, even though the velocity is about to move the iterate somewhere else. Nesterov's version looks ahead first, evaluating the gradient at $W + \rho v$—the point the velocity alone would reach—and mixing that into the update:

$$v \leftarrow \rho v - \alpha \nabla_W L(W + \rho v), \qquad W \leftarrow W + v. \tag{4.8}$$

The correction is anticipatory: if the velocity is about to carry the iterate up the far wall of a valley, the look-ahead gradient already points back before the overshoot happens. Written this way the update needs the gradient at a point other than the current parameters, which most software is not organized to supply; a change of variables rearranges it into a form that only ever evaluates the gradient where the parameters already are, which is how it is implemented.

4.9. Per-parameter learning rates

Momentum treats every weight identically: one α, one ρ, applied elementwise to a vector whose entries may differ in scale by orders of magnitude. The second family of methods drops that assumption and gives each parameter its own effective step size, inferred from the gradients that parameter has received.

The original is AdaGrad (Duchi et al., 2011), which accumulates the squared gradient elementwise and divides the step by its square root:

$$g \leftarrow g + (\nabla_W L)^2, \qquad W \leftarrow W - \alpha \frac{\nabla_W L}{\sqrt{g} + \varepsilon}. \tag{4.9}$$

Every operation here is elementwise, and ε—typically 10^{-7}—only prevents division by zero. The effect is a direct attack on poor conditioning. A weight along a steep direction receives large gradients, accumulates a large g, and is divided by a large number, so its steps shrink. A weight along a flat direction accumulates little and keeps taking full-size steps. Progress is damped where it was wasteful and accelerated where it was too slow, which is exactly the correction the narrow valley needed.

The flaw shows up over long runs. Since g only ever grows, the denominator only ever grows, and the effective learning rate decays monotonically towards zero. On a convex problem that is a feature, since the schedule anneals automatically. On a neural network, where training may need to keep moving for a hundred thousand iterations, it means the optimizer quietly stops before the loss has finished falling.

RMSProp repairs it with one change: replace the running sum by a running *average*, so that old gradients are forgotten.

$$g \leftarrow \delta g + (1 - \delta)(\nabla_W L)^2, \qquad W \leftarrow W - \alpha \frac{\nabla_W L}{\sqrt{g} + \varepsilon}. \tag{4.10}$$

The decay rate δ, usually 0.9 or 0.99, plays the same role for squared gradients that ρ plays for gradients in Equation 4.7. Because g now tracks the recent magnitude rather than the cumulative one, it can fall as well as rise, and the effective learning rate no longer collapses. The nickname is leaky AdaGrad, and it is accurate: Equation 4.10 is Equation 4.9 with the accumulator given a leak.

At this point two independent improvements are on the table. Momentum averages the gradients to decide *which way* to go; RMSProp averages their squares to decide *how far*. There is no reason not to do both, and Adam is the method that does (Kingma and Ba, 2015). It maintains a running mean of the gradient and a running mean of its square—the *first* and *second moments*—and combines them:

$$m \leftarrow \beta_1 m + (1-\beta_1)\nabla_W L, \qquad v \leftarrow \beta_2 v + (1-\beta_2)(\nabla_W L)^2, \tag{4.11}$$

$$W \leftarrow W - \alpha \frac{m}{\sqrt{v}+\varepsilon}. \tag{4.12}$$

Written this way the method is broken at the first step, and the reason is worth working through because it is the one part of Adam that looks arbitrary. Both accumulators start at zero, and the standard settings $\beta_1 = 0.9$, $\beta_2 = 0.999$ mean they fill up slowly. After one step with gradient g, the moments are $m = 0.1g$ and $v = 0.001g^2$, so the update in Equation 4.12 is $\alpha \times 0.1g/(0.0316\,|g|) \approx 3.16\,\alpha$. The first step is more than three times the intended learning rate, in a direction determined by a single noisy minibatch, and it is large regardless of how small the gradient actually was—the two moments are both biased towards zero, but the second is under a square root, so the bias does not cancel.

Adam's fix is to divide out the known bias. At iteration t the expected shrinkage of m is a factor $1-\beta_1^t$ and that of v is $1-\beta_2^t$, so scaling each accumulator up by its own factor removes it:

$$\hat{m} = \frac{m}{1-\beta_1^t}, \qquad \hat{v} = \frac{v}{1-\beta_2^t}, \qquad W \leftarrow W - \alpha \frac{\hat{m}}{\sqrt{\hat{v}}+\varepsilon}. \tag{4.13}$$

At $t = 1$ this gives $\hat{m} = g$ and $\hat{v} = g^2$, so the first step is exactly α, as intended. As t grows, $\beta^t \to 0$ and the correction fades to nothing, which is why it is described as a warm-up for the moment estimates rather than a permanent part of the method.

Equation 4.13 with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and α of 10^{-3} or 5×10^{-4} is a sound default for a model whose optimizer has not yet been tuned, and it is the reason Adam is the most widely used optimizer in deep learning. It is not universally best—the whole point of the previous paragraphs is that each ingredient addresses a specific failure, and a problem that does not have that failure gains nothing from the machinery that treats it.

4.10. Why AdamW is not Adam plus L_2

One question remains from the first half of the chapter: where does $\lambda R(W)$ enter, once the optimizer is no longer plain gradient descent? The natural answer is that it does not need to enter anywhere special—it is part of the loss, so its gradient is part of $\nabla_W L$, and every method above uses $\nabla_W L$. That answer is correct, and its consequences are not the ones intended.

Follow the L_2 term through Equation 4.13. Its contribution to the gradient is $2\lambda W$, proportional to the weight, which is precisely the uniform shrinkage that "weight decay" describes. But that contribution is then folded into m and v along with everything else, and the final update divides by $\sqrt{\hat{v}}$. A weight that has been receiving large gradients has a large $\hat{v}$ and therefore has its decay term divided down; a weight receiving small gradients has its decay term divided up. The regularization has stopped being uniform: how strongly a weight is pulled towards zero now depends on the recent gradient history of that weight, which is not what Equation 4.2 asked for and not something anyone chose.

AdamW decouples the two (Loshchilov and Hutter, 2019). The moments are computed from the data loss alone, and the decay is applied to the weights directly, after the adaptive scaling and outside it:

$$W \leftarrow W - \alpha \left(\frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon} + \lambda W \right). \tag{4.14}$$

The optimizer's estimate of the loss landscape is now a function of the data alone, while the decay does exactly what its name says to every weight at the same rate. Practically the change is small; in results it is not, and AdamW rather than Adam is the default for large models today, including the LLaMA family. The general lesson outlives the specific fix: a regularizer and an optimizer are not independent modules, and a term added to the loss is not necessarily the term that reaches the weights.

4.11. Learning rate schedules

Every optimizer above—SGD, momentum, RMSProp, Adam, AdamW—still takes α as an input, and none of them chooses it. Its effect is easy to see in the training curve, and Figure 4.3 shows the four characteristic shapes. Too large, and the loss rises rather than falls: the iterate is bouncing out of the valley described in Section 4.7. Too small, and the loss falls steadily but so slowly that the budget runs out before training does. In between sits a rate high enough to make quick early progress but too high to settle, where the loss drops sharply and then plateaus well above where it could have gone. The one to want falls fast at first and keeps improving afterwards.

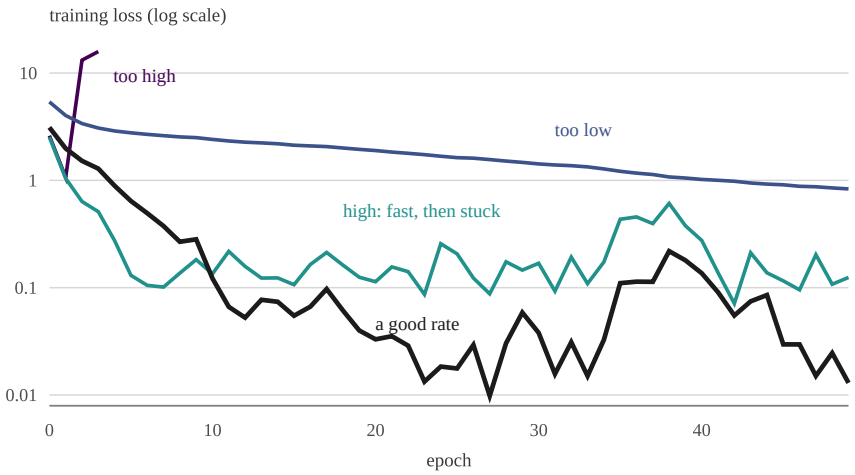


Figure 4.3.: Training loss against epoch at four learning rates on the same objective, with loss on a logarithmic axis. The largest rate diverges off the top within three epochs. The smallest descends steadily and is still an order of magnitude short when the budget ends. An intermediate rate falls fastest of all at first and then flattens, held up by the noise in its own steps. The remaining one descends past it and keeps going. *Drawn for these notes.*

The interesting observation is that the third and fourth curves are the same run seen at different times. A rate that is right for the first epoch, when the weights are random and any direction is an improvement, is too coarse for the last, when the iterate is near a minimum and needs to settle into it. Rather than compromise on one value, vary it: start high and reduce it as training proceeds. Every competitive training recipe in modern deep learning does this.

The simplest form is *step decay*—hold α fixed, then multiply it by a constant at a few chosen points. The schedule long used for ResNets on ImageNet multiplies by 0.1 after epochs 30, 60

and 90, and its signature is unmistakable in a loss curve, which drops abruptly at each cut as the finer step lets the optimizer descend where it had been stuck. Its drawback is that the cut points are three more hyperparameters.

Smooth schedules avoid that. *Cosine decay* sweeps the rate from its initial value down to zero along a half period of a cosine (Loshchilov and Hutter, 2017),

$$\alpha_t = \tfrac{1}{2}\,\alpha_0 \left(1 + \cos \frac{t\pi}{T}\right), \tag{4.15}$$

where α_0 is the initial rate, T the total number of epochs and t the current one. *Linear decay* does the same with a straight line, $\alpha_t = \alpha_0(1 - t/T)$, and *inverse square root decay* uses $\alpha_t = \alpha_0/\sqrt{t}$, the schedule that trained the original Transformer. Figure 4.4 puts them on one axis. Cosine and linear both need only α_0 and T, which is much of their appeal: they add one hyperparameter where step decay adds several, and T is usually fixed by the compute budget anyway.

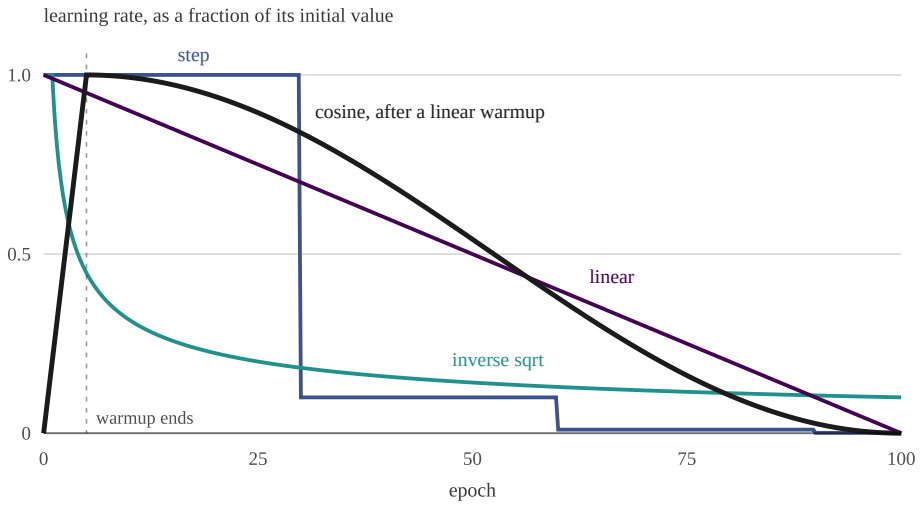


Figure 4.4.: Four decay schedules over a hundred epochs from the same initial rate, with a linear warmup applied to the cosine schedule over the first five. Cosine and linear reach zero at T; inverse square root never does. *Drawn for these notes.*

Decay handles the end of training; the beginning needs the opposite treatment. At initialization the weights are random and the first gradients can be large and badly aimed, so starting at full α risks the divergence in Figure 4.3 before the run has done anything. *Linear warmup* ramps the rate from zero to α_0 over the first few thousand iterations and only then hands over to the decay schedule. Warmup followed by cosine decay is one of the most widely used configurations in current practice.

Warmup matters most when the learning rate is large, and it becomes large for a reason worth stating. The *linear scaling rule* holds that multiplying the batch size by N should be accompanied by multiplying the learning rate by N (Goyal et al., 2017). The intuition is that a batch N times larger gives a gradient estimate whose noise is correspondingly smaller, so a proportionally longer step can be trusted; the justification is empirical rather than derived, and it holds over a wide but not unlimited range. Training at large batch sizes therefore means training at large learning rates, which is exactly where the first few steps become dangerous—and warmup was developed for that regime.

4.12. Second-order methods

Everything in this chapter is *first-order*: it uses the gradient and nothing else. Geometrically, that means approximating the loss near the current point by a plane and stepping downhill along it. A plane has no bottom, which is why a step size is needed at all—the linear approximation says which way to go and is silent on how far, so α supplies an answer that the mathematics does not.

Keeping the next term of the Taylor expansion removes that gap. Around a point W_0,

$$L(W) \approx L(W_0) + (W - W_0)^\top \nabla_W L + \tfrac{1}{2}(W - W_0)^\top H\,(W - W_0), \tag{4.16}$$

where H is the *Hessian*, the matrix of second partial derivatives $\partial^2 L / \partial W_i \partial W_j$. This approximation is a paraboloid, and a paraboloid does have a bottom. Setting the derivative to zero and solving gives the Newton update,

$$W \leftarrow W_0 - H^{-1} \nabla_W L, \tag{4.17}$$

which jumps directly to the minimum of the local quadratic model. There is no learning rate in Equation 4.17, and no schedule: the curvature that H encodes already says how far to go in each direction, which is the information the whole of Section 4.9 was trying to estimate from gradient histories.

The reason this is not how networks are trained is arithmetic. For N parameters, H has N^2 entries and inverting it costs $O(N^3)$. A modest network with a million parameters has a Hessian of 10^{12} entries, four terabytes in single precision, and modern models run to hundreds of millions or billions of parameters—for 10^8 parameters the Hessian would hold 10^{16} numbers, tens of petabytes, before anything is done with them. The matrix cannot be formed, let alone inverted, and the compute that forming it would consume buys far more if spent on additional gradient steps over more data.

Quasi-Newton methods reduce the cost without eliminating it. BFGS maintains an approximation to the inverse Hessian using rank-one updates, at $O(N^2)$ per step instead of $O(N^3)$; L-BFGS keeps only a short window of past updates and never forms the matrix at all. Both work well when the objective is deterministic and can be evaluated on the full dataset, which makes them a reasonable choice for small problems—but they transfer poorly to the minibatch setting, where the objective changes from step to step and the curvature information they accumulate is contaminated by sampling noise.

Which leaves the practical summary. Adam or AdamW is the right first thing to try on a new problem, and it often works acceptably even at a constant learning rate. SGD with momentum can beat it, at the cost of considerably more tuning of both the rate and the schedule, because it has no adaptive term to absorb a badly scaled direction. And if the dataset is small enough that full-batch updates are affordable, second-order methods become worth considering—with every source of stochasticity switched off first.

4.13. Where the linear model stops

The whole of this chapter has optimized $f(x, W) = Wx$, and optimized it as well as it can be optimized. That is worth stating plainly, because it isolates what comes next: no schedule, no adaptive rate and no amount of regularization can make a linear model represent something a linear model cannot represent.

The standard demonstration is two classes arranged in concentric rings—one colour in the middle, the other surrounding it. No straight line separates them, so no setting of W classifies them correctly, and the optimizer converges perfectly well to a solution that is perfectly useless. But rewrite each point in polar coordinates, replacing (x, y) by radius and angle, and the rings become two horizontal bands that a single line separates easily. The data were always separable; the representation was wrong.

That suggests learning the transformation rather than choosing it. A two-layer network does exactly this, composing two linear maps with a non-linearity between them:

$$f(x, W_1, W_2) = W_2 \max(0, W_1 x). \tag{4.18}$$

The elementwise $\max(0, \cdot)$ is essential and not incidental. Without it the composition $W_2 W_1 x$ is a single linear map with a more complicated factorization, no more expressive than what we started with. With it, the first layer learns a representation and the second classifies in that representation, and both are fitted by the same gradient descent developed above. The name "neural network" is broad enough to be unhelpful here; models of this specific shape are more precisely called fully connected networks, or multi-layer perceptrons.

Only one thing is missing. Every optimizer in this chapter consumes $\nabla_W L$, and Equation 4.18 has two weight matrices with a non-linearity between them, so the closed-form differentiation of Section 4.5 no longer suffices. Computing gradients through arbitrary compositions of functions—efficiently, and without deriving each one by hand—is backpropagation, and it is the subject of the next chapter.

References

- J. Duchi, E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, 12, 2011. Paper
- T. Tieleman and G. Hinton, "Lecture 6.5 — RMSProp," *Neural Networks for Machine Learning*, Coursera, 2012.
- I. Sutskever, J. Martens, G. Dahl and G. Hinton, "On the Importance of Initialization and Momentum in Deep Learning," *ICML*, 2013. Proceedings
- Y. Dauphin et al., "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization," *NIPS*, 2014. arXiv:1406.2572
- D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *ICLR*, 2015. arXiv:1412.6980
- I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," *ICLR*, 2017. arXiv:1608.03983
- P. Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," 2017. arXiv:1706.02677
- A. Vaswani et al., "Attention Is All You Need," *NIPS*, 2017. arXiv:1706.03762
- I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," *ICLR*, 2019. arXiv:1711.05101

5. Neural Networks and Backpropagation

Lecture 4

Based on Lecture 4 of CS231n, Stanford University, Spring 2025.

5.1. What the hidden layer buys

Equation 4.18 arrived at the end of the previous chapter as a repair. The linear model could not represent what the data required, so the proposal was to learn the representation rather than choose it. That says what the extra layer is for without saying what it does, and the difference is worth making concrete, because it is visible in the shapes alone.

Take CIFAR-10, where an image is $D = 3072$ numbers and there are $C = 10$ classes. The linear model of Equation 3.4 has a single matrix of shape 10×3072. The two-layer model splits that into W_1 of shape 100×3072 and W_2 of shape 10×100, with the hidden vector $h = \max(0, W_1 x)$ carrying a hundred numbers between them. Nothing has been added to the input and nothing has been added to the output; the model has simply been made to pass its evidence through a hundred-dimensional bottleneck of its own design.

The consequence follows from what a row of a weight matrix can do. As Section 3.4 put it, each row of W scores an image by correlating it with a fixed pattern, so the ten rows are ten templates and the classification is a contest between them. A single row for *horse* has to respond to every horse in the training set. Horses photographed facing left put bright pixels where horses facing right put background, and one row that scores both of them highly is a compromise between the two—which is why the learned template, viewed as an image, tends to show an animal with a head at each end. That is not a failure of the optimizer. It is the best that one template can do when the class it describes appears in incompatible configurations.

The hundred rows of W_1 are under no such obligation, because none of them is required to correspond to a class. One row is free to respond to a left-facing horse and another to a right-facing one, and the *horse* row of W_2 can weight both positively, so the two poses are recognised separately and then added rather than averaged into one. The same freedom cuts the other way: a hidden unit that responds to a dark round region surrounded by lighter pixels is useful to bird, cat, deer, dog and horse at once, and every one of those output rows can draw on it. The hidden layer holds features that classes share; the output layer holds the part that is specific to a class. A hundred templates instead of ten, and reuse between them, is the entire structural gain.

Depth extends the same move rather than introducing a new one. A three-layer network,

$$f(x, W_1, W_2, W_3) = W_3 \max(0, W_2 \max(0, W_1 x)), \tag{5.1}$$

lets the second hidden layer build its features from the first layer's features instead of from raw pixels, so that a unit can respond to a configuration of parts rather than to a pattern of pixels. The dimensions have to agree along the chain and nothing else constrains them. In practice each layer also carries a learnable bias, written $W_1 x + b_1$, which is omitted here only to keep the composition readable.

The naming convention is a persistent source of confusion and is worth stating once. A network is counted by its weight matrices, not by its non-linearities: Equation 4.18 is a two-layer network with one hidden layer, and Equation 5.1 is a three-layer network with two hidden layers. Layers of this kind, in which every input to a layer influences every output of it, are called fully connected layers, and networks built from them are fully connected networks or multi-layer perceptrons—both more precise than "neural network", which names a family far larger than this.

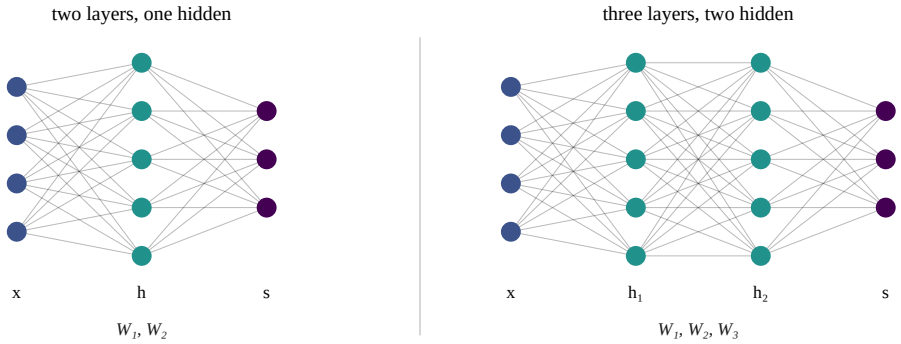


Figure 5.1.: Two fully connected networks drawn as graphs. Left: one hidden layer, so two weight matrices and a two-layer network. Right: two hidden layers and three weight matrices. Every unit in a layer is connected to every unit in the next, which is what "fully connected" names; the arrows carry the weights and the layers carry the activations. *Drawn for these notes.*

5.2. What the max is doing

The previous chapter observed that removing the max from Equation 4.18 collapses $W_2 W_1$ into a single matrix and returns the model to a linear classifier. That is a proof that the non-linearity is necessary. It is not an account of what the non-linearity contributes, and the account turns out to be simple enough to see in one dimension.

Consider a single hidden unit reading a scalar input, computing $\max(0, wx + b)$. Its output is zero for every x on one side of the point $x = -b/w$ and rises linearly on the other. The unit is two straight pieces joined at a corner, and the position of that corner is set by b/w—which is to say, by the parameters, which is to say, by training. A weighted sum of H such units is again made of straight pieces, with at most H corners, each placed where one unit switches on.

This is what the network's capacity to fit a curve consists of. A two-layer network with a ReLU non-linearity computes a piecewise-linear function, gradient descent adjusts both the slopes of the pieces and the locations of the joints between them, and the number of joints available is bounded by the width of the hidden layer. Fitting a smooth function means approximating it by a polyline whose vertices the optimizer is free to move, which is why width buys accuracy and why the approximation is exact nowhere and adequate everywhere.

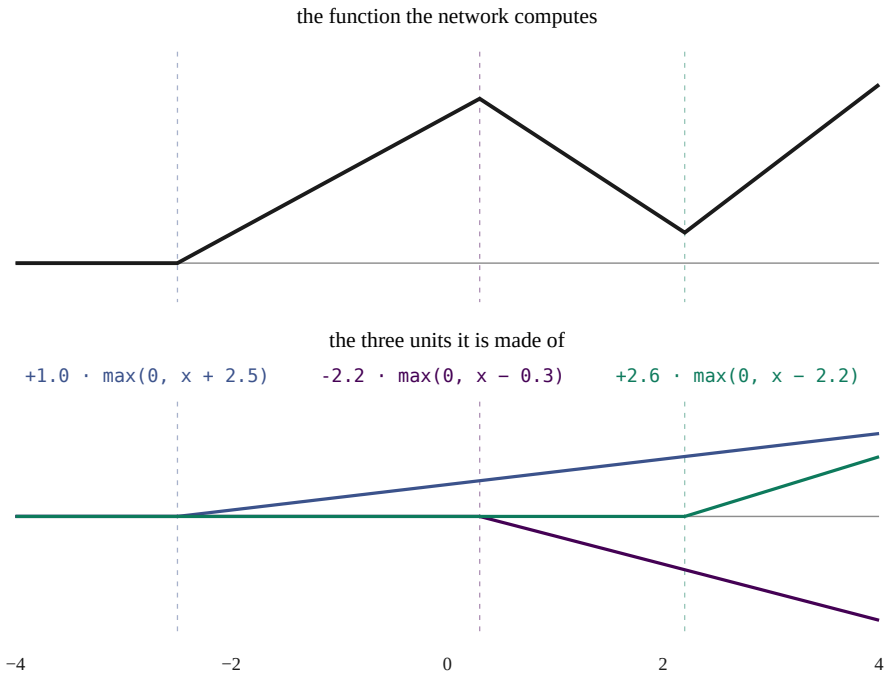


Figure 5.2.: A weighted sum of three ReLU units, each contributing one corner. The three units are drawn beneath the axis they act on, with the location of each corner set by that unit's own weight and bias; their weighted sum, above, is the piecewise-linear function the network computes. Adding units adds corners, which is the sense in which width buys the ability to fit a curve. *Drawn for these notes.*

In D dimensions the picture is the same with the corners replaced by hyperplanes. Each hidden unit divides input space in two along the plane where its pre-activation crosses zero, the units together cut the space into regions, and within any one region the set of active units is fixed, so the network is exactly affine there. The network is a collection of linear models with a learned rule for choosing between them. Depth matters here in a way width does not, because a second hidden layer folds the regions produced by the first, and the number of distinct regions available grows much faster with depth than with width (Montúfar et al., 2014).

One detail of this construction matters later. A piecewise-linear function is not differentiable at its corners, so $\max(0, z)$ has no derivative at $z = 0$, and every framework simply picks a value there—PyTorch returns zero. The choice is arbitrary and inconsequential, because the set of inputs landing exactly on a corner has measure zero, but it is worth knowing that it was made rather than derived.

5.3. Choosing an activation

The rectified linear unit, $\max(0, z)$, is the default and deserves to be. It costs one comparison, it does not saturate for positive inputs, and its derivative is exactly one wherever it is active, so a gradient passing through many layers of ReLU is neither amplified nor attenuated by the activation itself. Every alternative below is an attempt to fix one specific complaint about it, and the complaints are worth understanding because the same failure modes recur throughout the course.

The complaint against ReLU is that a unit can die. If a unit's pre-activation is negative for every example in the training set, its output is zero for all of them, its derivative is zero for all of them, and no gradient reaches its weights ever again. It is not merely inactive but unrecoverable, and a network can lose a substantial fraction of its units this way, typically to a learning rate large enough to push a whole row of W_1 off the data in one step. Leaky ReLU replaces the flat half with a shallow slope,

$$\text{LeakyReLU}(z) = \max(\alpha z, z), \qquad \alpha = 0.01, \tag{5.2}$$

so that a unit with negative pre-activation still passes a small gradient and can be pushed back. The exponential linear unit does the same with a smooth curve that approaches $-\alpha$ instead of falling away linearly, which also brings the mean output nearer zero.

The smooth variants matter for a reason that only shows up in deeper models. The Gaussian error linear unit multiplies the input by the probability that a standard normal variable falls below it,

$$\text{GELU}(z) = z\,\Phi(z), \tag{5.3}$$

giving a function that behaves like ReLU far from the origin but has a continuous derivative and a small negative dip just below zero (Hendrycks and Gimpel, 2016). The sigmoid linear unit, $z\,\sigma(z)$, has the same shape from a different construction. These are the activations used in transformers and in several convolutional families, and the argument for them is empirical rather than principled: the loss surface they produce is easier to descend, and the gain is real but small.

Sigmoid and tanh predate all of these and are now the wrong choice inside a network, for a reason that is quantitative rather than aesthetic. The derivative of $\sigma(z) = 1/(1+e^{-z})$ is $\sigma(z)\,(1-\sigma(z))$, which attains its maximum of 0.25 at $z = 0$ and decays towards zero in both directions. A gradient crossing a sigmoid is therefore multiplied by at most a quarter even under the most favourable conditions, and by very much less once the unit is anywhere near saturation, so ten such layers attenuate the gradient by a factor of at least $4^{10} \approx 10^6$ before it reaches the first weight matrix. This is the vanishing gradient problem in its simplest form, and it is why networks of this depth were not trainable before rectifiers.

Sigmoid carries a second, subtler defect: its output is always positive. Every input to the next layer is therefore positive, and by the local-gradient rule developed in Section 5.10, that forces every element of the gradient with respect to a row of the next weight matrix to carry the same sign. The row can only move up in all coordinates or down in all coordinates, so a direction requiring one weight to rise and another to fall must be reached by a zigzag—exactly the poorly conditioned descent of Section 4.7, produced here by the activation rather than by the data. Tanh, being $2\sigma(2z) - 1$, fixes the centring and raises the maximum derivative to one, which makes it strictly better than sigmoid as a hidden activation while leaving the saturation intact.

Both still have a use at the output, where saturation is the point rather than the problem: a sigmoid maps a score to a probability for a binary decision, and tanh maps one to a bounded quantity. Inside the network, the practical advice is what the empirical record supports. Use ReLU unless the architecture family you are working in has an established alternative, in which case use that one, and treat the choice as a hyperparameter to be validated rather than a question with a general answer.

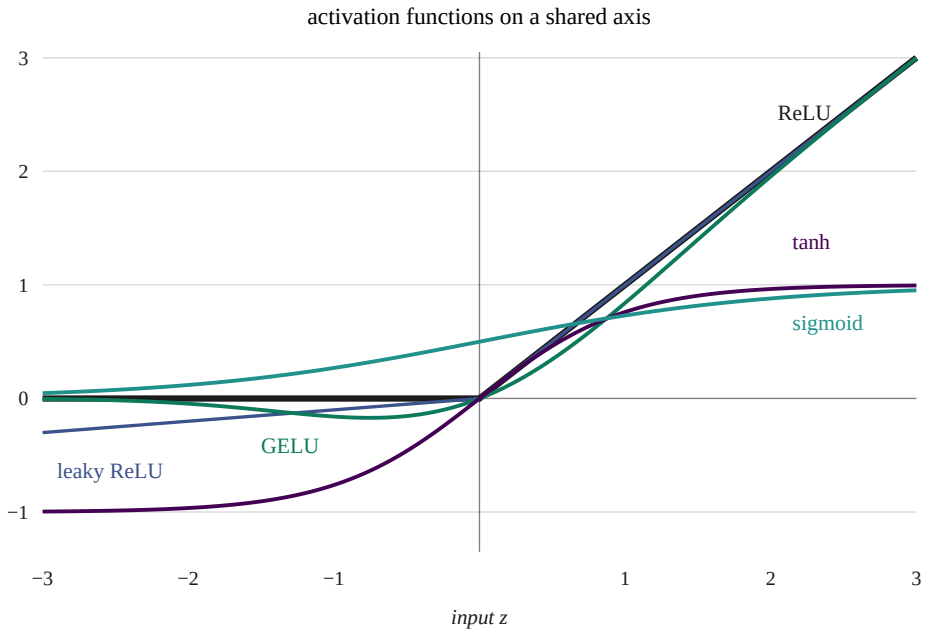


Figure 5.3.: Five activation functions on shared axes. ReLU and leaky ReLU coincide exactly for positive inputs and differ only for negative ones; leaky ReLU is drawn here with $\alpha = 0.1$ rather than the 0.01 of Equation 5.2, because at 0.01 the negative slope is under half a pixel across the whole plot. GELU tracks ReLU away from the origin but is smooth through it, dipping slightly below zero. Sigmoid and tanh are bounded, and both flatten at each end—the regions where a gradient arriving from above is destroyed. *Drawn for these notes.*

5.4. Depth, width and the forward pass

Evaluating Equation 4.18 is a matrix product, an elementwise maximum, and a second matrix product. Nothing in it is expensive to state and nothing in it is expensive to run, which is the reason this particular model became the default unit of construction rather than any of the more biologically motivated alternatives: it is exactly the operation that a GPU executes fastest. The regularity of a fully connected layer—every input reaching every output, in one dense matrix—is a decision about hardware as much as about modelling.

The forward pass over a batch is worth writing with its shapes attached, because in an implementation the shapes are what you debug. With N examples stacked as rows, the input is a matrix x of shape $N \times D$, the first weight matrix is $D \times H$, and the hidden activations $h = \max(0, xW_1)$ come out $N \times H$; the second weight matrix is $H \times C$ and the scores are $N \times C$. Every example travels through independently and the batch dimension is carried along untouched, which is why a batch of any size works without changing a line.

Training the whole thing is that forward pass, the gradient of the loss with respect to each weight matrix, and the update from Equation 4.5. In a numerical library it fits in twenty lines with nothing hidden:

```
import numpy as np

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = np.random.randn(N, D_in), np.random.randn(N, D_out)
w1 = np.random.randn(D_in, H) / np.sqrt(D_in)
w2 = np.random.randn(H, D_out) / np.sqrt(H)

for t in range(2000):
    h = np.maximum(0, x.dot(w1))          # forward: hidden activations
    y_pred = h.dot(w2)                    # forward: scores
    loss = np.square(y_pred - y).sum()    # forward: squared error

    grad_y_pred = 2.0 * (y_pred - y)      # backward: through the loss
    grad_w2 = h.T.dot(grad_y_pred)
    grad_h = grad_y_pred.dot(w2.T)
    grad_h[h <= 0] = 0                    # backward: through the ReLU
    grad_w1 = x.T.dot(grad_h)

    w1 -= 1e-4 * grad_w1                  # gradient descent
    w2 -= 1e-4 * grad_w2
```

Two details in that listing are not decoration. The weight matrices are divided by the square root of their input dimension, because drawing them from a unit normal would give a pre-activation with standard deviation $\sqrt{1000} \approx 31.6$ and a first loss large enough to diverge at any usable learning rate; initialization is a subject in its own right and this is the crudest form of the fix. And the line `grad_h[h <= 0] = 0` is the backward pass of the max—a gradient reaches a hidden unit only where that unit was active going forwards. Everything else is the chain rule applied by hand, which is precisely the thing that will not survive the next section.

Run it and the loss falls from 862 to under 10^{-3} within two hundred steps and to zero shortly after. That is not a good result; it is the point of the next section. The targets `y` are random numbers, so there is nothing to learn, and a hundred hidden units have simply memorised sixty-four arbitrary vectors.

5.5. Size is not a regularizer

More units mean more capacity, and more capacity means a decision boundary that can wander further to accommodate individual training points. The shape of the failure is familiar from Section 3.2: a one-nearest-neighbour classifier carves an island around every noisy example, and a wide network does the same thing with a smooth boundary instead of a Voronoi cell. Faced with a model that fits the training set and fails on held-out data, the obvious repair is to build a smaller one.

The advice is to resist it, and the reason is not that smaller networks generalise badly. It is that width and regularization strength are not interchangeable knobs even when they move the same quantity. The penalty weight λ of Equation 4.1 is continuous, so a validation sweep over it explores capacity finely; the width of a layer is an integer that changes the model's architecture, its initialization, its memory footprint and the shape of its loss surface all at once, and a sweep over it is a sweep over different training runs rather than different settings of one run. When a change hurts, λ tells you by how much and in which direction. Width tells you much less, at a much higher cost per sample.

The working procedure that follows from this is to grow the network until it is capable of overfitting, and only then to regularize. Overfitting is evidence that the model has enough capacity to represent the structure in the data, which is a prerequisite for learning it; a model that cannot overfit its training set is not being cautious, it is being inadequate, and no amount of tuning λ will recover what it cannot represent. Once the model can overfit, λ is the instrument that decides how much of that capacity is spent on structure and how much on noise.

That instrument has two ends. Too little regularization gives the boundary in the previous paragraph. Too much constrains the weights so severely that the boundary loses the detail it needed, and the model underfits—the classifier is now failing for the opposite reason and the validation curve looks similar from a distance. Neither end is a subtle effect and both are visible in a sweep, which is exactly the argument for sweeping the quantity that is cheap to sweep.

None of this settles how wide to make the network in the first place, and there is no principled answer. The practice is to start from a width that has worked for similar data and a similar problem, and to explore around it. Recognising which established architecture the problem resembles is a more useful skill than any formula for choosing a layer size, and most of the rest of this course is a tour of those architectures.

5.6. The brain analogy and its limits

The vocabulary comes from biology and the resemblance is real but shallow. A biological neuron collects electrical impulses from many upstream cells through its dendrites, integrates them in the cell body, and emits a signal down its axon when the accumulated input is sufficient. Written as arithmetic, that is a weighted sum followed by a threshold, which is one unit of Equation 4.18 with the activation function playing the part of the threshold. The analogy earned the field its name and is a fair description of where the idea came from.

It is a poor description of what either system is. A cortical neuron is not a scalar rate: it has spike timing, dozens of morphological types, dendrites that themselves compute non-linearly, and neuromodulatory state that changes its behaviour wholesale. Nothing in the brain corresponds to a global scalar loss differentiated back through every synapse, and the layered, densely connected regularity of the models in this chapter was chosen because it maps onto a matrix multiply, not because it maps onto cortex. Networks with irregular connectivity have been built and studied;

they perform about as well and are much harder to run, which is the honest reason they are not used. Take the analogy as etymology rather than as evidence.

5.7. Why gradients need a system

Return to the four gradient lines in Section 5.4. They are correct, and they were obtained by differentiating a specific composition—squared error, on top of a linear map, on top of a rectifier, on top of a linear map—once, by hand, on paper. Every part of that sentence is a liability.

Change the loss from squared error to the softmax objective of Equation 3.7 and `grad_y_pred` is wrong and every line below it inherits the error. Insert a third layer and the derivation restarts. Replace the rectifier and the masking line changes shape. The work is proportional to the architecture, it has to be redone whenever any part of the architecture moves, and it is done in the one place where a mistake is hardest to notice—a gradient that is wrong by a constant factor still trains, slowly and to a worse optimum, and looks like a badly chosen learning rate. Chapter 3's numerical gradient check exists precisely because this failure is silent.

Tedium is the mild version of the objection. The severe one is that hand differentiation does not scale past architectures you can hold in your head. A recurrent model unrolled over a hundred timesteps, or a network with branches that merge, or anything with a memory that is read and written—the differentiation is not tedious for these, it is infeasible, and the field would have stopped at two layers if it had depended on doing it.

The fix is to stop treating the model as one large formula. Write it instead as a graph in which each node performs a single primitive operation—an addition, a multiplication, a maximum, an exponential—and each edge carries a value from the node that produced it to the nodes that consume it. The inputs to the graph are the data and the parameters; the single output is the scalar loss. Nothing about the model has changed; it has only been decomposed to a granularity at which every individual derivative is something you already know.

What makes the decomposition pay is that the chain rule is local. To send a gradient through a node, that node needs the derivative of its own output with respect to its own inputs, and the gradient of the loss with respect to its output. It needs to know nothing whatsoever about the rest of the graph—not what produced its inputs, not what consumes its outputs, not what loss is being minimised. So if every primitive knows how to differentiate itself, the gradient with respect to every parameter in the graph can be assembled by traversing the graph backwards, once. That procedure is backpropagation.

5.8. The chain rule on a graph

The procedure is easiest to trust after watching it work on something small enough to check by hand. Take

$$f(x, y, z) = (x + y)\, z, \tag{5.4}$$

evaluated at $x = -2$, $y = 5$, $z = -4$. As a graph it has two nodes: an addition consuming x and y and producing an intermediate value $q = x + y$, and a multiplication consuming q and z and producing f. The forward pass evaluates them in order, giving $q = 3$ and $f = -12$, and each node keeps its inputs, because the backward pass will need them.

What is wanted is $\partial f/\partial x$, $\partial f/\partial y$ and $\partial f/\partial z$. Two of the four local derivatives are immediate from the addition node, since $q = x + y$ gives $\partial q/\partial x = 1$ and $\partial q/\partial y = 1$. The multiplication node gives the other two: $f = qz$ gives $\partial f/\partial q = z$ and $\partial f/\partial z = q$. Note what those say—the derivative of a product with respect to one factor is the *other* factor, which is why a multiplication node has to remember both of its inputs.

Now traverse backwards. The gradient of f with respect to itself is 1. At the multiplication node, the two inputs receive $\partial f/\partial z = q = 3$ and $\partial f/\partial q = z = -4$. At the addition node, the chain rule composes what has arrived from above with what the node contributes locally:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\,\frac{\partial q}{\partial x} = (-4)(1) = -4, \tag{5.5}$$

and identically $\partial f/\partial y = -4$. The results are checkable without any of this machinery: raising x from -2 to -1.99 raises q to 3.01 and lowers f to -12.04, a change of -0.04 for an input change of 0.01, which is the -4 just computed.

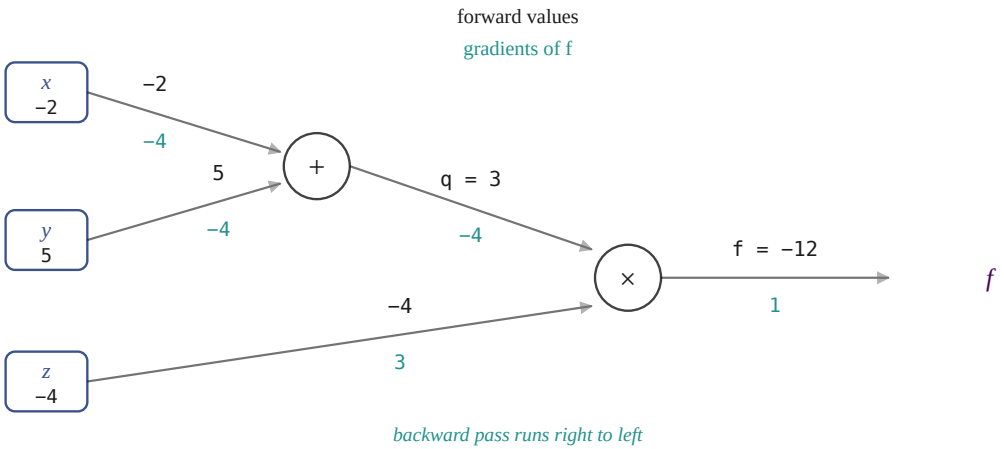


Figure 5.4.: The graph of Equation 5.4 evaluated at $x = -2$, $y = 5$, $z = -4$. Values computed on the forward pass are shown above each edge and gradients of f with respect to that edge below it, in the second colour. The backward pass starts from $\partial f/\partial f = 1$ at the right and moves left, each node multiplying what arrives from above by its own local derivative. *Drawn for these notes.*

Equation 5.5 is the entire algorithm, and it is worth naming its two factors, because the names are how the rest of this chapter and most framework source code talk about it. The quantity arriving from the node above is the **upstream gradient**: the derivative of the loss with respect to this node's output. The derivative the node computes from its own inputs is the **local gradient**. Their product is the **downstream gradient**, which the node passes to whatever produced its inputs, where it becomes that node's upstream gradient in turn.

The recursion terminates because the graph is finite, and it visits each node exactly once, so the whole backward pass costs about what the forward pass costs. This is the property that makes the method viable rather than merely correct. A network with 10^8 parameters yields all 10^8 partial derivatives in one traversal, whereas the finite-difference estimate of Section 4.5 would need 10^8 separate forward passes to get the same information. The asymmetry is not an implementation detail; it is the reason gradient-based learning at this scale exists at all, and it was the point of the paper that introduced the technique to the field (Rumelhart, Hinton and Williams, 1986).

Two things a node emphatically does not need are worth stating, because they are what make the whole scheme composable. It does not need to know what produced its inputs, and it does not need to know what its output feeds into or what loss is eventually computed. It needs its own inputs, its own local derivative, and one number arriving from above.

5.9. A longer example, and a choice about granularity

Take a logistic unit over two inputs,

$$f(w, x) = \frac{1}{1 + \exp(-(w_0 x_0 + w_1 x_1 + w_2))}, \tag{5.6}$$

at $w_0 = 2$, $x_0 = -1$, $w_1 = -3$, $x_1 = -2$, $w_2 = -3$. Decomposed to primitives the graph runs: two multiplications giving -2 and 6; an addition giving 4; a second addition with w_2 giving 1; a negation giving -1; an exponential giving 0.37; an addition of one giving 1.37; and a reciprocal giving 0.73.

The backward pass walks the same chain in reverse, each step multiplying by one local derivative. The reciprocal $1/u$ has local derivative $-1/u^2 = -1/1.37^2 = -0.53$, so the gradient below it is -0.53. Adding a constant has local derivative 1 and changes nothing. The exponential has local derivative equal to its own output, 0.37, giving $-0.53 \times 0.37 = -0.20$. The negation flips the sign to $+0.20$. From there the two additions distribute 0.20 unchanged to w_2 and to both products, and each multiplication hands each of its inputs the value of the other scaled by 0.20: $\partial f/\partial w_0 = x_0 \times 0.20 = -0.20$ and $\partial f/\partial x_0 = w_0 \times 0.20 = 0.40$, and likewise -0.40 and -0.60 for the second pair.

Eight nodes, eight local derivatives, no step harder than a first-year exercise. But four of those nodes—negate, exponentiate, add one, reciprocate—exist only to compute the logistic function, and the logistic function differentiates in closed form. Writing $\sigma(u) = 1/(1 + e^{-u})$,

$$\frac{d\sigma}{du} = \frac{e^{-u}}{(1 + e^{-u})^2} = \left(\frac{1 + e^{-u} - 1}{1 + e^{-u}}\right)\left(\frac{1}{1 + e^{-u}}\right) = (1 - \sigma(u))\,\sigma(u), \tag{5.7}$$

which at $\sigma = 0.73$ gives $(1 - 0.73)(0.73) = 0.20$—the same number the four-node chain produced, from one multiplication.

So the decomposition into primitives is a choice, not a fact about the function, and the choice is a real engineering trade. A coarse node is faster, because it replaces four kernel launches with one and four cached intermediate tensors with none. It is often numerically better, because a closed form can avoid the cancellations that its own decomposition runs into. And Equation 5.7 has a property worth noticing on its own: the derivative is expressed in terms of the node's *output*, so a sigmoid node need not remember its input at all. Against that, every coarse node is a derivative someone has to derive and test by hand, which is the cost the whole method was introduced to avoid. Frameworks resolve this empirically, by writing fused nodes for the operations that are common enough to be worth the effort and leaving everything else composed from primitives.

5.10. Four patterns worth recognising

Because the local derivatives of the common primitives are so simple, a backward pass can usually be read off a graph rather than computed. Four patterns cover most of what appears in practice.

An **addition distributes**. Since $\partial(a+b)/\partial a = 1$, the upstream gradient passes to both inputs unchanged: a node adding 3 and 4 to make 7, receiving an upstream gradient of 2, sends 2 to each input regardless of their values. Addition nodes do not need to cache anything.

A **multiplication swaps**. Each input receives the upstream gradient scaled by the *other* input: a node multiplying 2 and 3 to make 6, receiving upstream 5, sends $3 \times 5 = 15$ to the first and $2 \times 5 = 10$ to the second. The consequence is worth holding onto, because it explains a failure mode. If one input to a multiplication is very large, the gradient on the other is scaled by it, so poorly scaled inputs produce poorly scaled gradients everywhere downstream, which is much of why normalizing inputs matters.

A **maximum routes**. Only the winning input affected the output, so it receives the whole upstream gradient and the others receive zero: $\max(4, 5) = 5$ with upstream 9 sends 9 along the branch carrying 5 and 0 along the branch carrying 4. This is the pattern behind `grad_h[h <= 0] = 0` in Section 5.4, and behind the dead units of Section 5.3—a unit that always loses the maximum is a branch the gradient never travels down.

A **copy adds**. When one value is consumed by several nodes, the graph has a branch, and each consumer sends back its own gradient for that value. They are summed: a value used twice, receiving 4 from one consumer and 2 from the other, has total gradient 6. This follows from the multivariable chain rule—the value influences the loss along every path, and the influences add—and it is not a special case but the general rule, of which the single-consumer situation is the degenerate instance. Everything downstream depends on it: a weight matrix reused across the timesteps of a recurrent network is a copy node with as many consumers as timesteps, and its gradient is the sum over all of them.

5.11. Forward and backward as an interface

The locality established in Section 5.8 is what turns backpropagation from a procedure into a software architecture. If a node needs only its own inputs and one upstream gradient, then a node can be an object with two methods, and a framework can be a library of such objects plus a traversal:

```
class Multiply:
    def forward(self, x, y):
        self.x, self.y = x, y     # cache what backward will need
        return x * y

    def backward(self, grad_z):
        grad_x = self.y * grad_z  # upstream gradient x local gradient
        grad_y = self.x * grad_z
        return grad_x, grad_y
```

The cache is the whole of the contract between the two methods, and deciding what belongs in it is the one design decision each operator makes. A multiplication caches both inputs, because

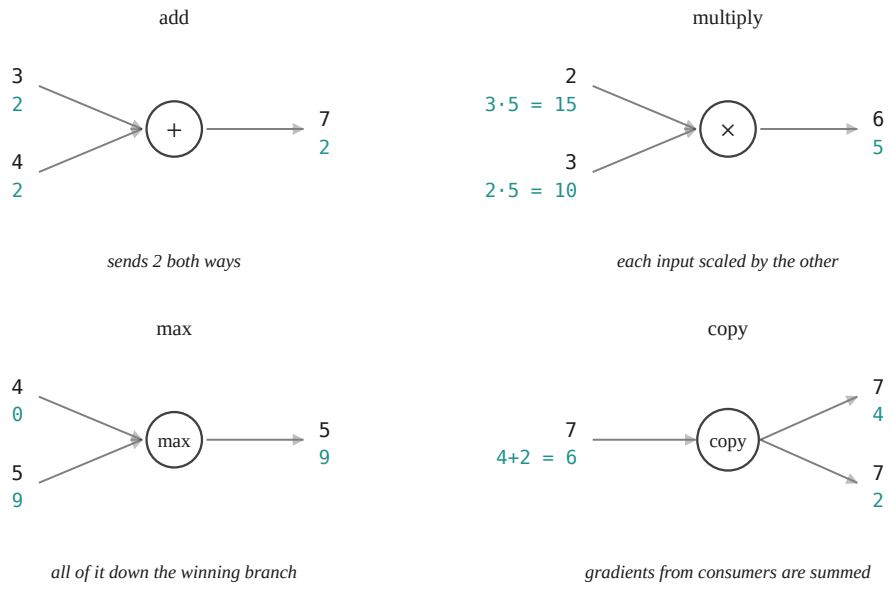


Figure 5.5.: The four patterns, each shown with forward values above the wire and gradients below. Addition sends the upstream gradient to both inputs unchanged. Multiplication sends each input the upstream gradient times the other input. Maximum sends everything down the winning branch and nothing down the others. A copy sums the gradients returning from its consumers. *Drawn for these notes.*

each local gradient is the other one. An addition caches nothing. A sigmoid, by Equation 5.7, caches its *output*. The choice is not cosmetic at scale: activations saved for the backward pass are usually the largest consumer of memory during training, larger than the parameters, and the reason a model that fits in memory for inference may not fit for training.

Nothing in this depends on the node being scalar, and nothing depends on the graph being known in advance. Both facts matter. PyTorch builds the graph as the forward pass runs, recording each operation and its cached tensors as it goes, so control flow in the host language—a loop whose length depends on the data, a branch taken on a value—is captured for free, and the backward pass traverses whatever was actually executed (Paszke et al., 2019). The general technique is reverse-mode automatic differentiation, and it long predates its use in machine learning (Baydin et al., 2018).

Its production operators are the interface above with the loop pushed into a compiled kernel. PyTorch's sigmoid backward computes `grad * output * (1 - output)`, using the saved output exactly as Equation 5.7 says it can—the fused node from the previous section, in the library, for the reason given there.

5.12. Vectors, and the Jacobian you do not build

Everything so far has been scalar, and the operations in an actual network are not. Three kinds of derivative appear, and keeping them apart is most of the difficulty.

For a scalar function of a scalar, $\partial y/\partial x$ is one number answering one question: if x moves a little, how much does y move? For a scalar function of a vector $x \in \mathbb{R}^D$, the answer is one number per component of x, assembled into a gradient $\partial y/\partial x \in \mathbb{R}^D$. For a vector function of a vector,

$y \in \mathbb{R}^M$ and $x \in \mathbb{R}^D$, every output can respond to every input, so the answer is a matrix—the **Jacobian** $\partial y/\partial x \in \mathbb{R}^{M \times D}$, whose entry (i, j) is $\partial y_i/\partial x_j$.

One asymmetry runs through the whole subject: the loss is always a scalar, and never anything else. It has to be, because gradient descent needs a single quantity to descend and there is no ordering on vectors to descend with respect to. Everything inside the network is a vector or a matrix or a tensor; the last node collapses all of it to one number. So while intermediate nodes have Jacobians, the object propagated backwards is always the gradient of a *scalar* with respect to something, and therefore always has exactly the shape of the thing it is a gradient of. A gradient with respect to a 64×4096 activation is itself 64×4096. This is the single most useful check available when writing a backward pass by hand.

With that convention fixed, the vector chain rule reads

$$\frac{\partial L}{\partial x} = \left(\frac{\partial y}{\partial x}\right)^{\top} \frac{\partial L}{\partial y}, \tag{5.8}$$

whose shapes are $[D \times M][M] = [D]$, as required. The Jacobian transpose is the local gradient and $\partial L/\partial y$ is the upstream gradient, so Equation 5.8 is Equation 5.5 with the multiplication promoted to a matrix–vector product.

Written that way it looks ruinous, and taken literally it would be. Consider an elementwise ReLU on a vector of four numbers, $x = [1, -2, 3, -1]$ giving $y = [1, 0, 3, 0]$. Its Jacobian is 4×4, but because each output depends only on the input in the same position, every off-diagonal entry is zero, and the diagonal holds 1 where the input was positive and 0 where it was not. Applying Equation 5.8 to an upstream gradient of $[4, -1, 5, 9]$ gives $[4, 0, 5, 0]$.

That result required no matrix. It required looking at the upstream gradient and zeroing the entries where the input was negative, which is exactly the line of code in Section 5.4. Forming the Jacobian would mean allocating D^2 numbers to store D meaningful ones—for a realistic $D = 4096$, sixteen million entries of which four thousand are non-zero—and then spending a matrix–vector product to apply what is a mask. No implementation does this. **Equation 5.8 states what the backward pass computes; it does not describe how.** Every operator implements the product of its Jacobian with a vector directly, and the Jacobian itself is never a thing that exists in memory.

5.13. Matrices, and the same trick at scale

The operation that actually dominates a network is the one Section 5.4 wrote as a matrix product: $y = xW$ with x of shape $N \times D$, W of shape $D \times M$, and y of shape $N \times M$. Here the gap between the formal statement and the implementation becomes absurd rather than merely wasteful.

Take the shapes from a realistic layer, $N = 64$ and $D = M = 4096$. The Jacobian $\partial y/\partial x$ relates every one of the $N \times D = 262{,}144$ entries of x to every one of the $N \times M = 262{,}144$ entries of y, so it has $262{,}144^2 \approx 6.9 \times 10^{10}$ entries, which at four bytes each is 256 gibibytes—for one layer, for one minibatch, for a quantity that exists only as an intermediate step. Nothing about the problem is that large: the input is four megabytes and the answer is four megabytes.

The way through is to write down one entry of the result and read the structure off it. Since $y_{n,m} = \sum_d x_{n,d} W_{d,m}$, the entry $x_{n,d}$ influences only row n of y, and influences $y_{n,m}$ with local derivative $W_{d,m}$. Summing its contributions along every path, as the copy rule of Section 5.10 requires,

$$\frac{\partial L}{\partial x_{n,d}} = \sum_{m} \frac{\partial L}{\partial y_{n,m}} W_{d,m}, \tag{5.9}$$

and that sum over m of an $[N \times M]$ quantity against a $[D \times M]$ quantity is one entry of a matrix product with W transposed. The same argument for W, where $W_{d,m}$ influences column m of every row, gives the companion result. Together:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^{\top}, \qquad \frac{\partial L}{\partial W} = x^{\top} \frac{\partial L}{\partial y}. \tag{5.10}$$

Two matrix products, each about the cost of the forward pass, against the 256 gibibytes that the literal reading demanded. The backward pass of a fully connected layer is the same primitive as its forward pass, which is why training costs roughly three times inference rather than thousands of times.

These are the two lines `grad_h = grad_y_pred.dot(w2.T)` and `grad_w2 = h.T.dot(grad_y_pred)` from Section 5.4, now derived rather than asserted. And they suggest a practical shortcut that is worth using and worth distrusting: given the upstream gradient and the cached inputs, there is usually only one arrangement of transposes and products whose shapes come out right, so matching shapes will often find the answer. It is an excellent check. It is not a derivation—it cannot distinguish a correct expression from a wrong one that happens to have the same shape, and it is silent about signs. Use it to catch mistakes, then verify against a numerical gradient.

5.14. What this makes possible

Nothing in the rest of this course adds a new principle to the one in Equation 5.5. Convolutional layers, attention, normalization layers, the residual connection, every loss function and every architecture through to diffusion models and vision–language models: each is a different graph assembled from primitives, each primitive supplying a forward rule and a local derivative, and each trained by the same backward traversal and the same optimizers from the previous chapter. The reason the field could move as fast as it did after 2012 is that changing the model stopped requiring changing the mathematics.

What changes from here is which primitives are worth having. A fully connected layer applied to an image ignores everything about images—that a pixel's neighbours are informative, that a cat is a cat wherever it appears in the frame—and pays for the omission with a weight matrix of 3072×100 parameters for a thirty-two pixel thumbnail. Building those two facts into the layer itself, rather than hoping the optimizer will discover them, gives the convolution, and with it the architectures that made the difference. That is the next chapter.

References

- D. Rumelhart, G. Hinton and R. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, 323, 1986. Paper
- V. Nair and G. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," *ICML*, 2010. Paper
- X. Glorot, A. Bordes and Y. Bengio, "Deep Sparse Rectifier Neural Networks," *AISTATS*, 2011. Proceedings

- A. Maas, A. Hannun and A. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.
- G. Montúfar, R. Pascanu, K. Cho and Y. Bengio, "On the Number of Linear Regions of Deep Neural Networks," *NIPS*, 2014. arXiv:1402.1869
- D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)," 2016. arXiv:1606.08415
- A. Baydin, B. Pearlmutter, A. Radul and J. Siskind, "Automatic Differentiation in Machine Learning: a Survey," *Journal of Machine Learning Research*, 18, 2018. Paper
- A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *NeurIPS*, 2019. arXiv:1912.01703

6. Convolutional Networks

Lecture 5

Based on Lecture 5 of CS231n, Stanford University, Spring 2025.

6.1. Two places to put the design

For about a decade before any of this worked, the field solved image classification with a pipeline that had a learned classifier at the end of it and something else in front. That something else was a feature extractor: a function, written by a person, that converted the pixels of an image into a vector believed to capture what mattered about it. The linear classifier of Section 3.4 was then fitted not to the pixels but to that vector. Understanding what the convolution is for is easier from this starting point than from a blank page, because the convolution is not the first attempt to give a classifier something better than raw pixels — it is the attempt that stopped choosing.

The simplest of these representations is a colour histogram. Divide the space of colours into some fixed set of buckets, assign every pixel of the image to the bucket its colour falls in, and let the feature vector be the count of pixels per bucket. The resulting representation is entirely insensitive to where anything is: an image with a red patch in the top-left corner and the same image with the patch moved to the bottom-right produce identical histograms, while in pixel space they have almost nothing in common. That is either exactly what you want, if you are sorting ripe apples from unripe ones, or a catastrophe, if you are not.

The histogram of oriented gradients (HoG) is the dual of that trade. It discards colour and keeps geometry. The image is divided into eight-by-eight pixel cells; within each cell the local edge directions are quantised into nine orientation bins and counted; the counts are concatenated. A 320×240 image yields a 40×30 grid of cells and therefore $30 \times 40 \times 9 = 10{,}800$ numbers. What survives is the pattern of oriented structure across the image — the diagonal veins of a leaf, the circular outline of an eye — with the colours thrown away.

A third family borrowed its idea from document retrieval. Bag-of-words representations first build a codebook: extract patches at random from a corpus of images, cluster them, and call the cluster centres visual words. An image is then encoded by matching each of its patches to the nearest word and counting how often each word occurs, exactly as a document is encoded by the frequencies of the terms in it. The version of this used for scene classification treats an image as an unordered collection of local appearances, which is a strong assumption and, for a while, a productive one.

Asked which of these representations was best, practitioners generally answered by using all of them: extract several, concatenate the results, and hand the classifier a long vector assembled from many partial views of the image. It is not an elegant answer, and its inelegance is a symptom.

Set the two systems side by side. Both take the pixels of an image and emit ten class scores. In the first, a fixed function computes features and a learned linear classifier sits on top; in the second, the whole path from pixels to scores is differentiable and every parameter in it moves under gradient descent. The systems differ only in where the boundary falls between what a person specified and what the data determined. Pushing that boundary all the way back to the pixels is the entire content of the phrase *end-to-end learning*, and the argument for doing so is not that human intuition is worthless but that it is a bottleneck: a person may be wrong about which aspects of the problem matter, and even when right, may be unable to write down the function that extracts them.

It would be a mistake to read this as the abolition of design. The convolutional network is not what you get by removing the human from the pipeline; it is what you get by moving the human's contribution one level up. A feature extractor is a particular function. An architecture is a *family* of functions — specified by which operators appear in the computational graph, in what order, with what tensor shapes — inside which gradient descent selects a member. Choosing the family remains a design act, and a consequential one; the next several chapters are largely about how that choice has been made and remade. What has changed is that the design is now expressed in a language the optimizer can work inside of.

6.2. What flattening throws away

The only architecture available so far is the one from Section 5.1: reshape the image into a vector, multiply by a matrix, apply a ReLU, repeat. Applied to CIFAR-10, the $3 \times 32 \times 32$ input becomes 3072 numbers and the first weight matrix is 100×3072. The reshape looks like bookkeeping. It is not.

Consider what the model would do if the pixels were shuffled. Fix any permutation of the 3072 input positions, apply it to every training and test image, and permute the columns of W_1 to match. The two networks compute identical outputs on identical inputs, reach identical losses, and generalise identically. A fully connected network cannot distinguish a photograph from a scrambled version of the same photograph, provided the scrambling is the same every time. Whatever information is carried by the fact that pixel 33 sits directly beneath pixel 1 is information the architecture has no way to use.

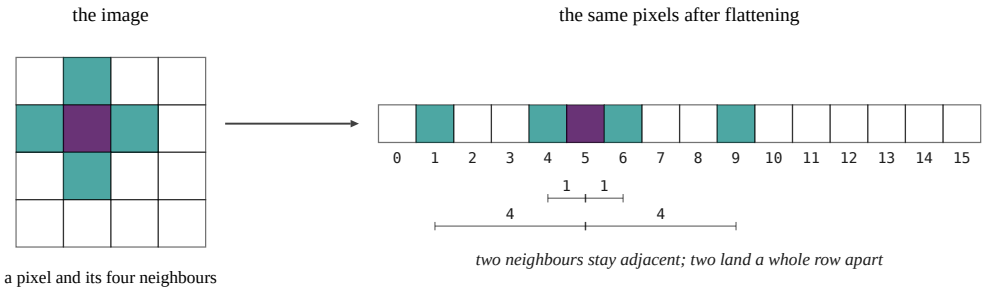


Figure 6.1.: Left: a 4×4 image, with one pixel and its four neighbours marked. Right: the same pixels after flattening. Two of the four neighbours land beside the pixel; the other two land a full row away, at distance 4, and would land at distance 224 in a real image. The layer that reads this vector treats all 16 positions as unrelated coordinates, so any consistent re-shuffling of them gives an identical model. *Drawn for these notes.*

The second cost is arithmetic. As Section 5.2 described, a row of W_1 is a template compared against the whole image by a 3072-dimensional dot product, and the comparison is anchored: the template's entry for position 500 is only ever multiplied against the pixel at position 500. A row that has learned to respond to a wheel in the lower-left of the frame contributes nothing when the wheel appears in the upper-right, and a second row must learn the same pattern again from scratch for the new location. The parameter count grows with the area of the image multiplied by the number of patterns, and the training data must exhibit every pattern in every place.

Both problems point at the same two facts about images, neither of which the fully connected layer knows. The first is locality: what a pixel means is largely determined by the pixels near it, so a feature worth computing can usually be computed from a small neighbourhood. The second is translation: a pattern signifies the same thing wherever it occurs, so whatever computation is applied at one location should be applied unchanged at every other. An operator that assumes both is not a heuristic bolted onto a general architecture. It is a narrower hypothesis class chosen because the narrowing is true.

6.3. A bit of history

Convolutional networks are old enough that their arrival can be dated twice. The architecture used for handwritten digit recognition in LeCun et al. (1998) — a body of alternating convolution and downsampling layers, feeding a small stack of fully connected layers, the whole thing trained by backpropagation — is recognisably the same object as the networks of the mid-2010s. It worked. It was also expensive, on hardware without anything resembling a GPU, and the datasets that would have rewarded a larger version of it did not yet exist.

Both constraints lifted at once. AlexNet, in 2012, is structurally the 1998 network with more layers, wider layers, a training set of a million labelled images and a pair of GPUs to train on, and Section 2.9 describes what it did to the ImageNet leaderboard. For roughly the following eight years, the answer to almost any question about images was a convolutional network: object detection, semantic segmentation, image captioning, and — in the form of the U-Net that denoises the latent in latent diffusion models — the first text-to-image systems that worked well enough to be worth arguing about. When this course was first taught in 2015 it was called *Convolutional Neural Networks for Visual Recognition*, which was a reasonable name at the time because the two halves of it referred to nearly the same thing.

The name changed because the field did. The transformer was published in 2017 for machine translation and spent a few years confined to text, until the Vision Transformer, presented at ICLR in 2021, cut an image into patches, treated the patches as a sequence of tokens, and ran essentially the unmodified language architecture over them. Across many of the tasks listed above it turned out that replacing the convolutional network with a transformer and changing nothing else improved results, and that the improvement widened as data and compute were scaled up.

That is an honest reason to be careful about how hard convolution is sold, and not a reason to skip it. Convolutional networks remain in wide practical use, particularly where inputs are small or latency is tight; large modern systems are frequently hybrids, with convolutional stems, convolutional decoders, or convolutional blocks interleaved with attention. The more durable reason to study the convolution first is that it is the clearest case in the subject of a prior about the data being written directly into an operator, and of what that buys — which is a lens that applies just as well to the architectures that displaced it.

6.4. The convolution layer

The generalisation from a fully connected layer to a convolutional one is smaller than it looks, and the right way to see it is through the dot product. A dot product between two vectors is large when they point in the same direction and zero when they are orthogonal, so any layer built out of dot products is performing template matching. The fully connected layer holds ten templates, each the size of the entire input, and reports how well each one matches the whole image. Keep the templates and the matching; change only the size of the templates.

A convolutional layer therefore does not flatten its input. It receives the image as the three-dimensional tensor it is, $3 \times 32 \times 32$, with two spatial dimensions and a channel dimension of depth 3 for the colours. Its templates, called filters or kernels, are small along the spatial dimensions and full-depth along the channels: a $3 \times 5 \times 5$ filter is a five-by-five patch of learnable weights on each of the three input channels. Filters always extend the full depth of the input volume, so there is only one spatial size to choose.

Position that filter over one $3 \times 5 \times 5$ chunk of the image, take the dot product of the filter with the chunk — a dot product in $5 \cdot 5 \cdot 3 = 75$ dimensions — and add a scalar bias. The result is a single number reporting how strongly this piece of the image resembles the template. Now slide the filter to every spatial position and repeat. A 5×5 filter fits into a 32×32 image in 28 positions horizontally and 28 vertically, so the numbers arrange themselves into a 28×28 grid, which is called an activation map. The map is an image of the filter's agreement with the input, evaluated everywhere.

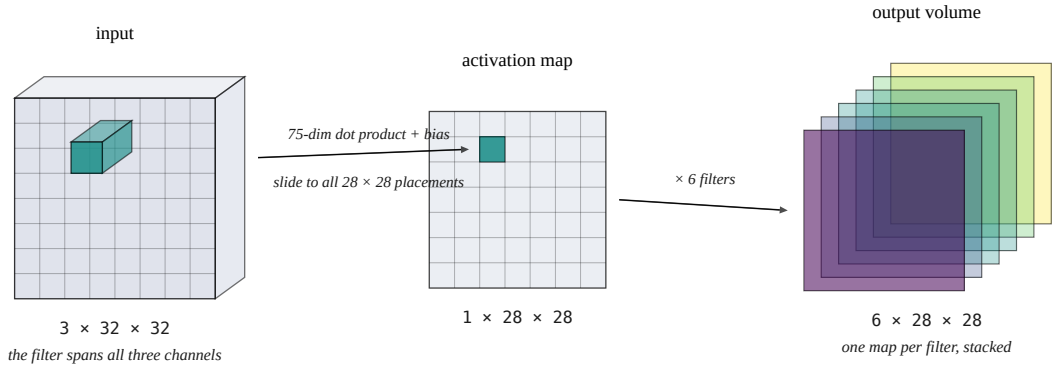


Figure 6.2.: One filter, applied. The filter spans all three input channels and five pixels in each spatial direction, so each placement contributes a 75-dimensional dot product plus a bias, producing one scalar. Sliding it over the 28 by 28 valid placements fills an activation map. Six filters produce six such maps, stacked into a $6 \times 28 \times 28$ output volume whose channel dimension indexes filters rather than colours. *Drawn for these notes.*

One filter answers one question about every location. A layer asks several. Six filters of shape $3 \times 5 \times 5$ — a single parameter tensor of shape $6 \times 3 \times 5 \times 5$, accompanied by a bias vector with one entry per filter — are each slid across the input independently, and the six resulting maps are stacked along a new channel dimension to give an output of shape $6 \times 28 \times 28$. Nothing couples the filters to each other; the layer is six separate slidings whose results are stored in one tensor.

That output tensor supports two readings, and both are worth carrying. Read by channel, it is

six activation maps, each showing where in the image one particular template fired. Read by position, it is a 28×28 grid at each point of which sits a six-dimensional vector describing the local appearance of the image there. The second reading explains why the output is legitimately an input for another convolution: it is again a stack of spatial planes, differing from the original image only in having six channels of learned features instead of three channels of colour.

In practice the layer is applied to a batch. With N images at once, an input of shape $N \times C_{\text{in}} \times H \times W$ meets a filter tensor of shape $C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w$ and a bias of length C_{out}, and produces

$$N \times C_{\text{out}} \times H' \times W', \tag{6.1}$$

where H' and W' are the numbers of valid filter placements, computed in Section 6.7. Four indices is the working dimensionality of nearly everything that follows.

One point deserves emphasis because it is easy to lose among the shapes. Nobody writes the filters. They are initialised randomly and learned by gradient descent, exactly as the weight matrices of Section 5.1 are, and the gradient of the loss with respect to each scalar in each filter is obtained by the backward traversal of Section 5.8 with no new machinery — the layer supplies a forward rule and a local derivative, which is all the interface of Section 5.11 requires of it. What a person chooses is the number of filters and their spatial size — hyperparameters, fixed before training begins because they fix the shapes of the tensors, and tuned by cross-validation like the hyperparameters of Section 3.3. What the data chooses is every number inside them.

6.5. Stacking convolutions

A convolutional network is a computational graph in which some of the nodes are convolutions. A first layer might apply six $3 \times 5 \times 5$ filters to a $3 \times 32 \times 32$ image and produce $6 \times 28 \times 28$; a second might apply ten $6 \times 5 \times 5$ filters to that and produce $10 \times 24 \times 24$; and so on, the input depth of each layer fixed by the output depth of the one before it. Because the layer is just another node with a forward rule and a local derivative, it slots into the graph beside the fully connected layers and losses already defined, and the whole network is trained by the same loop: batch, forward, loss, backward, step.

Stacking convolutions directly on top of one another, however, gains nothing. The argument is the one from Section 5.2 in different clothing. A convolution is a linear function of its input, so the composition of two convolutions is linear, and in fact is itself a convolution — two stacked 5×5 layers compute some 9×9 convolution, which a single layer with 9×9 filters could have computed directly. Depth without a non-linearity buys a larger filter and nothing else. So a ReLU is applied elementwise to every activation map after every convolution, and it is the interleaving, not the stacking, that makes depth mean anything.

The layer's hyperparameters are few, and worth separating from its parameters once and for all. The number of output channels C_{out} and the kernel size K set the shape of the weight tensor and are chosen before training; the stride and padding of Section 6.7 join them. The weights and biases inside that tensor are parameters, initialised randomly and moved by the optimizer. It is a distinction the shapes make obvious — a hyperparameter is something a gradient cannot be taken with respect to, because changing it changes what the tensors are.

Filters within one layer conventionally share a spatial size, which is a decision about implementation rather than about modelling: a fixed kernel size makes the layer a single dense arithmetic kernel that a GPU can execute efficiently. Architectures that want several receptive field sizes at

one depth get them by running several convolutional layers in parallel and concatenating the results — the Inception module of the next chapter is exactly this — which keeps the primitive simple and pushes the variety into the graph built from it.

6.6. What the filters learn

Three architectures, three answers to the question of what a learned weight looks like. A linear classifier holds one template per class, each the size of the whole image, and Section 3.4 showed what those templates look like when visualised: blurry class averages, with the two-headed horse as the emblem of their limits. A fully connected hidden layer holds a bank of such whole-image templates, no longer tied one-to-one to classes but still each spanning the entire frame. A convolutional layer holds a bank of *local* templates, each far too small to cover the image and each applied at every position in it.

Because a first-layer filter has exactly three channels and a small spatial extent, it can be displayed as what it is: a tiny colour image. AlexNet's first layer is 64 filters of shape $3 \times 11 \times 11$, and rendering them as sixty-four eleven-pixel thumbnails shows two recurring kinds. Some are oriented edges — a light half against a dark half, the boundary running vertically, horizontally or at an angle, at a range of spatial frequencies. The rest are opposing-colour patches: green against red, pink against green, responding to colour contrast with little spatial structure. Nothing in the training objective asked for either. The network was told to classify a thousand categories of photograph and these are what minimising that loss produced.

The result is close to universal. Convolutional networks trained on different datasets, for different tasks, with different architectures, converge to first layers that look substantially like this one. It is also a rediscovery: the oriented-edge detector is precisely what Section 2.3 describes Hubel and Wiesel finding in the cat's primary visual cortex in 1959, arrived at here by gradient descent on labelled photographs rather than by evolution. That two such different processes settle on the same first operation is a reasonable hint that the operation is forced by the statistics of natural images rather than by either process.

That the filters within a layer end up different from one another is not automatic, and the reason is worth pausing on. Suppose every filter in a layer were initialised to the same values. Each would then compute the same activation map, receive the same upstream gradient, and take the same step, forever; the layer would be a single filter copied C_{out} times, with C_{out} times the cost and none of the benefit. Random initialisation, with a *different* draw per filter, is what breaks this symmetry, and the diversity visible in a trained first layer is diversity that the initialisation made possible and the loss then found a use for.

Deeper layers cannot be shown this way. A filter in the sixth convolutional layer might have shape $512 \times 3 \times 3$, which is not an image of anything; its input is a stack of learned features, not colours, so displaying its weights as a picture is meaningless. The standard substitute is to run a large set of images through the network, find the input patches that drive a given filter hardest, and look at those. Done for a sixth layer of an ImageNet model, the resulting patches group into recognisable things: one filter answers to eyes, another to fragments of text, another to wheels and the upper arcs of circles. The progression from edges to parts is not designed and not supervised. It is what happens when each layer is given the previous layer's features to build from.

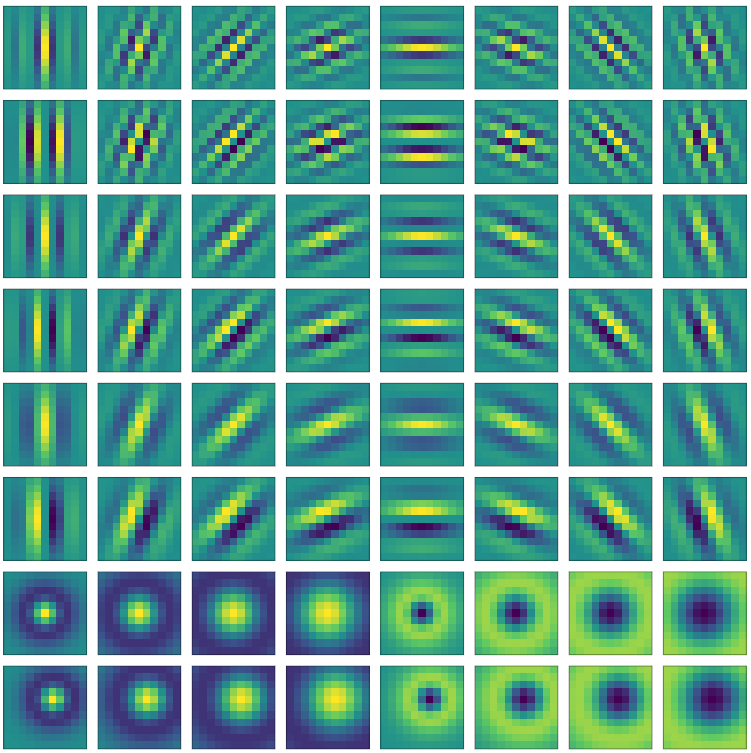


Figure 6.3.: Sixty-four templates at eleven pixels square, on the same value ramp. The first six rows are Gabor functions — an oriented sinusoid under a Gaussian envelope — at eight orientations, three spatial frequencies and two phases; the last two rows are centre-surround blobs at four scales and both polarities. These are *not* filters from a trained network. They are the analytic family that trained first layers reliably approximate, drawn here because the widely reproduced grid of AlexNet's real first layer is a figure from that paper and cannot be used in these notes. The point they carry is the one that matters: a template this small can encode an edge at an orientation and a scale, or a blob against its surround, and very little else. *Computed for these notes.*

6.7. Padding, stride, and the shape arithmetic

The output size in Equation 6.1 was deferred; it is elementary, and the two hyperparameters that appear in it are both introduced to fix problems it creates. Ignore the channel dimension, which never affects spatial size, and consider a 7×7 input with a 3×3 filter. The filter can be placed with its left edge at column 1, 2, 3, 4 or 5 and nowhere else, so the output is 5×5. In general an input of extent W and a filter of extent K admit

$$W' = W - K + 1 \tag{6.2}$$

placements along each axis. Every convolution therefore shrinks its input, and the shrinkage compounds: seven successive 5×5 layers reduce a 32×32 map to 4×4, and an eighth cannot be applied at all. Some architectures live with this. Most would rather not have the depth of the network limited by an accident of arithmetic, and rather not recompute every shape each time a layer is inserted.

The fix is to pad the input with P rows and columns of zeros on every side before sliding the filter, which restores the placements lost at the border:

$$W' = W - K + 1 + 2P. \tag{6.3}$$

Setting $P = (K - 1)/2$ makes $W' = W$, so the map keeps its spatial size through the layer. This is called *same* padding and is the usual choice; it also explains why kernel sizes are almost always odd, since $(K - 1)/2$ is an integer only then. A 3×3 convolution is paired with $P = 1$, a 5×5 with $P = 2$, and the spatial dimensions become something the architect sets deliberately rather than something that erodes.

Downsampling, when wanted, is then requested rather than suffered. A strided convolution moves the filter S positions at a time instead of one, so a 3×3 filter with stride 2 on a 7×7 input lands in three places along each axis and returns 3×3. Combining all three:

$$W' = \frac{W - K + 2P}{S} + 1. \tag{6.4}$$

A stride that does not divide $W - K + 2P$ leaves the division inexact, and implementations resolve it by taking the floor — the trailing rows and columns that no placement reaches are simply not looked at. It is a small thing and a common source of off-by-one confusion when a network is being assembled from shapes.

6.8. Receptive fields

Every element of a convolution's output is a dot product over a $K \times K$ window of the input, so it depends on exactly K^2 input positions and is blind to the rest of the image. That window is the element's receptive field. Composing layers widens it: a 3×3 filter in the second layer reads nine adjacent first-layer outputs, each of which read a 3×3 window, and those windows together span 5×5 of the original image. Each further layer adds $K - 1$, so after L convolutions of kernel size K the receptive field in the input is

$$1 + L(K - 1) \tag{6.5}$$

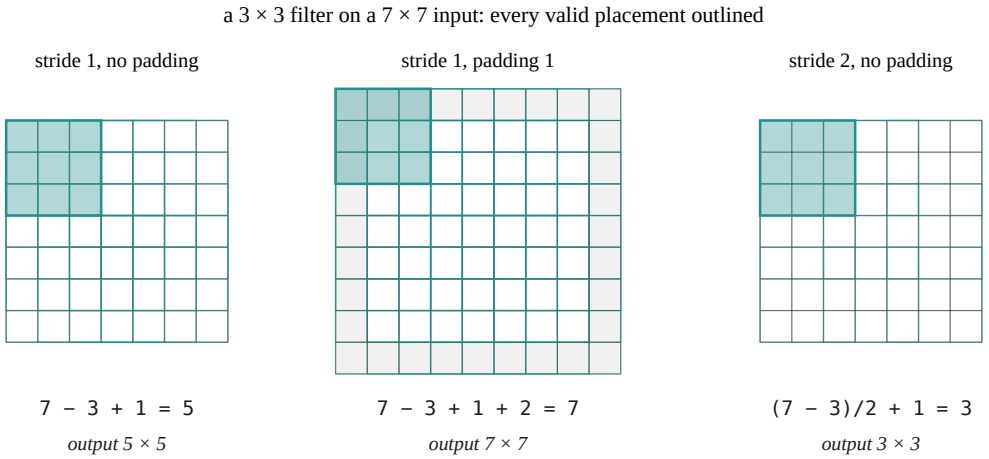


Figure 6.4.: Left: a 3×3 filter on a 7×7 input at stride 1, giving five placements per axis and a 5×5 output. Centre: one ring of zero padding restores the lost border placements, and the output is 7×7 again. Right: stride 2 on the unpadded input takes every other placement, giving 3×3. All three are Equation 6.4 with different arguments. *Drawn for these notes.*

pixels across. The phrase is used in two senses that are easy to confuse and worth keeping apart: the receptive field of a unit *in the previous layer* is always $K \times K$, while its receptive field *in the input image* is what Equation 6.5 describes.

Linear growth is the problem. With $K = 3$ the receptive field widens by two pixels per layer, so an output unit that must see all of a 224×224 image requires 112 convolutional layers to do it, and a 1024-pixel image requires more than 500. Classifying an image means combining evidence from all of it, and a network whose deepest units still see only a fraction of the frame cannot do that at any reasonable depth. This is also the answer to why deeper filters respond to larger structures: a unit can only be selective for an eye once its receptive field is big enough to contain one.

The remedy is to shrink the image inside the network rather than to lengthen the network. If a layer downsamples by a factor of two, every subsequent convolution's receptive field, measured back in the original image, is worth twice as much — the field grows geometrically with depth instead of arithmetically, and a few dozen layers suffice for any image size that will be met in practice. Strided convolution is one way to do it. Pooling, in Section 6.10, is the other.

6.9. Counting a layer

Everything above is arithmetic, and the arithmetic is worth doing once on a concrete layer, because the three numbers it produces — output shape, parameter count, operation count — are what an architecture is judged by. Take a $3 \times 32 \times 32$ input and a layer of ten 5×5 filters with stride 1 and padding 2.

The output shape follows from Equation 6.4: $(32 - 5 + 2 \cdot 2)/1 + 1 = 32$ along each spatial axis, and ten filters give ten channels, so the layer emits $10 \times 32 \times 32$. The parameter count follows from the filter shape: each filter is $3 \times 5 \times 5 = 75$ weights plus one bias, so 76 parameters, and ten filters give 760. The operation count follows from the output shape and the filter size

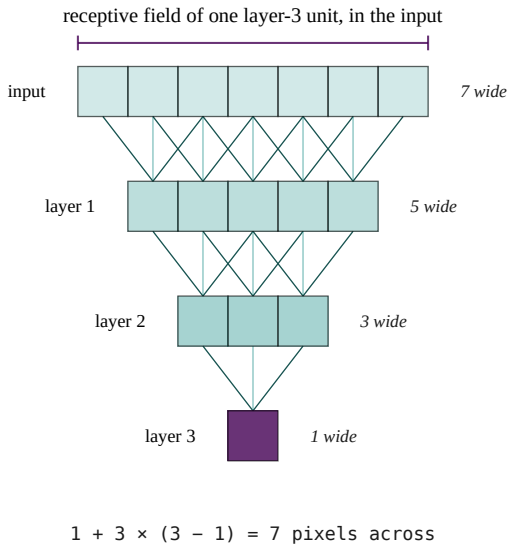


Figure 6.5.: Three stacked 3×3 convolutions, drawn in one spatial dimension so the dependencies are visible. One output element reads three elements of layer 2, which between them read five of layer 1, which between them read seven of the input — $1 + 3 \times 2 = 7$, as Equation 6.5 gives. The growth is two pixels per layer regardless of how deep the stack becomes. *Drawn for these notes.*

together: there are $10 \cdot 32 \cdot 32 = 10{,}240$ output elements, each a 75-dimensional dot product, so $75 \cdot 10{,}240 = 768{,}000$ multiply-adds.

Set that against a fully connected layer doing nominally the same job — 3072 inputs to 10,240 outputs. It would hold about 31.5 million weights and perform about 31.5 million multiply-adds. The convolution reaches the same output size with roughly one forty-thousandth of the parameters and one fortieth of the arithmetic. The parameter saving comes from weight sharing, since one filter serves all 1024 positions instead of each position owning its own weights; the arithmetic saving comes from locality, since each output consults 75 inputs instead of 3072. These are the two priors of Section 6.2, and this is their price in cash.

The asymmetry between the two counts matters more than either. Doubling the height and width of the input leaves the parameter count of a convolutional layer completely unchanged and quadruples its arithmetic. A convolutional network's size is a property of its architecture; its cost is a property of what you feed it. Nothing similar is true of a fully connected layer, whose weight matrix must grow with the input.

The full contract of the layer, then. It takes $C_{\text{in}} \times H \times W$ and has four hyperparameters: kernel size $K_h \times K_w$, number of filters C_{out}, padding P and stride S. It holds a weight tensor of shape $C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w$ and a bias vector of length C_{out}. It emits $C_{\text{out}} \times H' \times W'$ with H' and W' given by Equation 6.4. In practice the settings cluster tightly:

Setting	K	P	S	Effect
3×3 convolution	3	1	1	The default; preserves spatial size
5×5 convolution	5	2	1	Wider receptive field, preserves size

Setting	K	P	S	Effect
1×1 convolution	1	0	1	Changes channel count only
Downsampling convolution	3	1	2	Halves height and width

with channel counts conventionally powers of two — 32, 64, 128, 256. PyTorch's `nn.Conv2d` takes exactly these arguments, in the order `in_channels`, `out_channels`, `kernel_size`, `stride`, `padding`, and holds `weight` and `bias` tensors of exactly the shapes above.

The 1×1 entry in that table deserves a sentence, because it looks degenerate and is not. With $K = 1$ each output element is a dot product over a single spatial position across all input channels, so the layer applies one $C_{\text{in}} \to C_{\text{out}}$ linear map identically at every point of the feature map — a fully connected layer over channels, run in parallel over positions. It computes nothing about spatial structure, which is the point: it is the cheap way to change how many channels a representation has, and later chapters use it constantly to widen and narrow the channel dimension between expensive spatial convolutions.

Finally, nothing about the operator requires two spatial dimensions. A 1D convolution takes $C_{\text{in}} \times W$ and holds filters of shape $C_{\text{out}} \times C_{\text{in}} \times K$, sliding along one axis; it is the natural layer for audio waveforms and for text treated as a sequence. A 3D convolution takes $C_{\text{in}} \times H \times W \times D$ with filters of shape $C_{\text{out}} \times C_{\text{in}} \times K \times K \times K$, sliding through a volume, and is used for medical scans and for video where time is the third axis. The slab of weights changes shape; the sliding dot product does not.

6.10. Pooling

Strided convolution downsamples while doing work. Pooling downsamples and does nothing else, which is its appeal: it is nearly free, and it holds no parameters at all.

The operation treats the channels as independent. Given an input of shape $C \times H \times W$, each of the C planes is pulled out on its own, reduced spatially, and the results restacked, so the channel count passes through untouched and only H and W change. Within a plane the reduction is described with the same vocabulary as convolution — a kernel size K and a stride S — and adds one hyperparameter that convolution does not have, namely the function used to reduce each window. Max pooling takes the largest value in the window; average pooling takes the mean.

The overwhelmingly common setting is $K = 2$, $S = 2$ with the max, which cuts the tiles into a non-overlapping grid and halves both spatial dimensions. On the four-by-four plane

$$\begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}, \tag{6.6}$$

each output being the maximum of the 2×2 tile above it. The output size follows Equation 6.4 with the padding term dropped, $H' = (H - K)/S + 1$.

Padding is not used here, and the reason is a small piece of arithmetic rather than a convention. Pooling normally follows a ReLU, so its inputs are non-negative; padding the plane with zeros

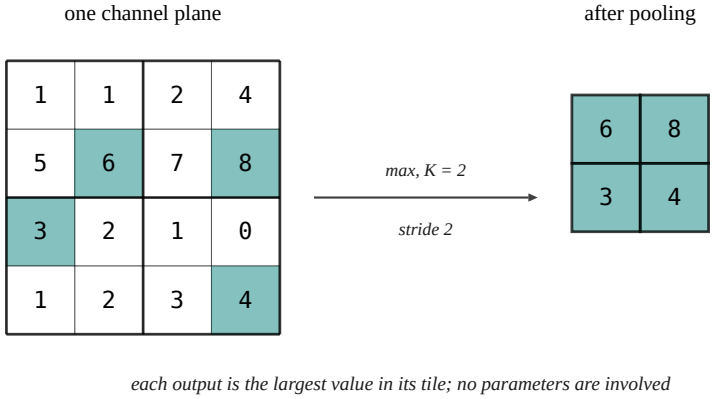


Figure 6.6.: Max pooling with a 2×2 kernel at stride 2, on one channel plane. The four non-overlapping tiles are reduced to their maxima; the channel dimension is untouched, so a $64 \times 224 \times 224$ input becomes $64 \times 112 \times 112$. Shifting the 6 anywhere else within its tile leaves the output unchanged, which is the local invariance the layer buys. *Drawn for these notes.*

and taking a maximum over a window containing them therefore changes nothing, and the whole exercise reduces to a no-op at the border.

Two further consequences of choosing the max are worth naming. It is a non-linearity in its own right, so a convolution followed by max pooling has one whether or not a ReLU was inserted between them, whereas average pooling is linear and supplies none. And it grants a small amount of local invariance: the value 6 in Equation 6.6 can move to any of the four positions in its tile and the output is unchanged, so the layer is deliberately blind to shifts smaller than its kernel.

Interleaved with convolutions, this gives the shape of a classical convolutional network — two convolutions, a pool, two convolutions, a pool, and so on until the spatial dimensions are small, then a flatten and one or two fully connected layers to produce class scores. The next chapter is largely a history of variations on that skeleton, including the observation that a stride-2 convolution can do the same downsampling with parameters attached, which is why pooling is less ubiquitous in recent architectures than the pattern above suggests.

One practical consequence of the flatten at the end deserves flagging. The fully connected head expects a fixed number of inputs, so a network built this way accepts exactly one input resolution, and images of any other size must be resized, padded, or grouped so that each batch is internally uniform — aspect-ratio bucketing, in the larger training pipelines. It is a constraint the convolutional body does not have and the classifier head imposes, and later architectures remove it by replacing the flatten with a pooling operation over the whole spatial extent.

6.11. Equivariance

The claim that convolution and pooling respect the spatial structure of images has been used repeatedly above as an intuition. It has a precise form. Let f be a convolution or a pooling layer and let T shift an image some number of pixels to the side. Then

$$f(T(x)) = T(f(x)) \tag{6.7}$$

for every input x — shifting the image and then applying the layer gives the same thing as applying the layer and then shifting the result. The equality holds up to boundary conditions, which is a real caveat and not an interesting one; away from the edges of a large image it is exact.

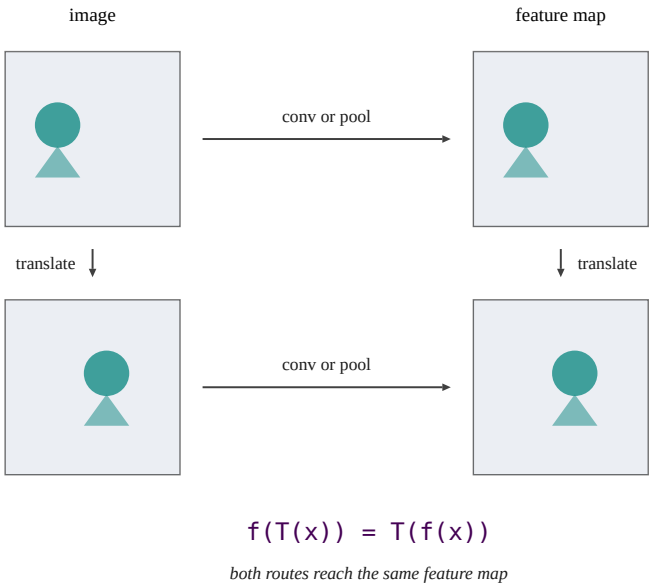


Figure 6.7.: The commutative square of Equation 6.7. Both paths from the top-left to the bottom-right produce the same feature map: the operator commutes with translation. Nobody performs the shift in practice — it is a statement about what the operator is, not a step in a computation. *Drawn for these notes.*

This is a property of the operator, not a preprocessing trick and not something performed during training. What makes it worth stating is that it is exactly the structural commitment Section 6.2 asked for, written down as an equation. A convolution computes each output from the same weights regardless of where in the frame it is looking, so a pattern that moves produces a response that moves with it, unchanged in value. The features an image yields depend on its content and not on where in the frame that content sits. A fully connected layer satisfies nothing of the kind: shift the input by one pixel and every dot product in the layer changes, with no relationship between the old output and the new.

Equivariance should not be confused with invariance, and the distinction organises how a network is put together. An equivariant layer's output moves when the input moves; an invariant one's output does not change at all. Convolution is equivariant, which is what you want in the body of a network, because a feature map that shifted with the image is still a usable description of where things are. Max pooling adds a small amount of genuine invariance, discarding shifts smaller than its window. Invariance to the whole frame arrives only at the end, from the classifier that collapses the spatial dimensions — a cat anywhere in the image should produce the same class score, but only after the layers that needed to know where it was have finished.

6.12. What this leaves

Two operators were added to the vocabulary of Section 5.11 in this chapter, and neither required anything new of the machinery. A convolution is a bank of small learned templates, applied at

every position and stacked into a volume; a pooling layer reduces that volume spatially without learning anything. Both are differentiable, both fit into the computational graph beside the layers already there, and both are trained by the same loop as everything else. What distinguishes them is not their mathematics but the assumption they encode — that features are local and that location does not change meaning — which is the reason they need so few parameters to work.

What remains open is how to assemble them. The chapter has specified a layer without specifying a network: how deep, how wide, how the channel count should change as the spatial dimensions shrink, where to downsample and by how much, and whether the answers found for one problem transfer to another. Those questions were answered empirically, over roughly a decade of competition results, and the answers turned out to be more interesting than the layer. That is the next chapter.

References

- D. Lowe, "Object Recognition from Local Scale-Invariant Features," *ICCV*, 1999. Paper
- Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, 86(11), 1998. Paper
- N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," *CVPR*, 2005. Paper
- L. Fei-Fei and P. Perona, "A Bayesian Hierarchical Model for Learning Natural Scene Categories," *CVPR*, 2005. Paper
- A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *NeurIPS*, 2012. Paper
- J. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *ICLR*, 2015. arXiv:1412.6806
- A. Vaswani et al., "Attention Is All You Need," *NeurIPS*, 2017. arXiv:1706.03762
- A. Dosovitskiy et al., "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ICLR*, 2021. arXiv:2010.11929
- R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," *CVPR*, 2022. arXiv:2112.10752

7. CNN Architectures

Lecture 6

Based on Lecture 6 of CS231n, Stanford University, Spring 2025.

7.1. The rest of the kit

A convolutional network is built from six kinds of layer, and Section 6.4 and Section 6.10 have covered two of them while Section 5.1 covered a third. Convolution supplies the learned templates, pooling reduces resolution without learning anything, and the fully connected layer at the end turns whatever the stack produced into class scores. Activation functions were treated in Section 5.3. That leaves two, and both are there for reasons that have nothing to do with representing images.

Normalization layers exist because a deep stack is numerically fragile. The output of one layer is the input to the next, and nothing in the definition of a convolution keeps the scale of its output anywhere near the scale of its input; over thirty layers a modest drift compounds into activations that are either negligible or enormous, and in both regimes the gradient that has to travel back through them is useless. A normalization layer intervenes directly, rescaling its input to a fixed distribution before passing it on. Dropout exists for a different reason entirely: it deliberately damages the network during training, in order that the network cannot come to depend on any particular part of itself.

The two are worth taking together because the second is the first clear instance of a pattern this chapter returns to at length. Dropout is not a mechanism for representing anything. It is randomness injected during training and removed at test time, and the several other techniques that share that shape — data augmentation, and, incidentally, batch normalization — turn out to be doing the same job by the same logic. Section 7.11 makes that argument once, for all of them. Here, dropout is introduced as what it is on the page: a layer.

7.2. Normalization layers

Every normalization layer in current use does the same two things in the same order, and the entire family differs only in a choice made inside the first. Step one computes a mean and a variance from some subset of the activations and uses them to standardize that subset to zero mean and unit variance. Step two undoes this, partially, through two learned parameters — a scale and a shift — applied per feature.

The second step deserves an explanation before the first, because it looks like it cancels the work just done. It does not, and the reason is that the parameters are learned. Forcing every layer's output to be a unit Gaussian is a strong constraint and not obviously the right one; a layer

feeding a sigmoid might want its inputs concentrated near zero, and a layer feeding a ReLU might want a mean well above it. Standardizing and then applying a learned γ and β does not fix the distribution, it fixes the *parameterization* of the distribution. The mean and variance of the layer's output are now controlled by two parameters that gradient descent can set directly, instead of emerging as an accident of every weight in every layer below. That the network could in principle recover the identity function by learning $\gamma = \sigma$ and $\beta = \mu$ is the point: nothing has been taken away, and something has been made easier to steer.

Take layer normalization, the variant most common today and the one used in essentially every transformer. Given a batch of N samples each of dimension D, written $x_{i,j}$, the statistics are computed within each sample, across its features:

$$\mu_i = \frac{1}{D}\sum_{j=1}^{D} x_{i,j}, \qquad \sigma_i^2 = \frac{1}{D}\sum_{j=1}^{D} \left(x_{i,j} - \mu_i\right)^2. \tag{7.1}$$

There are N means and N variances, one pair per sample. The normalized value and the layer's output are then

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}, \qquad y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j. \tag{7.2}$$

The ε is a small constant, typically 10^{-5}, present only so that a feature vector with no variation does not divide by zero. Note the index on γ and β: they are indexed by feature, not by sample, so there are D of each and they are shared across the batch. The statistics vary per sample; the learned correction does not.

Now change the choice. Suppose the statistics are computed per feature instead, averaging over the samples in the batch:

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}, \tag{7.3}$$

with the variance defined correspondingly. That is batch normalization, and the arithmetic downstream of Equation 7.3 is identical. Only the axis of the average has moved.

For a convolutional network the activations are a four-dimensional block of shape $N \times C \times H \times W$ — batch, channel, height, width — and the choice becomes richer, because there are now three axes to average over rather than one. Layer normalization takes one sample and averages over all of its channels and all of its spatial positions: one mean per image. Batch normalization takes one channel and averages over the batch and over space: one mean per channel, computed across every image in the mini-batch. Instance normalization is the intersection, one mean per image per channel. Group normalization splits the channels into a fixed number of groups and normalizes within each, sitting between the layer and instance cases and reducing to each of them at the extremes of the group count.

The reason the choice matters — and the reason layer normalization displaced batch normalization in most new architectures — is that batch normalization makes the output for one image depend on the other images that happen to share its mini-batch. During training that is merely strange. At test time it is unusable: predictions must not depend on what else was in the batch, and there may not be a batch at all. Batch normalization therefore has to behave differently in the two regimes. During training it accumulates a running estimate of each channel's mean and variance across all batches seen, and at test time it uses those fixed estimates in place of the

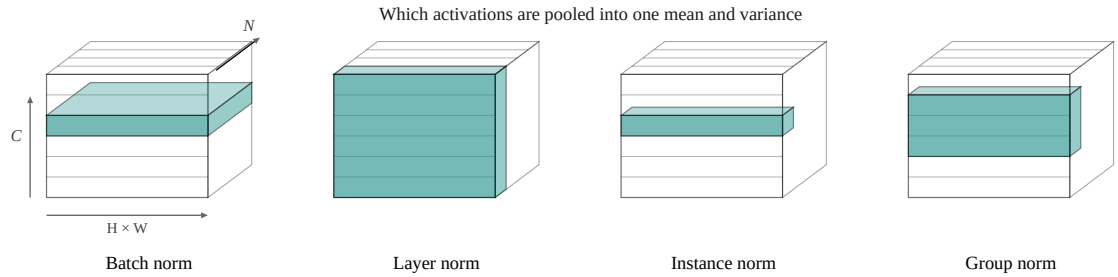


Figure 7.1.: Which activations each normalization layer averages over. The block is one mini-batch of convolutional activations, N samples deep, C channels tall, with the $H \times W$ spatial grid flattened onto the horizontal axis; the shaded cells are the ones pooled into a single mean and variance, and the same cells then receive that statistic. All four then apply a per-channel learned scale and shift. The only thing that distinguishes them is the shape of the shaded region. *Drawn for these notes.*

batch statistics. This works, but it introduces a discrepancy between the function being trained and the function being evaluated, it degrades badly when the batch is small enough that its statistics are noisy, and it is a recurring source of bugs in which a model in the wrong mode silently produces different numbers. Layer normalization has none of these problems, because Equation 7.1 never looks outside the sample. It is the same computation at training and at test, for a batch of one thousand or a batch of one.

One caution about the standard explanation. The batch normalization paper attributed the benefit to reducing *internal covariate shift*, the drift in each layer's input distribution as the layers below it are updated, and that phrase is still repeated. The empirical case for it is weak: later work found that deliberately injecting distribution shift after a normalization layer does not remove the benefit, and argued instead that normalization helps by making the loss surface smoother, so that the gradient at a point remains a good guide over a larger step. The layer works. The original account of why is not something to repeat with confidence.

7.3. Dropout

Dropout is a layer with no parameters and one hyperparameter. On each forward pass during training it samples a fresh binary mask and multiplies the incoming activations by it, setting each one to zero independently with probability p:

$$m_j \sim \text{Bernoulli}(1-p), \qquad y_j = m_j\, x_j. \tag{7.4}$$

A dropped unit contributes nothing forward, and — by exactly the argument of Section 5.10, since multiplication by zero is what a closed ReLU also does — receives no gradient back, so the weights that feed it go unmodified on that step. The mask is resampled every forward pass, so no unit is dropped for long, and p is typically 0.5 in a fully connected layer.

The obvious objection is that this is vandalism. Half the network's computed features are thrown away at random, training accuracy suffers, and none of it is targeted at anything. The standard defence is that it prevents co-adaptation. Consider a layer near the output of a cat classifier whose units have come to detect ears, a tail, fur, claws. If the training set is such that ears and fur reliably occur together, the layer above can learn a rule that requires both, and that rule

will be brittle on any image where one of them is occluded or unusual. Under dropout the two are only sometimes both present, so a rule that requires both is punished during training, and what survives is a broader and more redundant set of correspondences between features and classes. The network is prevented from concentrating its evidence.

A second reading is sharper and explains the test-time behaviour. Each sampled mask defines a different sub-network, and training with dropout trains an enormous ensemble of these sub-networks, all sharing one set of weights. A fully connected layer of 4096 units admits 2^{4096} masks, a number with more than twelve hundred digits, so no mask is ever seen twice and no sub-network is ever trained to convergence individually. Ensembling normally means averaging the predictions of separately trained models; here the averaging has to be approximated in a single forward pass.

That approximation is the whole of the test-time rule. At test time no units are dropped, which means each unit now receives roughly twice the input it saw during training and every activation downstream is inflated. What is wanted is that the test-time output equal the *expected* training-time output, and since the expectation of Equation 7.4 is $(1-p)\,x_j$, the fix is to multiply the test-time activations by the keep probability $(1-p)$. In practice the scaling is moved to the other side — divide by $(1-p)$ during training and leave the test path untouched — which is called inverted dropout and is what every framework implements, because it keeps inference free of any dropout-specific arithmetic.

Dropout has largely disappeared from convolutional architectures, and the reason is instructive rather than incidental. It was introduced when the parameter mass of a network sat in large fully connected layers, which is where it does its work; the architectures of Section 7.6 and Section 7.8 moved that mass into convolutions and then removed the large fully connected layers altogether, at which point there was much less left to co-adapt. It survives where those layers survive, in transformers most of all, and in convolutional networks it has been displaced by the regularizers of Section 7.11, which achieve the same end without touching the architecture.

7.4. Activations, settled

Section 5.3 derived what the choice of activation function is a choice between: saturation at both ends for the sigmoid and tanh, a gradient that is exactly zero or exactly one for the ReLU, smoothed variants such as GELU that keep the ReLU's shape without its kink at the origin. What that section could not supply, because it had no architectures to compare across, is how much the choice is worth.

The answer is: not much, and this is worth stating plainly because the number of proposed activation functions suggests otherwise. The most thorough attempt to find a better one searched a space of candidate functions automatically and reported, for its winner against ReLU on ImageNet with the architecture held fixed, a gain of 0.9 points of top-1 accuracy on one network and 0.6 on another. That is the result of an explicit search for the best available replacement; the spread across the functions someone would actually pick between is smaller still. The gap between the architectures compared in Section 7.5 is an order of magnitude larger. An activation function is a decision that should be made once, cheaply and early — ReLU for a convolutional network, GELU where a smooth alternative is conventional, as in transformers — and then left alone while the effort goes somewhere it can pay.

The single exception is the one Section 5.3 already made: do not use a sigmoid or a tanh in the body of a deep network. That is not a matter of a point of accuracy but of whether the gradient survives the trip back at all.

7.5. The revolution of depth

Section 6.12 left the question of how to assemble the layers, and noted that it was settled empirically. The venue for that was the ImageNet challenge, run annually from 2010 to 2017, and its results are unusually legible: one task, one test set, one number per year, over the exact period in which the field changed. What the sequence records is not a series of clever ideas. It is a single variable increasing.

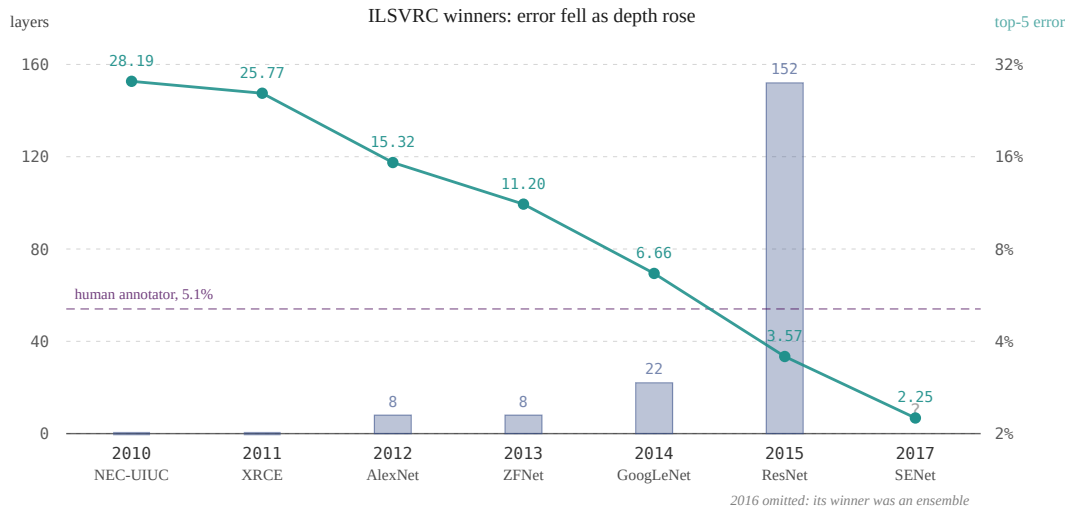


Figure 7.2.: Winning top-5 error on ILSVRC against the depth of the winning entry, 2010–2017. The two pre-2012 winners are feature pipelines with a shallow classifier on top and are drawn at zero depth; from 2012 the winner is a convolutional network every year. Human performance on the same task, measured at 5.1% by the challenge organisers, is marked. Error is plotted on a log scale, because a drop from 28% to 16% and a drop from 3.6% to 2.3% are not comparable as differences and are comparable as ratios. Plotted from the published challenge results and the papers cited in the text. *Computed for these notes.*

The two entries before 2012 are hand-designed feature pipelines of the kind Section 6.1 described, and their error is around 26%. AlexNet has eight layers and takes it to 15.3%, a margin over the best non-network entry that was itself the news of the year. The 2013 winner has eight layers and refines AlexNet's hyperparameters. Then 2014, in which two entries arrive at once — VGG with nineteen layers and GoogLeNet with twenty-two — and the error halves again. In 2015 ResNet has a hundred and fifty-two layers and reports 3.57%, passing the 5.1% that careful human annotators achieve on the same thousand categories. The two remaining years refine rather than restructure, and the challenge stopped.

Reading this as a story about depth is not a retrospective imposition; it is what the winning papers say they are doing. But depth alone is not a design, and the interesting content of the period is in the two questions it forced. If more layers are better, what should each layer be — because a network of a hundred and fifty-two distinct hand-chosen layers is not something anybody can design. And why did the answer to the first question stop working at around twenty layers, which it demonstrably did. VGG answers the first. ResNet answers the second.

7.6. VGG: design by rule

VGG is the network that stopped choosing layers individually. Its entire architecture follows from three rules. Every convolution is 3×3 with stride 1 and padding 1. Every pooling layer is 2×2 max pooling with stride 2. After every pooling layer, the number of channels doubles. Everything else — sixteen or nineteen layers depending on the variant, arranged in blocks of two or three convolutions between pools — is a matter of how many times the rules are applied. AlexNet, by comparison, has 11×11 filters in its first layer, 5×5 in its second, 3×3 thereafter, with strides and pool sizes chosen per layer. VGG looks like a scaled-up AlexNet in a block diagram, and in the respect that matters it is the opposite: it has no per-layer choices left to make.

Each rule earns its place. Consider first why the filters are all 3×3, which looks like giving something up — a 3×3 filter sees almost nothing. Stack three of them and, by the arithmetic of Section 6.8, each output position depends on a 7×7 window of the input: the first layer's receptive field is 3×3, and each subsequent stride-1 layer adds one position on each side, so the field grows by two per layer. Three 3×3 layers and one 7×7 layer therefore see exactly the same region of the image.

They do not cost the same. For a layer with C input and C output channels, a 7×7 convolution holds

$$7 \times 7 \times C \times C = 49C^2 \tag{7.5}$$

weights, while three 3×3 layers hold

$$3 \times (3 \times 3 \times C \times C) = 27C^2. \tag{7.6}$$

Forty-five per cent fewer parameters for the same receptive field, and the same saving in arithmetic since both are applied at every spatial position. That is the smaller of the two gains. The larger one is that the stack has three nonlinearities where the single layer has one. A 7×7 convolution, however many channels it has, computes a linear function of its 7×7 window; the three-layer stack computes a composition of three linear maps separated by ReLUs, which is a strictly larger family of functions over the same window. Smaller filters, stacked, are both cheaper and more expressive. There is no trade here to balance, which is why the rule holds throughout the network rather than at selected layers.

The channel-doubling rule is the one that is easiest to mistake for arbitrary, and it follows from a similar calculation. The cost of a convolutional layer, in both weights and multiplications, is proportional to $C_{\text{in}}C_{\text{out}}$ per output position, and to $H \times W$ positions. Halving H and W at a pooling layer cuts the number of positions by four. Doubling the channels multiplies the per-position cost by four. The product is unchanged:

$$\frac{H}{2} \cdot \frac{W}{2} \cdot 9 \cdot (2C)(2C) \;=\; H \cdot W \cdot 9 \cdot C \cdot C. \tag{7.7}$$

Every stage of the network therefore performs the same amount of work, and the network spends its arithmetic evenly across scales instead of concentrating it at the resolution it happens to start with. This convention outlived VGG entirely; ResNet uses it, and so does essentially every convolutional architecture since.

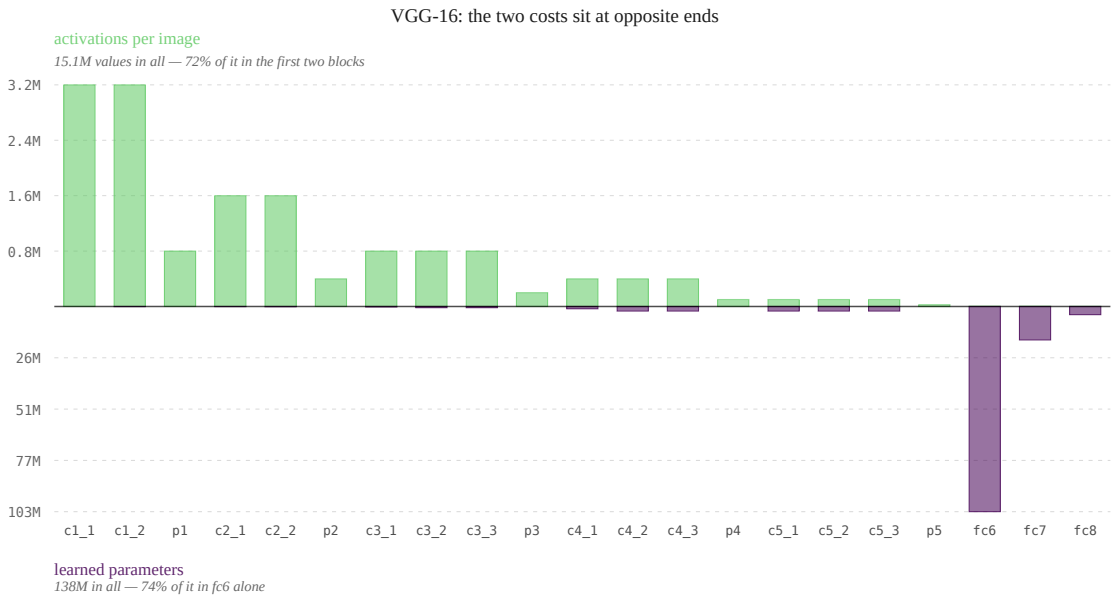


Figure 7.3.: Activation memory and parameter count for every layer of VGG-16, on a single 224×224 input. Both axes are counts, not bytes; the 15.1 million activation values come to about 60 MB per image at four bytes each, for the forward pass alone, and roughly double that once the values the backward pass needs are held. The two distributions barely overlap: memory is concentrated in the first two blocks, where the spatial grid is still large, and parameters are concentrated in the first fully connected layer, which alone holds 74% of the network. Computed from the architecture of configuration D of the VGG paper. *Computed for these notes.*

What the rules do not do is control where the *resources* end up, and VGG-16 is the clearest available illustration that the two costs of a network sit in different places.

Memory lives at the front. The first convolutional layer produces $224 \times 224 \times 64$ activations, over three million numbers from a single image, and the first two blocks together account for 72% of the network's activation memory while holding under a fifth of one per cent of its weights. This is why batch size is limited by early layers and why the input resolution is the most expensive hyperparameter in a convolutional network: both costs scale with $H \times W$.

Parameters live at the back, and specifically in one layer. After the final pooling the activation volume is $7 \times 7 \times 512$, and the first fully connected layer maps all 25,088 of those values to 4096 units, which requires

$$7 \times 7 \times 512 \times 4096 \approx 1.03 \times 10^8 \tag{7.8}$$

weights — about 74% of VGG-16's 138 million. One layer, doing nothing structurally interesting, holds three quarters of the model. It is the layer that discards spatial structure by flattening, which Section 6.2 argued against on other grounds, and it is also where dropout was most needed and most effective. Both observations point the same way, and the architecture in the next section acts on them.

7.7. Where depth stopped working

If the VGG rules are right, the obvious experiment is to apply them further. Take a plain convolutional network of twenty layers, build one of fifty-six by the same recipe, train both on CIFAR-10, and compare. The ResNet authors ran it, and the fifty-six-layer network was worse.

The reflex is to call this overfitting: more parameters, less generalization, a familiar story. The reflex is wrong, and the evidence that kills it is in the same figure. The fifty-six-layer network is also worse on the *training* set. An overfitting model fits the training data too well; this one fits it less well, having strictly more capacity with which to do so. Nothing about generalization is implicated. The deeper network is failing at the easier task.

This is worth stating carefully, because the conclusion is stronger than it first appears. Any function a twenty-layer network computes can be computed by a fifty-six-layer network: take the twenty-layer solution, and set the remaining thirty-six layers to the identity. The deeper network's hypothesis space contains the shallower one's entirely, so its optimal training error cannot be higher. The observed training error is higher. Therefore the problem is not representational but optimizational — the solution exists and gradient descent does not find it — and, as the lecture notes, training for longer does not close the gap. The construction just described is what points at the fix. It requires thirty-six layers to learn the identity function, and a stack of convolutions with a ReLU on top has no particularly easy way to do that. Every weight matrix would have to arrive at exactly the right value, and there is no gradient signal pushing any of them there. The identity is available in principle and unreachable in practice.

7.8. Residual networks

The residual block changes what the layers are asked to learn. Where a plain block computes some mapping $H(x)$ directly from its input, a residual block computes a correction and adds the input back:

$$y = F(x) + x, \tag{7.9}$$

where F is the same pair of 3×3 convolutions a plain block would have contained, and the second addend is the input carried unchanged along a *skip connection* that goes around them. Rearranged, $F(x) = y - x$: the convolutions are no longer fitting the desired output but the difference between the desired output and what already arrived, which is what gives the block its name.

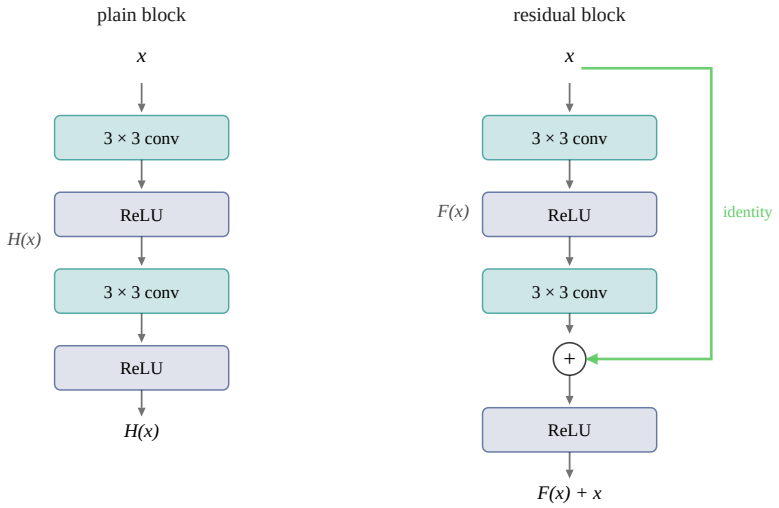


Figure 7.4.: A plain block and a residual block. The convolutions are identical; the only structural difference is the path on the right, which carries the input around them and adds it back before the final activation. Setting F to zero leaves the block computing the identity, which is why the difficulty described in Section 7.7 goes away. *Drawn for these notes.*

Now reconsider the identity. To make Equation 7.9 compute $y = x$, the block needs $F(x) = 0$, which it achieves by driving its convolutional weights to zero — a state that weight decay actively pulls towards, that small initialization already starts near, and that gradient descent reaches by shrinking values rather than by coordinating them. The identity has gone from the hardest thing for a block to represent to the easiest. A deep residual network that needs only twenty layers' worth of computation can switch the rest off, and it can do so smoothly, one block at a time, without any block having to be correct before the others are. Depth stops being a commitment. It becomes a budget the network may decline to spend.

There is a second reading, in terms of gradients rather than functions, that explains why the effect is so reliable. Differentiating Equation 7.9 gives $\partial y/\partial x = \partial F/\partial x + 1$. By the addition rule of Section 5.10, the skip connection distributes the incoming gradient unchanged to both branches, so every block passes its gradient backwards intact regardless of what its convolutions are doing. The gradient reaching layer one of a hundred-and-fifty-two-layer network no longer

has to survive a hundred and fifty-one multiplications; there is a path along which it survives none of them. The honest position is that neither reading is a proof — the lecture is explicit that why residual connections work is still argued over — but the empirical fact is not in dispute, and it is why the same connection appears in every transformer.

The full architecture assembles from there with few surprises. A single 7×7 convolution with stride 2 and a pooling layer reduce the input, a choice the authors report as empirical rather than principled. The body is a stack of residual blocks, each two 3×3 convolutions, following VGG's conventions throughout: same padding, same filter size, and periodically a stride-2 block that halves the spatial dimensions and doubles the channels, which keeps Equation 7.7 satisfied. At a downsampling block the skip connection cannot be a bare copy — the tensors no longer match in shape — so the shortcut carries a 1×1 convolution with stride 2 to bring x into agreement with $F(x)$.

The end of the network is where ResNet departs from VGG most sharply, and Figure 7.3 explains why. There are no large fully connected layers. The final activation volume is reduced to one number per channel by global average pooling — the mean over the entire spatial grid — and a single linear layer maps those channels to the class scores. The hundred-million-parameter layer of Equation 7.8 is gone, and with it most of the network's parameter count: ResNet-50 has about 25 million parameters against VGG-16's 138 million, while being three times as deep and considerably more accurate. The deeper variants add one more economy, the *bottleneck* block, which replaces the two 3×3 convolutions with a 1×1 that cuts the channel count by four, a 3×3 at that reduced width, and a 1×1 that restores it. The expensive 3×3 then operates on a quarter of the channels, which is what makes a hundred and fifty-two layers affordable at all.

The family was released at several depths — 18, 34, 50, 101, 152 — and accuracy improves monotonically with depth across all of them, which is precisely the property Section 7.7 showed plain networks lack. The improvement flattens: the gap from 101 to 152 layers is around a point, and the authors stopped there rather than at a principled limit. What the result established is that depth had not been exhausted in 2014; it had been blocked by an optimization failure, and removing that failure let a decade of architecture work proceed. For roughly five years afterwards, a ResNet was the default backbone for nearly every task in computer vision.

7.9. Initialization

The architecture is now specified, and none of it trains from an arbitrary starting point. Every weight has to be given a value before the first forward pass, and although Section 4.6 treated that as a detail, at depth it is not one.

Two options can be dismissed immediately. All zeros makes every unit in a layer compute the same thing, receive the same gradient, and remain identical forever — the network never breaks symmetry and behaves as though each layer had one unit. All the same nonzero constant fails for the same reason. Whatever is done must be random, and the only remaining question is the scale.

Take a six-layer network of 4096 units and draw every weight from a Gaussian with standard deviation 0.01, then look at the distribution of activations layer by layer. Layer one produces a reasonable spread. Layer two is narrower. By layer six the activations are indistinguishable from zero. Nothing has gone wrong arithmetically; each layer multiplies by weights smaller than unity in aggregate and the scale shrinks geometrically. The consequence is fatal for training, because a layer whose input is zero produces a zero gradient for its weights — Section 5.10 established that the gradient with respect to a weight is proportional to the activation it multiplies — so nothing

below the collapse ever learns. Now raise the standard deviation to 0.05. The activations grow instead, layer by layer, and by layer six they are large enough that a tanh saturates completely or a ReLU produces values that overflow into the loss. The gradient dies at the other end.

What the two failures have in common is that the *variance* of the activations is not preserved from layer to layer, and that suggests the fix. For a linear layer $y = Wx$ with D_{in} inputs, weights drawn independently with variance σ_w^2, and inputs independent with variance σ_x^2, each output is a sum of D_{in} independent products, so

$$\operatorname{Var}(y_j) = D_{\text{in}}\,\sigma_w^2\,\sigma_x^2. \tag{7.10}$$

Requiring $\operatorname{Var}(y) = \operatorname{Var}(x)$ gives $\sigma_w^2 = 1/D_{\text{in}}$, which is Xavier initialization. It holds the scale of the activations constant however deep the network is, and it is derived rather than tuned: the layer's own width supplies the number.

Xavier assumes the layer is linear, and a ReLU is not. About half of the outputs of any layer are set to zero, which halves the variance of what the next layer receives, and over many layers a factor of one half per layer is exactly the geometric collapse the derivation was supposed to prevent. He et al. — the same group that produced ResNet — corrected for it by doubling the variance:

$$\sigma_w^2 = \frac{2}{D_{\text{in}}}. \tag{7.11}$$

This is Kaiming initialization, and it is the default for convolutional networks. For a convolutional layer D_{in} is the number of inputs to one filter, $k^2 C_{\text{in}}$, not the number of channels. For the six-layer example above it prescribes $\sqrt{2/4096} \approx 0.022$, between the two values that failed, and the activation histograms hold their shape from the first layer to the last.

It is worth being clear that the whole problem is a symptom rather than a disease, and that normalization layers address the same symptom from the other side. A network with a normalization layer after every convolution is far less sensitive to how it was initialized, because any drift in scale is corrected at the next layer rather than compounding. The two techniques were developed in parallel and both remain in use: initialization gets the first forward pass into a sane range, normalization keeps it there.

7.10. Preprocessing

The same argument applies once more at the input, where the data are not something the network produced and cannot be corrected by a layer that has not run yet. Raw pixels arrive in $[0, 255]$, all positive and all of comparable magnitude, which is a poor input distribution for the reason Section 7.9 just gave — a layer receiving inputs with a large mean produces outputs dominated by that mean, and the gradients with respect to a row of W then all share a sign.

The convention is to center and scale per channel: compute the mean and standard deviation of each of the three colour channels over the training set, then subtract and divide. That is three numbers of each kind, applied identically at every spatial position, which is why the operation costs nothing and is folded into the data loader. Models pretrained on ImageNet ship with the statistics of that dataset baked into their expected input — torchvision's classification models assume a mean of $(0.485, 0.456, 0.406)$ and a standard deviation of $(0.229, 0.224, 0.225)$ on pixels already scaled to $[0, 1]$ — and applying a different normalization to an image at inference is a common and quiet source of degraded accuracy.

Two points about method rather than arithmetic. The statistics must be computed on the training split alone and then applied unchanged to validation and test data; computing them over the full dataset leaks information about the test set into training, in a small way that is nonetheless indefensible. And the transformation is fixed once and never re-estimated, which is what makes it a property of the model rather than of the batch — the distinction Section 7.2 drew between layer and batch normalization, arriving here for the same reason.

7.11. Noise you add and then average away

Section 7.3 described a layer that behaves differently during training and testing, and the shape of that difference is worth extracting because several unrelated-looking techniques have it. During training, inject randomness into the computation. During testing, remove the randomness and use the average of what it would have produced. The network is being trained not on a fixed function but on a distribution of functions, and asked to do well in expectation, which is why it cannot depend on any accident of one of them.

Dropout is the version that randomizes the network. Data augmentation is the version that randomizes the data, and it is the more important of the two in vision. Rather than train on each image once, apply a random transformation to it every time it is drawn: the label is by assumption unchanged, so the model sees a much larger effective training set and is forced to produce the same answer across the variation the transformation spans. Every augmentation is therefore a claim about which changes to an image do not change what it is, and choosing them is choosing what invariances to demand.

The standard ones are few. Horizontal flips are nearly free and nearly always correct for natural photographs, since a cat facing left is a cat; vertical flips are usually wrong for the same reason, and flips of any kind are wrong for text or digits, where a mirrored 2 is not a 2. Random cropping and rescaling does the most work: ResNet's training procedure samples a length L uniformly from $[256, 480]$, resizes the image so its short side is L, and takes a random 224×224 patch, which varies both the scale and the framing of the object on every epoch. Colour jitter perturbs brightness, contrast and saturation, on the claim that the identity of an object does not depend on the lighting it was photographed under. Cutout zeroes a random rectangle of the image outright, forcing the classifier to work from partial evidence rather than one decisive region — the same argument dropout makes about features, transposed to pixels. It helps substantially on small datasets like CIFAR-10 and much less on ImageNet, which is the general pattern for regularization: the less data there is, the more of it is worth adding.

The test-time half of the pattern is usually just "do nothing", since the identity transformation is the natural centre of the distribution. It need not be. ResNet's reported ImageNet numbers use test-time augmentation, evaluating each image at five scales with ten crops each — four corners, the centre, and horizontal flips of all five — and averaging the fifty sets of scores. That is the averaging step performed explicitly rather than approximated, and it buys a fraction of a point for fifty times the inference cost, which is why it appears in competition results and almost nowhere else.

Batch normalization belongs to this family too, and it is a nice check on whether the pattern has been understood. Its training-time output depends on which other images share the batch, which is a source of randomness; its test-time output uses running averages accumulated over training, which is the averaging step. That is precisely the dropout structure, arrived at accidentally by a technique designed for something else entirely — and it explains the observation, otherwise puzzling, that a network with batch normalization often needs less explicit regularization than one without.

The practical reading of all this is a dial rather than a checklist. Add regularization when the training accuracy is far above the validation accuracy, remove it when the two are equal and both are low, and choose the specific mechanism by what invariance the problem actually has. Section 7.13 is where that judgement gets made.

7.12. Transfer learning

Everything so far has assumed a million labelled images. Almost no real problem has one, and the reason convolutional networks are nonetheless usable on problems with a few thousand examples is the observation Section 6.6 began: the features a network learns are far less specific to its training task than the task would suggest. AlexNet's first layer, trained to separate a thousand categories of photograph, contains oriented edges and colour-contrast patches — structures that are useful for any problem involving natural images and were not chosen for ImageNet. The same holds, less obviously, several layers up. Take the activations of the penultimate layer of a trained network as a feature vector and find nearest neighbours in that space, and images land near each other when they depict similar things, including things the network was never trained to name.

The procedure that follows is short. Train a network on a large dataset — or download one, since the pretrained weights for every standard architecture are a line of code away. Discard the final classification layer, which is the only part specific to the original label set. Replace it with a fresh layer of the right output size for the new problem, and train.

What varies is how much of the rest is allowed to move, and the answer depends on two things: how much data the new problem has, and how much it resembles the old one. With very little data and a similar domain, freeze everything and train only the new final layer. The pretrained network is then a fixed feature extractor and the trainable model is a linear classifier on its output — cheap, fast, and hard to overfit, since there are few parameters to overfit with. This is the setting in which DeCAF and related work found that features from a network trained on ImageNet, with a linear classifier on top, beat purpose-built systems on tasks the network had never seen. With a similar domain and a substantial dataset, initialize from the pretrained weights and fine-tune the whole network at a learning rate perhaps a tenth of what would be used from scratch — small enough that the existing features are refined rather than destroyed in the first few steps, while the freshly initialized final layer produces large gradients.

The remaining two cases are less tidy. A large dataset in a different domain — medical scans, satellite imagery, microscopy — can be fine-tuned end to end or trained from scratch, and which wins is an empirical question with no reliable prior. A small dataset in a different domain is the genuinely hard case: there is not enough data to train and not enough similarity for the transfer to carry, and the useful move is to find a pretrained model closer to the target domain rather than to accept ImageNet and hope. There is no technique here that always works.

Between freezing the last layer and fine-tuning everything lies a continuum, and much of the interesting recent work is in it. Unfreezing the last few blocks is the obvious intermediate. More sophisticated is LoRA, which fine-tunes every layer but constrains the update to each weight matrix to be low-rank, so that a small number of new parameters expresses a change to a large number of old ones — structurally the trick of Equation 7.9 applied to weights instead of activations, learning the difference from the pretrained model rather than the model.

The framing worth taking away is not that transfer learning is a workaround for insufficient data. Pretraining on a large corpus and adapting to the task at hand is the default way models are built now, in vision and elsewhere; training a network from random initialization on a

task-specific dataset is the special case, reserved for when the dataset is enormous or genuinely unlike anything already trained on.

7.13. Choosing hyperparameters

The architecture, the initialization, the regularizers, the learning rate and its schedule from Section 4.11 — all of it has to be set, and there are more free choices here than a search can cover. What makes it tractable is an order, in which the cheap diagnostics run first and each step assumes the previous one passed.

Check the initial loss before anything else. A softmax classifier over C classes at initialization should produce roughly uniform scores, and therefore a loss of $\log C$ — about 2.30 for CIFAR-10 and 6.91 for ImageNet. Disable regularization and run one forward pass. If the number is not close, something is wrong in a way no amount of tuning will fix, and the second before it takes to check is the cheapest debugging in deep learning.

Then overfit a tiny sample. Take a handful of training examples — a dozen, or even one — turn off all regularization, and train until the loss reaches zero. A model that cannot memorize twelve images cannot learn a million, and failure here is nearly always a bug rather than a hyperparameter: a mismatch between predictions and labels, a detached gradient, a learning rate off by orders of magnitude. This is the standard first move when training does not work, and it also gives a rough sense of which learning rates move the loss at all.

With that passing, find a learning rate on the full training set. Turn on a small weight decay and try the usual ladder — 10^{-1} down to 10^{-5} — asking only which value drops the loss substantially within about a hundred iterations. This is not a search for the best learning rate but for the right order of magnitude, and it is fast because it does not require training to completion.

Only now is a search worth running: a coarse grid over the few hyperparameters that matter, each configuration trained for one to five epochs, then a refined range around whatever region looked best, trained longer. And when the search covers more than one dimension, sample the configurations randomly within their ranges rather than laying them out on a grid. The argument, from Bergstra and Bengio, is that the hyperparameters are almost never equally important. A grid of nine points over two parameters tries only three distinct values of each, so if one of them barely matters, six of the nine runs are re-measuring the same three values of the one that does. Nine random points try nine distinct values of every parameter. The advantage grows with the number of dimensions and with how unequal the importances are, which in practice is very.

The last step of the loop is to look at the curves, and the information is almost entirely in the gap between them. Both still rising means the run was stopped too early: train longer before concluding anything. A large and widening gap, with training accuracy climbing while validation accuracy flattens or falls, is overfitting, and the response is more regularization or more data — the dial of Section 7.11. No gap at all, with both curves low, is underfitting, and calls for the opposite: less regularization, a longer run, or a larger model. A loss that falls sharply and then plateaus at a poor value usually means the learning rate is too large for the current stage, which is what the schedules of Section 4.11 exist to address. Then return to the search with what was learned, and repeat.

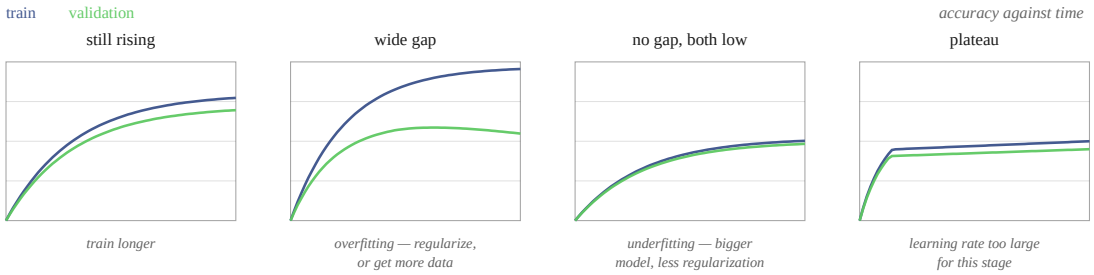


Figure 7.5.: The four shapes worth recognising in a training run, and what each one asks for. Training and validation accuracy are plotted against time. Only the last is a healthy stopping point; the others each have a specific remedy, and the diagnosis is in the gap between the two curves rather than in either curve alone. *Drawn for these notes.*

7.14. What this leaves

Two chapters ago there was an operator. There is now a network that can be specified end to end and trained to a competitive result, and it is worth noticing how little of what this chapter added is about images. The convolution encodes something true about pictures. Normalization, dropout, careful initialization, augmentation, transfer learning, and the hyperparameter protocol encode nothing of the sort: they are what it takes to optimize a deep composition of functions, and they transfer to architectures that have no convolutions in them at all. The residual connection of Equation 7.9 and the layer normalization of Equation 7.2 are in every transformer, unchanged.

What the architectures established is narrower and more specific. VGG showed that a network can be specified by a rule rather than designed layer by layer, and that small filters stacked deep beat large filters used shallowly. ResNet showed that the limit on depth encountered in 2014 was a failure of optimization rather than of representation, and removed it. Neither result says anything about what the convolution should be, and both are about how to build with it — which is why the second one survived the convolution's decline. When the field moved to attention, it kept the skip connections.

The gap that remains is the one Section 7.6 opened and did not close: the classifier at the end of a convolutional network produces one label for a whole image, and almost nothing anyone wants from computer vision has that form. Where in the frame is the object, how many are there, which pixels belong to it, and what happens when the answer must be a sequence rather than a class. Those questions need a different output structure, and in some cases a different architecture entirely.

References

- J. L. Ba, J. R. Kiros and G. E. Hinton, "Layer Normalization," 2016. arXiv:1607.06450
- J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *JMLR* 13:281–305, 2012. Paper
- T. DeVries and G. W. Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout," 2017. arXiv:1708.04552

- J. Donahue et al., "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition," *ICML*, 2014. arXiv:1310.1531
- X. Glorot and Y. Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," *AISTATS*, 2010. Paper
- K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *ICCV*, 2015. arXiv:1502.01852
- K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *CVPR*, 2016. arXiv:1512.03385
- J. Hu, L. Shen and G. Sun, "Squeeze-and-Excitation Networks," *CVPR*, 2018. arXiv:1709.01507
- S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *ICML*, 2015. arXiv:1502.03167
- A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *NeurIPS*, 2012. Paper
- P. Ramachandran, B. Zoph and Q. V. Le, "Searching for Activation Functions," 2017. arXiv:1710.05941
- A. S. Razavian et al., "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition," *CVPR Workshops*, 2014. arXiv:1403.6382
- O. Russakovsky et al., "ImageNet Large Scale Visual Recognition Challenge," *IJCV* 115:211–252, 2015. arXiv:1409.0575
- S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, "How Does Batch Normalization Help Optimization?," *NeurIPS*, 2018. arXiv:1805.11604
- K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *ICLR*, 2015. arXiv:1409.1556
- N. Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *JMLR* 15:1929–1958, 2014. Paper
- C. Szegedy et al., "Going Deeper with Convolutions," *CVPR*, 2015. arXiv:1409.4842
- Y. Wu and K. He, "Group Normalization," *ECCV*, 2018. arXiv:1803.08494

8. Recurrent Neural Networks

Lecture 7

Based on Lecture 7 of CS231n, Stanford University, Spring 2025.

8.1. Problems with a shape

Every network in the preceding chapters commits to a shape when it is built. A convolutional classifier accepts an image of fixed height and width and emits a vector of fixed length, one score per class, and both of those numbers are decided before a single weight is learned. That commitment is not a limitation anyone had to work around, because the problem was itself shaped that way: classification takes one image and returns one label.

A great many problems are not shaped that way, and they fail the commitment in different places. Captioning takes a single image — a fixed-size input, no trouble there — and must return a sentence, whose length is not known until the sentence is finished. Classifying an action in a video is the mirror image: the input is a run of frames of no fixed number, and the output is a single label. Translation varies at both ends and, worse, varies at the two ends independently, since a sentence and its translation need not have the same number of words. Labelling every frame of a video with what is happening in it varies at both ends too, but in lockstep, one output for each input.

Figure 8.1 draws the five patterns. It is tempting to read them as five architectures, and they are not; they are five ways of attaching inputs and losses to one machine. The machine has to consume input incrementally and produce output incrementally, and once it does, whether a particular input step carries a real vector or a placeholder, and whether a particular output step is scored or discarded, is a decision made per problem rather than per model. This chapter develops the machine on the rightmost pattern, where every step has both an input and a scored output, because that is where the mechanics are least cluttered. The others follow by changing what is attached where.

Note what the shapes do *not* distinguish. Nothing here says the input steps are moments in time. They are for video and for speech, but for a sentence the steps are positions in a string, and for a program they are tokens in a file; the ordering is real and the metric structure is not. "Time step" is the conventional name and this chapter uses it, but the only property the model actually relies on is that the steps come in an order and that order carries meaning.

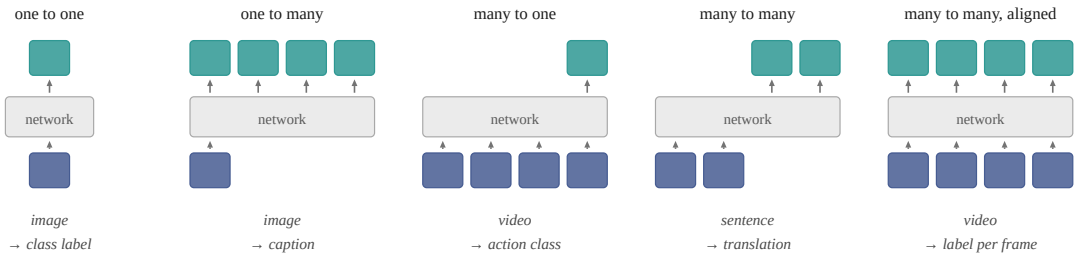


Figure 8.1.: The five ways a problem can be shaped, and where each one breaks the fixed-size commitment. Indigo blocks below are inputs, teal blocks above are outputs, and the grey band between them is the network. Only the first is served by a feedforward network; the other four need a model that can consume and emit one step at a time. The rightmost two both vary at input and output, but the last aligns them one to one, which is why it is the easiest to reason about. *Drawn for these notes.*

8.2. A state that persists

If the network is to read a sequence one element at a time and its behaviour at step t is to depend on what came before, then something must survive between steps. Call it the hidden state h, a vector the network carries forward and rewrites at each step. Then the whole model reduces to a single line:

$$h_t = f_W(h_{t-1}, x_t) \tag{8.1}$$

where x_t is the input at step t, h_{t-1} is the state left over from the step before, and f_W is a function with parameters W. This is the recurrence, and it is the entire idea. Everything else in this chapter is a choice of f, a choice of where to attach losses, or a consequence of the fact that Equation 8.1 is applied over and over.

The subscript on f_W deserves attention, because there is only one of them. The same function with the same parameters runs at step 1, at step 40 and at step 4000. This is not a simplification adopted for convenience; it is what makes the model well defined at all. A network with a distinct W_t per step would have a parameter count that grows with sequence length, would need to be rebuilt for every new length, and — decisively — could not apply anything it learned at step 5 to step 500, because the weights that fire at step 500 would never have been touched by a training sequence of length 40. Sharing the weights across time is the same argument that made a convolutional filter slide across an image in Section 6.11, moved from space to order: a pattern worth detecting is worth detecting wherever it occurs, so the detector should not be indexed by where it looks.

The state is not the output. It is an internal quantity, and turning it into whatever the task actually wants is a second function with its own parameters:

$$y_t = f_{W_{hy}}(h_t). \tag{8.2}$$

Two things are happening in Equation 8.2 and they are worth separating. The obvious one is dimensional: h_t lives in whatever space the designer chose for it, typically a few hundred dimensions, and y_t has to live in the space the task defines — one score per class, or one score per word in a vocabulary. The less obvious one is that the map is *learned*, so it is not merely a projection into the right shape but a transformation the network can tune. The division of

labour is that f_W decides what is worth remembering and $f_{W_{hy}}$ decides how to read it out, and those are different jobs.

Fixing the two functions to the simplest thing that could work gives the model that everyone means by *vanilla RNN*, and, after the psycholinguist who introduced this architecture in 1990 (Elman), the Elman network:

$$h_t = \tanh(W_{hh}\, h_{t-1} + W_{xh}\, x_t)\,, \qquad y_t = W_{hy}\, h_t. \tag{8.3}$$

Three weight matrices, each doing one job. W_{xh} maps an input vector into the hidden space; W_{hh} maps the previous hidden state into the hidden space; the two results are added, and a nonlinearity is applied to the sum. W_{hy} maps the result out. Bias terms are omitted here and are present in every implementation.

The choice of tanh rather than the ReLU that Section 7.4 settled on elsewhere is deliberate and specific to recurrence. In a feedforward network an activation is applied a fixed number of times, once per layer, and if ReLU lets values grow that growth is bounded by depth. Here the same nonlinearity is applied once per step, and the number of steps is a property of the data. A function whose output is bounded to $[-1, 1]$ keeps the state in a fixed range no matter how long the sequence runs, which an unbounded activation does not; that it is also zero-centred, and so can represent a quantity being pushed in either direction, is the second reason. Both of these will look different by the end of the chapter — bounding the state turns out to cost something on the backward pass — but the reasoning is sound as far as it goes.

8.3. A network built by hand

The equations above are short enough to hide how little is going on inside them. It is worth doing one recurrence completely, with numbers, and the clearest way is to abandon learning entirely and set the weights by hand for a task small enough to solve in one's head.

The task: read a sequence of zeros and ones, and emit 1 at any position where the current input and the previous input are both 1, and 0 everywhere else. This is the aligned many-to-many shape — one output per input — and it is the smallest problem that genuinely requires memory, since the correct output at step t is not a function of x_t alone.

Start by asking what the hidden state has to hold. The output at step t depends on x_t and on x_{t-1}, and the state is the only channel through which anything from the past can arrive, so the state must carry x_{t-1}. Since the output is computed from the state alone, it must carry x_t as well. Two numbers, then; take three, laying out the state as

$$h_t = \begin{pmatrix} \text{current} \\ \text{previous} \\ 1 \end{pmatrix}, \qquad h_0 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{8.4}$$

The constant 1 in the third slot is a bias smuggled into the state vector so that the readout can subtract a threshold without a separate bias term, and the initial state asserts that the sequence is preceded by zeros. Replace tanh with a ReLU, purely so every quantity below is an integer.

Now build the two matrices. The input contributes to the state through W_{xh}, which here maps a scalar to three dimensions and needs to place x_t into the first slot and nothing anywhere else:

$$W_{xh} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad W_{xh}\, x_t = \begin{pmatrix} x_t \\ 0 \\ 0 \end{pmatrix}. \tag{8.5}$$

The old state contributes through W_{hh}, which has to do three separate things at once, one per row. The first row must be zero, because the new "current" slot comes entirely from the input and must not inherit anything. The second row must copy the old first slot into the new second slot — last step's current becomes this step's previous. The third row must preserve the constant. Writing that down:

$$W_{hh} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad W_{hh} \begin{pmatrix} c \\ p \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ c \\ 1 \end{pmatrix}. \tag{8.6}$$

Adding the two contributions and applying the ReLU, which changes nothing because every entry is already non-negative, gives $h_t = (x_t,\ c_{t-1},\ 1)^\top$ — exactly the layout Equation 8.4 asked for, with the state shifted along by one position. The two matrices have divided the work cleanly: W_{hh} carries information forward from the past, and W_{xh} injects the present. That division is not an artefact of the toy problem. It is what those two matrices are always doing, in every RNN, however large.

The readout is a single row, since the output is one number:

$$W_{hy} = \begin{pmatrix} 1 & 1 & -1 \end{pmatrix}, \qquad y_t = \mathrm{ReLU}\big(W_{hy} h_t\big) = \max(c_t + p_t - 1,\ 0). \tag{8.7}$$

This is where the constant earns its slot: the -1 multiplies it, and the expression $c + p - 1$ is positive only when c and p are both 1, in which case it equals 1. Two ones give 1; a one and a zero give $\max(0, 0) = 0$; two zeros give $\max(-1, 0) = 0$. The ReLU has been turned into a logical AND.

Run it on $0, 1, 0, 1, 1, 1, 0, 1, 1$ and the states go $(0, 0, 1)$, $(1, 0, 1)$, $(0, 1, 1)$, $(1, 0, 1)$, $(1, 1, 1)$, $(1, 1, 1)$, $(0, 1, 1)$, $(1, 0, 1)$, $(1, 1, 1)$, and the outputs $0, 0, 0, 0, 1, 1, 0, 0, 1$, which is the answer. The network is nine parameters and detects a pattern no fixed-size classifier over a single input could.

Two things about this example are honest and one is not, and it is worth being clear which is which. The mechanics are exactly those of a real RNN: the same matrices multiply the same way, and a hidden state of three hundred dimensions runs the identical computation. What is unreal is the provenance of the weights. Nobody finds W_{hh} by staring at the problem; a real network is trained, and a hidden state large enough to be useful holds distributed quantities that correspond to no slot anyone named. Getting there requires computing gradients through a computation whose length is a property of the input, which is the subject of the next section.

8.4. Unrolling

The loop in Equation 8.1 is a loop only in the way it is written. Executed on a sequence of T elements it produces a chain of T applications of f_W, each consuming the state the last one produced, and that chain is an ordinary directed acyclic graph of the kind Section 5.8 handled. Drawing it that way is called *unrolling*, and it is what makes the model differentiable

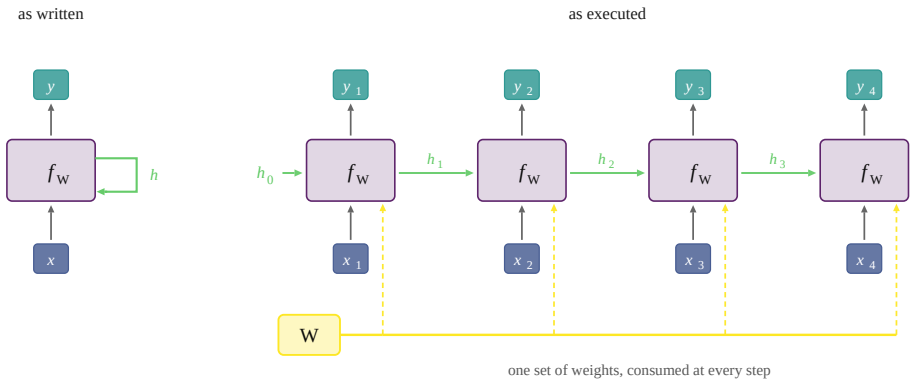


Figure 8.2.: The recurrence as written and the recurrence as executed. On the left, the loop notation, with the arrow that feeds the state back into the block; on the right, the same computation unrolled over four steps. The single box W at the bottom, with an edge into every application of f_W, is the whole difference between this graph and a four-layer feedforward network: the same parameters are consumed four times, so on the backward pass four gradient contributions arrive at the same node and are summed. *Drawn for these notes.*

by machinery already in hand: there is no new rule for recurrent networks, only an old rule applied to a graph whose size depends on the input.

Figure 8.2 makes the one genuinely new thing visible. W is a single node with an edge running into every step, so it is a variable used T times, and the multivariable chain rule says that the gradient arriving at a node used several times is the sum of the contributions along each use:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L}{\partial W}\Big|_{\text{step } t} . \tag{8.8}$$

This is worth stating because it is the point at which people expect something exotic and get arithmetic instead. Backpropagation through a shared weight is the same operation as backpropagation through a residual connection or through any other fan-out in a graph: contributions add. One useful way to hold it is to pretend, while working backwards, that each step has its own private copy $W_1, \dots, W_T$, compute the T gradients independently, and only at the end remember that all T copies are the same array and add them together. Applied to a recurrent network this procedure has a name of its own, *backpropagation through time*, though it introduces no rule that Section 5.11 did not already contain.

Where the loss enters depends on which of the shapes from Section 8.1 is in play, and the four cases differ only in what is attached to the graph rather than in the graph itself. In the aligned many-to-many case every step produces an output, every output is scored, and the total is

$$L = \sum_{t=1}^{T} L_t, \tag{8.9}$$

with each L_t typically a cross-entropy against the target at that position. Because the sum is over independent terms, the gradient of L decomposes into one contribution per step before Equation 8.8 ever gets involved.

Many-to-one is the case with a single loss at the end, and it is the more instructive of the two. Only h_T feeds the output, so the intermediate hidden states receive no gradient of their own; everything that reaches step 5 has travelled backwards from step T along the chain of state transitions. The states are still on the path, and they still get gradients, but those gradients are entirely upstream in origin. It is worth noting that using h_T alone is a choice, not a requirement: for video classification the frames near the beginning may carry as much evidence as the frames near the end, and pooling the hidden states across time before the readout is a common and often better alternative to reading only the last one.

One-to-many exposes a small gap in Equation 8.1. The recurrence needs an x_t at every step because $W_{xh}x_t$ is a term in it, but a captioning model has exactly one real input, the image, and $T-1$ steps with nothing to consume. Two answers are in use. Feeding zeros is the honest admission that there is no input; feeding the model's own previous output, $x_t = y_{t-1}$, is the more interesting one, because it makes the model autoregressive — each emitted word becomes the next input, so the sentence being generated conditions its own continuation. Section 8.6 develops that idea, and Section 8.8 uses it.

8.5. Truncated backpropagation

Equation 8.8 is correct and, on a long sequence, unaffordable. The backward pass needs the activations computed on the forward pass, so every hidden state, every input, and every intermediate quantity at every step must be held in memory until the gradient comes back for it. Memory therefore grows linearly in T, and nothing about the model bounds T: a book, a genome, or a month of sensor readings is a single sequence, and the forward pass runs to the end of it before any gradient can be computed at all. The practical failure is running out of GPU memory, and it arrives long before the sequence lengths anyone actually cares about.

The fix is to stop being exact. Cut the sequence into chunks of some fixed length — a few dozen steps, chosen to fit — and treat each chunk as a training example in its own right: run forward through the chunk, compute the losses in it, backpropagate to its start, apply the update, and discard everything. Then move to the next chunk.

What makes this more than simply chopping the sequence up is the asymmetry drawn in Figure 8.3. The hidden state at a chunk boundary is carried forward and used to initialize the next chunk, so the forward pass never restarts and the model's state at step 5000 still reflects step 1. The *gradient* is not carried across the boundary; it stops at the first step of the chunk it belongs to. Forward, the sequence is continuous and arbitrarily long; backward, it is a series of short independent problems.

The cost is specific and worth stating plainly rather than filing under "approximation". A dependency that spans more steps than the chunk length is not learned poorly; it is not learned at all. If the chunk is forty steps and the answer at step 100 depends on something at step 20, no gradient ever connects them, and no amount of training will make the model use that connection. Truncation does not blur long-range dependencies, it deletes the gradient signal for them, and the model retains only whatever such structure it can pick up indirectly through the state it inherits.

There is a second, milder cost. Applying an update per chunk rather than one update over the whole sequence means the weights used in chunk three are not the weights that produced the state that chunk three inherited, so the computation being differentiated is not quite the computation that was run. This is the same inconsistency that appears in distributed training, where gradients computed independently on several devices are applied to one set of weights,

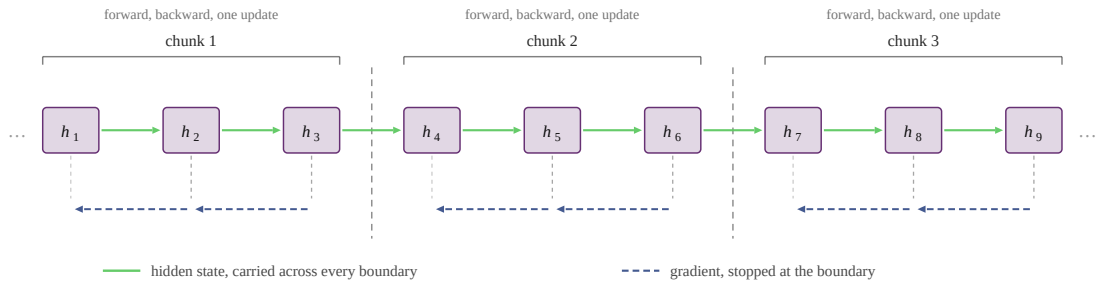


Figure 8.3.: Truncated backpropagation through time. The sequence runs left to right and is cut into chunks; within each chunk the forward pass and the backward pass are complete, and a parameter update is applied at the end of it. At the boundary the hidden state crosses (solid) but the gradient does not (dashed), so memory is bounded by the chunk length rather than by the length of the sequence. The cost is directly visible: nothing in the picture lets a loss inside the third chunk influence how the first chunk's inputs are processed. *Drawn for these notes.*

and it is tolerated for the same reason: the updates are small, and the alternative does not fit in memory.

8.6. One character at a time

The task that makes recurrence pay is deceptively small: given the text so far, predict the next character. Fix a vocabulary — for a worked example take just $\{\texttt{h}, \texttt{e}, \texttt{l}, \texttt{o}\}$ and the training string `hello` — and the problem becomes aligned many-to-many classification. At each position the model sees one character and must produce a distribution over the vocabulary for the next one, so the input at step t is the character at position t and the target is the character at position $t+1$.

Nothing about that requires a labelled dataset, and this is the property that matters more than any architectural detail in this chapter. The label at every position is simply the next character, which the text already contains. Any body of text is training data for this task the moment it is written down, and the reason language models became what they are is that this removes the annotation bottleneck entirely — the supervision is free, so the only limit is how much text exists.

Mechanically, each character enters as a one-hot vector $x_t \in \mathbb{R}^V$: a single 1 in the position of that character and zeros elsewhere, where V is the vocabulary size. The product $W_{xh}x_t$ then selects one column of W_{xh} and discards the rest, which means the matrix multiply is a table lookup written as arithmetic. Implementations do the lookup directly instead, and the matrix is called an *embedding* — a learned D-dimensional vector per vocabulary entry, held in a $D \times V$ table. Two things are gained. The multiply is skipped, which matters when V is in the tens of thousands. More interestingly, since the vectors are learned, characters or words that behave alike come to sit near one another, so the model is handed a representation with structure rather than V mutually orthogonal symbols with none.

Out the other end, y_t is a vector of V scores, one per vocabulary entry, turned into a distribution by the softmax of Section 3.6 and scored by the cross-entropy of the same section:

$$p_t = \operatorname{softmax}(y_t), \qquad L_t = -\log p_t[x_{t+1}]\,, \tag{8.10}$$

with $p_t[x_{t+1}]$ the probability the model assigned to the character that actually came next. The total loss over the sequence is Equation 8.9. There is no new machinery here at all: a character-level language model is a V-way classifier applied once per position, and the recurrence is what lets the classifier at position t see everything before it.

Generation is where the model stops resembling a classifier. Feed it a starting character, read the distribution it produces, choose a character from that distribution, feed that choice back in as the next input, and repeat. The loop terminates when the model emits a designated end-of-sequence token, which is why vocabularies carry `<START>` and `<END>` entries that correspond to no actual character.

The choice at each step is worth dwelling on, because taking the most probable character — *greedy decoding* — is both the obvious thing to do and usually wrong. A model with fixed weights and a fixed starting character will make the same greedy choice every time, so it produces exactly one output sequence, forever. Sampling from the distribution instead makes generation stochastic, and it also stops the model from committing to a locally attractive character that leads somewhere with no good continuation. That greedy decoding can be beaten by taking a *less* likely character now is the entire premise of beam search, which keeps several candidate sequences alive and scores them by the probability of the whole sequence rather than of the next symbol. How to sample from a language model remains an active research question and is not settled by anything in this chapter.

One asymmetry between training and generation is easy to miss and has a name. During training, the input at step t is the true character from the corpus, regardless of what the model predicted at step $t-1$; during generation, the input at step t is whatever the model itself produced. The first is *teacher forcing*, and it is what makes training parallel over positions within a chunk and stable early on, when the model's own output is noise. But it means the model is only ever trained on inputs drawn from real text, and at generation time it is asked to continue from its own output — which, after a mistake, is text no corpus would contain. The model has no experience of recovering from its own errors because it was never allowed to make any. This mismatch is called *exposure bias*, and it is why a generated passage often drifts: each small error moves the input further from anything seen in training, which makes the next error more likely.

8.7. What the state ends up holding

Train the model of Section 8.6 on Shakespeare and watch the samples as training proceeds. Early on they are gibberish — the weights are near their initialization and the distribution is close to uniform over the vocabulary. Some way in, the character statistics arrive: the output has the letter frequencies of English, plausible-looking non-words, and spaces in roughly the right places. Later still it produces real words, then correct spelling and punctuation, then the *layout* of a play — a name on its own line, a colon, an indented speech. All of it from a model that has only ever been asked what character comes next, and that has been told nothing about words, let alone about speakers.

Train it on the Linux kernel source instead and the same thing happens in a different key. The output is C that would not compile but reads at a glance as C: braces open and close in matched pairs, indentation tracks nesting depth, comments begin with `/*` and end with `*/`, function definitions are followed by bodies. Syntax is a long-range constraint — a brace opened forty characters ago has to be closed — and the model is learning it from nothing but character co-occurrence.

The line from those experiments to current practice is short and worth stating without the mythology. Today's coding assistants are trained on the same objective on the same kind of data; what changed is that the unit of prediction became a token rather than a character, the architecture became the one Section 8.12 points toward, and the corpus and the compute grew by many orders of magnitude. The task did not change.

Because a hidden state is just a vector, one can take a single coordinate of it, run the model over a passage, and colour each character by that coordinate's value. Doing this systematically (Karpathy, Johnson and Fei-Fei, 2015) turns up cells with legible jobs. One switches on at an opening quotation mark and off at the closing one, holding a value across everything between — the model has allocated a coordinate to remembering that a quote is open, because it needs to know when a closing mark is due. Another decays steadily from the start of a line towards where a newline is likely, functioning as a position counter. In the code model there is a cell active inside `if` statements, a cell active inside comment blocks, and one whose value rises with each level of indentation and falls on the way back out.

This is a genuinely strong result and it is routinely overstated, so two qualifications belong with it. The first is that nobody specified those cells. There is no term in Equation 8.10 mentioning quotation marks; the pressure to predict the next character was sufficient, and the mechanism the network found is recognisably the one the hand-built network of Section 8.3 used — a coordinate reserved to carry one fact forward. The second is that most cells are not like this. Scan the hidden state and the large majority of coordinates produce colourings that look like noise, carrying distributed fragments of many quantities at once and corresponding to nothing nameable. The interpretable cells are the exception that got published, and reading them as evidence that the state decomposes into human-legible variables is reading too much into a favourable sample.

8.8. One network's output as another's input

Captioning is the one-to-many shape and the first place in these notes where two networks of different kinds are joined. The recipe follows directly from Section 7.12: take a convolutional network trained on ImageNet, discard the final classification layer, and keep the vector the penultimate layer produces. That vector is not a set of class scores — it is a description of the image in a few thousand dimensions, and it is exactly what the recurrent model needs to be told about.

The join is a third term in the recurrence. Writing v for the image vector, every step computes

$$h_t = \tanh(W_{hh}\, h_{t-1} + W_{xh}\, x_t + W_{ih}\, v)\,, \tag{8.11}$$

with W_{ih} a new learned matrix mapping the image into the hidden space. Note that v carries no subscript: the same image vector is supplied at every step, so the image conditions the whole sentence rather than only its beginning. The alternative, using v to initialize h_0 and letting it fade thereafter, is also in use and is the weaker of the two for the obvious reason.

The input at each step is the previous word, as Section 8.4 anticipated, which makes the caption autoregressive: the model conditions on the image and on what it has said so far. The sequence starts from a `<START>` token, and it stops when the model emits `<END>` — a token that exists only so that the model can decide how long the caption should be, which is the whole point of the one-to-many shape.

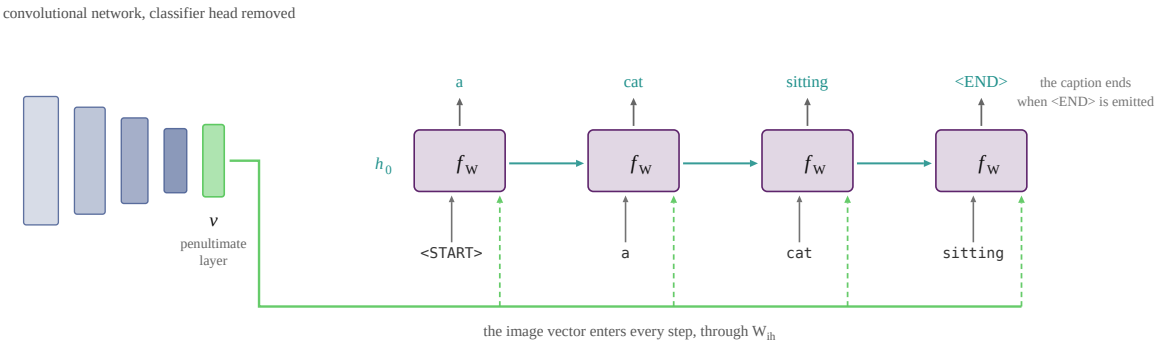


Figure 8.4.: Captioning as two networks joined at a vector. The convolutional stack runs once and its penultimate layer, rather than its class scores, becomes the image vector v; v then enters every step of the recurrence through W_{ih}, alongside the previous hidden state and the previously emitted word. Generation starts from `<START>` and ends when the model emits `<END>`, which is what allows the caption to be of a length the model chooses. *Drawn for these notes.*

These models worked, and their outputs from around 2015 (Karpathy and Fei-Fei; Vinyals et al.) still read as surprising: a cat sitting on a suitcase, a man riding a dirt bike on a dirt track. The failures are more instructive than the successes, because they fail in one characteristic way. A hand held palm-up around a flat object becomes "a person holding a computer mouse". A woman in a fur coat becomes a woman holding a cat. A person standing on a beach acquires a surfboard they are not carrying. A baseball player with a glove raised to receive a ball is described as throwing one.

Each of these substitutes what is common for what is present. The training objective in Equation 8.10 rewards the caption the corpus would most plausibly contain given this image, and in a corpus of captioned photographs beaches co-occur with surfboards, so a beach raises the probability of the word regardless of whether the object is in the frame. The model is not misperceiving the surfboard; it never needed to perceive one, because predicting it was the better bet. Nothing in the training signal distinguishes reporting from guessing, and so nothing forces the model to learn the difference. This is the same failure that vision-language models are still called out for a decade later, under the name hallucination, and its cause has not changed.

Two variations on the same join produce two further tasks. Feed the model a question alongside the image and score candidate answers and you have visual question answering; keep the dialogue history in the state and the model can be asked follow-up questions about the same image. Feed it an instruction and a stream of images that changes as it moves, and have it emit actions rather than words, and it becomes a navigation agent. At the time these were separate research problems with separate architectures; the observation worth carrying forward is that all of them are Equation 8.11 with a different choice of what conditions the recurrence and what the outputs are scored against.

8.9. Depth in the other direction

Everything so far has used one recurrent layer, and there is no reason to. Stacking gives a grid: the hidden states run along one axis in time and along the other in depth, with layer ℓ at step t taking its own previous state $h_{t-1}^{(\ell)}$ and the output of layer $\ell - 1$ at the same step. The first layer

reads the actual input; each layer above reads what the layer below produced. Recurrence stays within a layer — layer 2's state never depends directly on layer 1's state at an earlier step — and the vertical connections carry only the current step.

This helps, and it helps much less than depth helped in Section 7.5. Two or three layers is standard, four is unusual, and there is no recurrent counterpart to the hundred-layer networks of the previous chapter. The reason is that the unrolled graph is already deep: a sequence of length T puts T nonlinearities between the first input and the last output, so a two-layer network on a sequence of length 100 is differentiating through a graph of depth 200. Adding layers multiplies a depth that was never the scarce resource. Whatever limits an RNN, it is not that the computation between input and output is too shallow — which is precisely what Section 8.10 is about.

8.10. Why the gradient dies

Everything good about the recurrence comes from applying one function repeatedly. Everything wrong with it comes from the same place, and the mechanism is visible in a single derivative. Differentiating Equation 8.3 with respect to the state it consumed gives

$$\frac{\partial h_t}{\partial h_{t-1}} = \operatorname{diag}\left(\tanh'(W_{hh}h_{t-1} + W_{xh}x_t)\right) W_{hh}, \tag{8.12}$$

a diagonal matrix of activation derivatives multiplied by W_{hh}. Gradients flowing from step T back to step k pass through one such factor per step, and the chain rule multiplies them:

$$\frac{\partial h_T}{\partial h_k} = \prod_{t=k+1}^{T} \frac{\partial h_t}{\partial h_{t-1}}. \tag{8.13}$$

A product of $T-k$ matrices, every one of which contains the same W_{hh}. That is the whole problem, and it is a problem about repeated multiplication rather than about sequences.

To see what repeated multiplication does, delete the nonlinearity for a moment and suppose $\tanh'$ were identically 1. Then Equation 8.13 is W_{hh} raised to the power $T-k$, and the behaviour of a matrix power is governed by its largest singular value σ_1 — the most a unit vector can be stretched by one application. If $\sigma_1 > 1$, the norm grows geometrically and the gradient explodes. If $\sigma_1 < 1$, it shrinks geometrically and the gradient vanishes. Only σ_1 exactly equal to 1 is stable, and that is a knife edge no training procedure will balance on. There is no benign regime; a matrix applied a hundred times either amplifies or annihilates.

Restoring the nonlinearity breaks the symmetry, and not in the helpful direction. The derivative of tanh is at most 1, attained only at the origin, and falls away sharply on either side; any state that is not near zero contributes a factor well below 1. So the diagonal in Equation 8.12 scales the product down at every step, which drags even a W_{hh} with σ_1 slightly above 1 toward collapse. The property that made tanh attractive in Section 8.2 — that it keeps the forward state bounded — is exactly the property that starves the backward pass, and this tension is not resolvable by choosing a different bounded activation.

The two failures are not equally serious, which is worth being precise about because they are usually named as a matched pair. Explosion is loud and has a crude, effective fix. When the gradient's norm exceeds a threshold θ, rescale it:

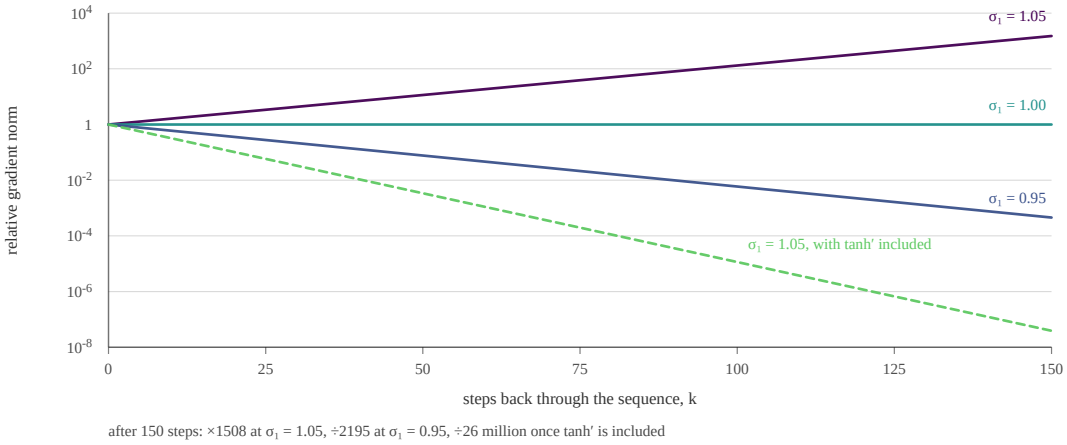


Figure 8.5.: What repeated multiplication does to a gradient. Each curve is the norm of a vector after k applications of a matrix with the largest singular value marked, on a logarithmic scale, so geometric growth and decay both appear as straight lines. At $\sigma_1 = 1.05$ the gradient is a thousand times its original size after 150 steps; at $\sigma_1 = 0.95$ it is a thousandth. The dashed curve adds the effect of $\tanh'$, whose value never exceeds 1 and is typically well below it: even the expanding matrix now decays. Nothing in the vanilla recurrence puts a curve near the horizontal. *Computed for these notes.*

$$g \leftarrow \frac{\theta}{\|g\|}\, g \qquad \text{whenever } \|g\| > \theta. \tag{8.14}$$

Gradient clipping (Pascanu, Mikolov and Bengio, 2013) keeps the direction and discards the magnitude, which is a defensible trade because a gradient of norm 10^6 was never a trustworthy step length in the first place. It is a one-line change and it works.

Vanishing has no such fix, and the reason is not that the gradient is small. Rescaling a small gradient upwards is easy; the difficulty is that Equation 8.8 sums contributions from every step, and the contributions from distant steps have decayed by a factor of σ_1^{T-k} relative to those from nearby ones. The total gradient is therefore not small — it is dominated by short-range dependencies, and the long-range signal is buried underneath them. Multiplying the sum by any constant multiplies both parts equally. The information about long-range structure has not been attenuated so much as overwritten, and it is unrecoverable after the fact.

That is why the honest conclusion is architectural rather than algorithmic. Learning long-range dependencies with a vanilla RNN would require W_{hh} to be, and to remain throughout training, a matrix whose repeated application neither grows nor shrinks its input — while also being the matrix that does the model's actual computational work. Those two demands are in direct conflict, and asking one matrix to satisfy both is the design error. The fix is to change the recurrence so that some path from h_{t-1} to h_t does not pass through a matrix multiply at all.

8.11. LSTM

The change was proposed in 1997, well before anything in this chapter had a use for it (Hochreiter and Schmidhuber), and it consists of carrying two vectors between steps instead of one. The *cell*

state c_t is the long-term memory and is not exposed outside the layer; the *hidden state* h_t is what the rest of the network sees. The point of the split is that the two can be updated by different mechanisms, and the cell's is chosen so that a gradient can travel along it undamaged.

One matrix multiply produces four vectors at once. Stack the previous hidden state and the current input, multiply by a single W of shape $4H \times (H + D)$, and split the $4H$-dimensional result into four blocks of size H:

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}. \tag{8.15}$$

Three blocks pass through a sigmoid and so lie in $(0, 1)$; these are the *gates*, and a value near 0 closes and near 1 opens. The fourth passes through tanh and lies in $(-1, 1)$; it is candidate content rather than a gate. Their jobs: f, the forget gate, decides how much of the existing cell to keep; i, the input gate, decides whether to write at all; g is what would be written; and o, the output gate, decides how much of the cell to expose. The stacking is a matter of implementation — four matrices concatenated into one so a single multiply computes all four — and the shapes above are what makes an LSTM roughly four times the parameters of a vanilla RNN with the same hidden size.

The state updates are two lines, and the first is the one that matters:

$$c_t = f \odot c_{t-1} + i \odot g, \qquad h_t = o \odot \tanh(c_t), \tag{8.16}$$

where $\odot$ is elementwise multiplication. Read Equation 8.16 as an instruction about memory: erase part of what is stored, scaled by f; write new content g, scaled by i; then reveal some fraction of the result, scaled by o. Each coordinate of the cell is governed independently, so the layer can hold one quantity untouched for hundreds of steps while rewriting another every step.

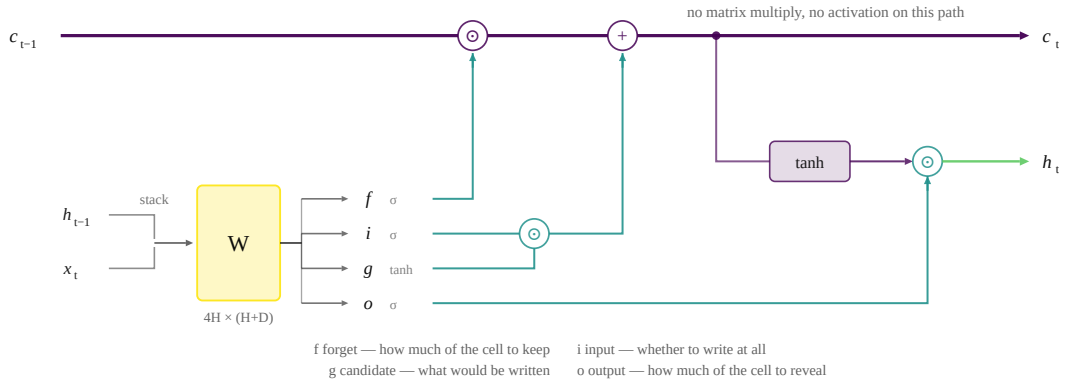


Figure 8.6.: One LSTM step. The cell state runs along the top and is modified only by an elementwise multiply with the forget gate and an addition of the gated candidate — no matrix multiply, no activation function stands on it. Everything below computes the four gates from one product of W with the stacked previous hidden state and current input. The hidden state that leaves on the right is a gated, squashed view of the cell, and is what the rest of the network sees. *Drawn for these notes.*

Now differentiate the cell update, which is the entire argument:

$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(f). \tag{8.17}$$

Compare it with Equation 8.12. There is no W_{hh} in it and no $\tanh'$ in it. A gradient travelling backwards along the cell state is multiplied by the forget gate and by nothing else, so the product in Equation 8.13 becomes a product of forget gates rather than a matrix power. The path along the top of Figure 8.6 is an uninterrupted channel from the end of the sequence to the beginning.

This is the residual connection of Section 7.8, arrived at independently and eighteen years earlier. With f near 1 the update reads $c_t \approx c_{t-1} + i \odot g$: a quantity carried forward unchanged plus a correction, which is Equation 7.9 with time in place of depth. The two mechanisms answer the same question — how does a signal cross many stages of processing without being destroyed — and both answer it by providing a route that skips the processing. The difference is what the stages are. ResNet stacks layers; the LSTM stacks time steps, and unlike layers, their number is set by the data.

Whether this *solves* vanishing gradients is the question the lecture asks and answers carefully, and the careful answer is no. The forget gate is a sigmoid, so $f < 1$ strictly, and a product of many numbers below 1 still decays. What has changed is who chooses. In a vanilla RNN the decay rate is a property of W_{hh}, one number for the whole model, applied identically to every coordinate and every step, and set by whatever the optimizer happened to need for the model's other work. In an LSTM the decay is per coordinate and per step, computed from the data, and learnable — so the network can drive f toward 1 on the coordinates holding something it will need later and toward 0 on the ones it wants to clear, and it can make that decision differently at different points in the sequence. Preserving information across a hundred steps went from something the model must arrange as a side effect of a single shared matrix to something it can simply ask for. The LSTM does not remove the mathematics of Section 8.10; it hands the model the controls.

8.12. What recurrence costs

Set the gradient problem aside for a moment and look at what the recurrence buys and what it charges, because the ledger is what determines the next chapter.

On the credit side, the model has no maximum input length. The state is the same size at step 10 and step 10 million, so the parameter count does not depend on how long the sequence is and there is no architectural quantity that has to be chosen in advance and then lived with. Compute grows linearly with the sequence: doubling the input doubles the work, exactly. Nothing about the model needs to be told the sequence length before it runs.

On the debit side, two things, and both are structural rather than incidental. The first is the bottleneck. Everything the model knows about the past is compressed into one fixed-size vector, and a vector of a thousand dimensions is the whole of the model's memory whether the sequence has been ten steps or a hundred thousand. Something must be discarded, and no amount of gating changes that — the LSTM improves how well the model can choose what to keep, not how much it can keep.

The second is sequential dependency, and it is the more damaging of the two in practice. Step t cannot begin until step $t-1$ has finished, because it consumes h_{t-1}. The sequence dimension

therefore does not parallelize, and the depth of the critical path in the computation is the length of the input. This is tolerable at generation time, where any autoregressive model must produce one symbol before the next regardless of architecture, but it is severe during training, where the targets at every position are known in advance and could in principle all be processed at once. A model that cannot exploit that cannot use a modern accelerator well, and cannot be scaled by throwing hardware at it — which, over the following decade, was how essentially all progress happened.

Both debits are removed by the mechanism of the next chapter, and it is worth saying now what it costs to remove them, because the trade is not free. Attention lets every position look directly at every other, so there is no fixed-size summary to squeeze through and no chain of dependencies to serialize; a whole training sequence is processed in one parallel operation. The price is that comparing every position with every other is quadratic in the sequence length, where the RNN was linear, and that the set of positions attended to must be bounded in advance — the context window that recurrent models do not have.

Which is why the recurrent idea has come back rather than expired. A line of recent work — Mamba, RWKV and the broader family of state space models — keeps the recurrent form, a state carried forward and updated in linear time with no context limit, while restructuring the update so that training can still be parallelized across the sequence. The equations look nothing like Equation 8.3, but the shape of the answer is the one this chapter has been describing: a fixed-size state, carried forward, rewritten at every step.

8.13. What this leaves

The chapter began with a shape problem — a network that fixes its input and output sizes cannot caption an image or classify a video — and answered it with one line, Equation 8.1, applied over and over. Everything else followed from that repetition. Unrolling made the model differentiable with machinery already built; sharing W across steps made it length-agnostic and made the gradient a sum; truncation made it fit in memory at the cost of long-range signal; and the repeated multiplication in Equation 8.13 made long-range learning hard in a way no choice of hyperparameter fixes. The LSTM is a direct answer to that last consequence and to nothing else.

What deserves to be carried forward is not the vanilla RNN, which is rarely the right choice today, but three things that outlived it. The first is that a shared weight used T times contributes T gradients that are summed — a fact about graphs, not about sequences, and one that reappears everywhere parameters are reused. The second is the additive path: whenever a signal must survive many stages, give it a route that skips them, which is the LSTM cell and the residual connection and the same idea twice. The third is the language-modelling objective of Section 8.6, which asks only for the next symbol, needs no labels, and turned out to be enough to induce syntax, structure, and rather more.

The next chapter takes up what Section 8.12 left open. If the difficulty is that information from step 5 must survive 500 sequential rewrites to be available at step 505, the sharper question is why it should have to travel through them at all.

References

- Y. Bengio, P. Simard and P. Frasconi, “Learning Long-Term Dependencies with Gradient Descent Is Difficult,” *IEEE Transactions on Neural Networks* 5(2):157–166, 1994. DOI

- J. L. Elman, “Finding Structure in Time,” *Cognitive Science* 14(2):179–211, 1990. DOI
- A. Gu and T. Dao, “Mamba: Linear-Time Sequence Modeling with Selective State Spaces,” 2023. arXiv:2312.00752
- S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation* 9(8):1735–1780, 1997. DOI
- A. Karpathy, J. Johnson and L. Fei-Fei, “Visualizing and Understanding Recurrent Networks,” 2015. arXiv:1506.02078
- A. Karpathy and L. Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions,” *CVPR*, 2015. Paper
- R. Pascanu, T. Mikolov and Y. Bengio, “On the Difficulty of Training Recurrent Neural Networks,” *ICML*, 2013. arXiv:1211.5063
- B. Peng et al., “RWKV: Reinventing RNNs for the Transformer Era,” *EMNLP Findings*, 2023. arXiv:2305.13048
- O. Vinyals, A. Toshev, S. Bengio and D. Erhan, “Show and Tell: A Neural Image Caption Generator,” *CVPR*, 2015. arXiv:1411.4555
- S. Antol et al., “VQA: Visual Question Answering,” *ICCV*, 2015. arXiv:1505.00468

9. Attention and Transformers

Lecture 8

Based on Lecture 8 of CS231n, Stanford University, Spring 2025.

9.1. One vector too few

Translation is the problem that forced the issue. An English sentence goes in and an Italian sentence comes out; the two need not have the same number of words, and where a word lands in one gives little warning of where its counterpart lands in the other. Both ends vary independently, which is the hardest of the five shapes of the previous chapter and the one recurrence was built for.

The architecture that answered it, from Sutskever, Vinyals and Le in 2014, uses two recurrent networks rather than one. The first, the encoder, consumes the source sentence one word at a time, running the recurrence $h_t = f_W(x_t, h_{t-1})$ over the input until it has been read. The second, the decoder, emits the translation one word at a time, running its own recurrence with its own weights. Between them sits a single vector, the context vector c, usually taken to be the encoder's final hidden state h_T. The decoder's update takes three arguments,

$$s_t = g_U(y_{t-1},\, s_{t-1},\, c) \tag{9.1}$$

where y_{t-1} is the word it produced last, s_{t-1} its own state, and c the summary of the source. Together with an initial decoder state s_0, also derived from the encoder, that is the whole model: read to the end, summarize, then generate until a stop token appears.

The arrangement has one structural defect, and it is visible in Equation 9.1 without running anything. The decoder consults c at every step and consults nothing else about the source. Whatever the encoder failed to pack into c is not merely hard to recover; it is gone. And c has a size fixed when the network is built — a thousand floats, say — which does not grow when the input does. For "we see the sky" a thousand floats is generous. For a paragraph it is thin, and for a chapter it is absurd: the model is being asked to compress an arbitrary quantity of information into a fixed allowance and then answer detailed questions about it. No amount of training fixes a channel that is too narrow, and no choice of c's width fixes it either, since whatever width is chosen is still a constant while the input is not.

The fix suggests itself once the defect is stated that way. Do not summarize once. Let the decoder look back at the whole input sequence each time it produces a word, and build whatever summary that particular word needs. Nothing then has to survive a lossy compression, and nothing has to be decided before the decoder knows what it is about to say.

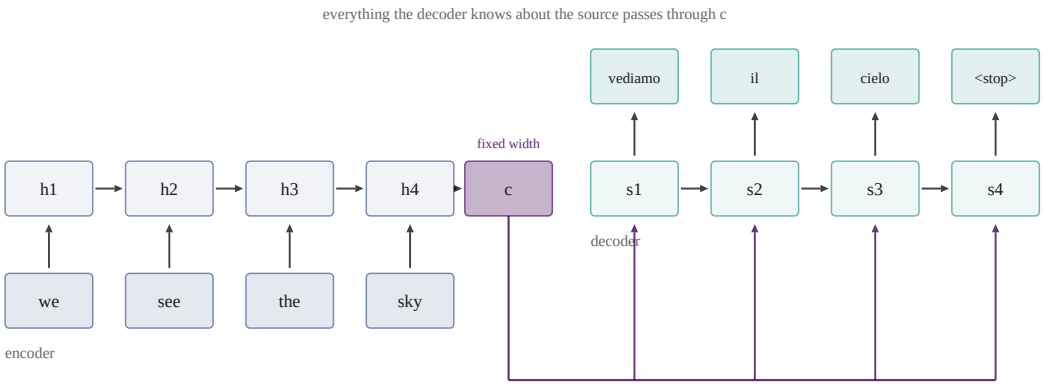


Figure 9.1.: The encoder–decoder model of Equation 9.1. Everything the decoder ever learns about the source sentence arrives through the single vector c, drawn in purple, whose width is fixed when the network is built and does not grow with the length of the input. *Drawn for these notes.*

9.2. Looking back

Turning that intent into a differentiable mechanism takes three operations, applied afresh at every decoder step. Bahdanau, Cho and Bengio introduced them in 2015 and the shape has not changed since.

Start with the decoder's current state s_{t-1}, which encodes what the decoder has said so far and therefore what it needs next. Compare it against each encoder state in turn to produce a scalar per input position,

$$e_{t,i} = f_{\text{att}}(s_{t-1},\, h_i) \tag{9.2}$$

where f_{att} is a small learned network — concatenate the two vectors, push them through a linear layer, read off one number. The number is an alignment score: how relevant input position i is to what the decoder is about to emit at step t. Nothing constrains it. It is a real number of arbitrary sign and magnitude, one per input position, and for a four-word source there are four of them.

Unbounded scores are awkward to combine, so the second operation is a softmax across the input positions,

$$a_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j} \exp(e_{t,j})} \tag{9.3}$$

which turns the scores into a distribution: every $a_{t,i}$ lies between zero and one and the whole set sums to one. What has been computed at this point is a probability distribution over positions in the source, conditioned on where the decoder stands. Call these the attention weights.

The third operation spends them. Take the encoder states and average them under that distribution,

$$c_t = \sum_{i} a_{t,i}\, h_i \tag{9.4}$$

which gives a context vector built for step t alone. If the weights concentrate on positions one and two, c_t is essentially h_1 and h_2 blended; if they spread evenly, c_t is the mean of everything. The decoder update is unchanged from Equation 9.1 — same three arguments, same recurrent unit — except that the constant c has become c_t, recomputed at every step from the same encoder states through the same f_{att}.

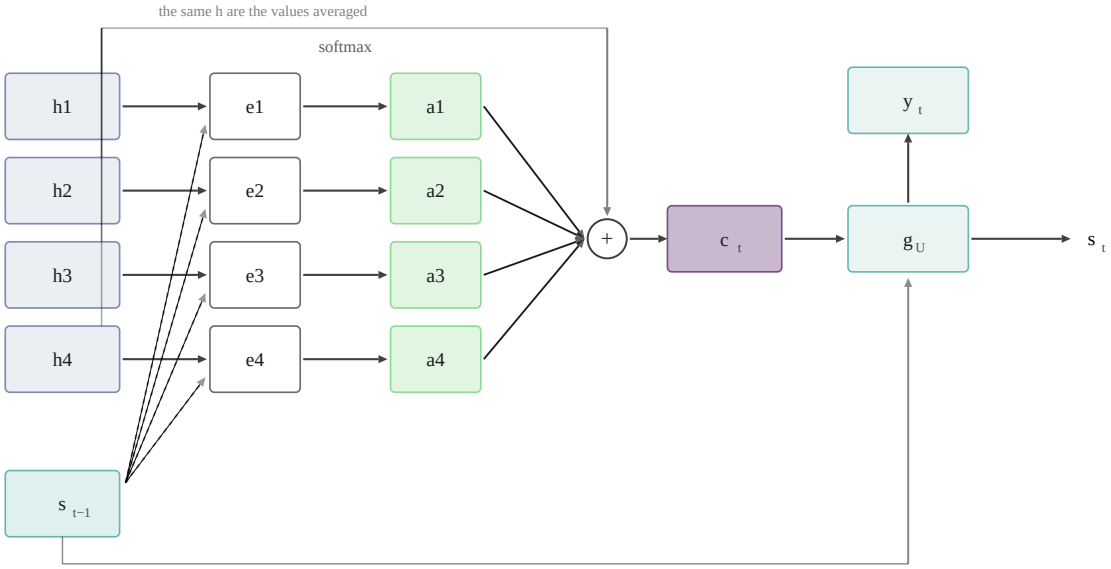


Figure 9.2.: One decoder step with attention. The decoder state s_{t-1} is compared against every encoder state to give scores $e_{t,i}$ (Equation 9.2), a softmax turns those into weights $a_{t,i}$ that sum to one (Equation 9.3), and the encoder states are averaged under those weights to give this step's context vector c_t (Equation 9.4). The same three operations run again, with the same f_{att} and different s, at every step of the output. *Drawn for these notes.*

The bottleneck is gone in the literal sense: there is no longer a fixed-size channel through which everything about the source must pass, because the decoder reads the encoder states directly and reads them anew each time. A source of a thousand words presents a thousand states, and the decoder is free to draw on any of them at any step.

What makes the mechanism work is not the intuition behind it but the fact that every operation in Equation 9.2 through Equation 9.4 is differentiable. Nothing supervises the weights. There is no annotated alignment in the training data, no signal that says *vediamo* covers *we see*; the only loss is the usual cross-entropy on the output tokens, and the gradient of that loss flows back through the weighted sum, through the softmax, into f_{att}, and on into the encoder. The model discovers that concentrating on the right source words makes the next word easier to predict, and it discovers this because concentrating on the wrong ones costs loss. Had the alignment needed supervising, the method would have died on the training data alone.

Two details of the recurrent framing are worth keeping straight, because they are easy to conflate. The decoder's *weights* are initialized the way any network's are, at random, and learned by gradient descent. The decoder's *initial state* s_0 is a different question, answered at the start of each sequence: sometimes the encoder's final state, sometimes a learned projection of it, sometimes zeros. Any of the three works provided the network is trained with the one it will be given.

9.3. What the weights show

Because the weights are a distribution over source positions and there is one distribution per output step, a trained model hands over a matrix: one row per word it produced, one column per word it read, every row summing to one. That matrix is readable, and reading it is the first thing attention gave the field that was not a performance number.

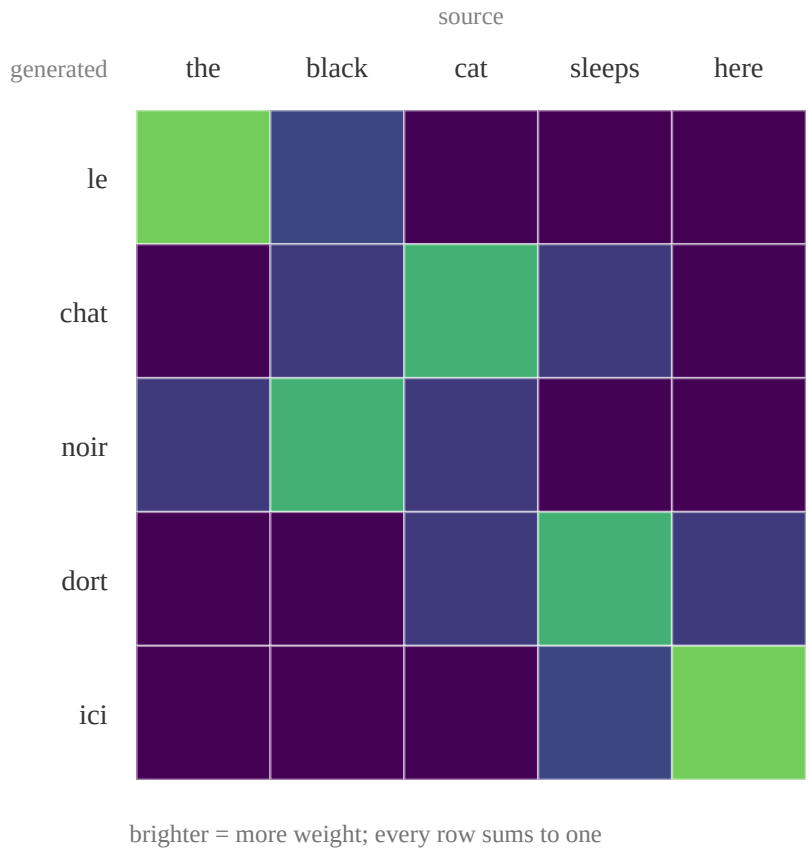


Figure 9.3.: The structure a learned alignment tends to have, drawn as a heat map with source words across the top and generated words down the side; brighter cells carry more weight. The mass sits near the diagonal because the two languages largely agree on order, and departs from it where they do not — the adjective and noun that swap places produce the crossing near the middle, and a phrase covered by a single word in the target draws weight from more than one column. This is a schematic of the structure, evaluated by the figure script; it is not the output of a trained model. *Drawn for these notes.*

What such matrices show, for a pair of languages that mostly agree on word order, is a bright band along the diagonal that leaves it exactly where the languages disagree — adjective and noun exchanging places in French, a compound in one language covered by two words in the other. The band is soft rather than sharp. A word rarely attends to precisely one source word and nothing else; it attends mostly to one and a little to its neighbours, which is the difference between an alignment learned as a continuous quantity and one imposed as a discrete matching.

The model was never told any of this. It was told to predict the next Italian word and it inferred

the correspondence as a by-product, which is worth pausing on: a mechanism introduced to widen a channel turned out to also expose what the network was using the channel for. That secondary property is why attention weights are now routinely plotted over images, over tokens and over patches whenever someone wants to ask a trained model what it was looking at — though the answer they give is where the weights went, which is not always the same as what the model relied on.

9.4. Attention as a layer in its own right

Nothing in Equation 9.2 through Equation 9.4 mentions translation, and on inspection very little of it mentions recurrence either. The mechanism takes one vector standing for a request, a set of vectors standing for available information, and returns a weighted blend of the second chosen by comparison with the first. The encoder states could be anything; the decoder state could be anything. What follows detaches the mechanism from the model it was invented inside, and the result is a layer that can be used anywhere.

Fix the vocabulary first, because the general version needs names that do not presume an encoder. The vector expressing what is wanted is the **query** q. The vectors that can be looked at are the **inputs** $x_1, \dots, x_N$. The comparison produces a similarity, the similarities are normalized, and the inputs are averaged. So far this is a renaming, and three generalizations turn it into something better.

The first replaces the learned comparison f_{att} with a plain dot product, $e_i = q \cdot x_i$. This is a reduction in expressive power and it buys parallelism: a dot product between a query and N inputs is a matrix–vector product, which hardware does at full speed, whereas f_{att} is a small neural network evaluated N times. The trade turned out to be overwhelmingly worth making, and it recurs in this lecture: a mechanism that is slightly weaker per unit but expressible as a matrix multiply beats a stronger one that is not.

The dot product needs one correction before it can be used, and the correction is not cosmetic. If the query and the inputs have dimension D and their entries are roughly independent with unit variance, the dot product is a sum of D such products and its typical magnitude grows like $\sqrt{D}$. Large-magnitude scores make the softmax of Equation 9.3 peaked: with a few hundred dimensions the largest score wins almost all of the mass, the distribution approaches a one-hot vector, and the softmax's gradient there is nearly zero. Training then stalls, not because the model is wrong but because the scores were too large. Dividing by $\sqrt{D}$ removes the dimension's effect on scale:

$$e_i = \frac{q \cdot x_i}{\sqrt{D}} \tag{9.5}$$

This is the *scaled* dot product, and the scaling is what keeps the gradient alive as models grow wide.

The second generalization lets several queries be answered at once. There is no reason to process one query at a time, and in every use below there are many: pack them as the rows of a matrix Q of shape $N_Q \times D$ and the inputs as the rows of X of shape $N_X \times D$. Then every score for every query is one matrix multiply,

$$E = \frac{QX^\top}{\sqrt{D}} \tag{9.6}$$

giving an $N_Q \times N_X$ matrix whose entry $E_{i,j}$ is query i against input j. Softmax is applied along the input axis, so that each query's row of weights sums to one, and the weighted sums of Equation 9.4 are a second matrix multiply,

$$A = \text{softmax}(E), \qquad Y = AX \tag{9.7}$$

with Y of shape $N_Q \times D$: one output vector per query. Two matrix multiplies and a softmax between them, and the layer is done.

> **Note**
>
> Which axis the softmax runs along is the detail most often got wrong when implementing this, and getting it wrong produces a network that trains to something mediocre rather than one that fails loudly. The weights for a given query must form a distribution over the *inputs*, so the normalization is along the axis indexing inputs. With E laid out as queries × inputs, that is the row.

9.5. Keys and values

The third generalization is the one that turns a retrieval rule into a learnable layer, and the argument for it is worth making slowly because the mechanism as stated has a conflict buried in it.

Each input vector x_i is doing two jobs. It is what the query is compared against, and it is what gets returned when the comparison succeeds. Those are different jobs and there is no reason the same vector should be good at both. The features that make a word findable — its position in the sentence, its part of speech, the fact that it is the subject — need not be the features another part of the network wants to receive about it. Forcing one vector to serve both roles means every improvement to one degrades the other.

So split them. Learn two linear transformations of the inputs:

$$K = XW_K, \qquad V = XW_V \tag{9.8}$$

The **keys** K are what queries are matched against; the **values** V are what gets returned. The queries themselves are usually a learned projection too, $Q = X_Q W_Q$, so that the vectors doing the asking can be shaped independently of the vectors doing the answering. The layer becomes

$$Y = \text{softmax}\left(\frac{QK^\top}{\sqrt{D_K}}\right) V \tag{9.9}$$

which is the equation the rest of this chapter is about. Its parameters are three matrices, W_Q, W_K and W_V, and nothing else; the attention operation itself has no weights at all.

The database analogy is close enough to be useful and worth stating precisely so it can then be qualified. A lookup table stores key–value pairs, receives a query, finds the key that matches, and returns the corresponding value. Equation 9.9 does the same thing with two changes: the match is a graded similarity rather than an equality test, and the return is a blend of every value weighted by how well its key matched rather than a single row. Soft retrieval, in other

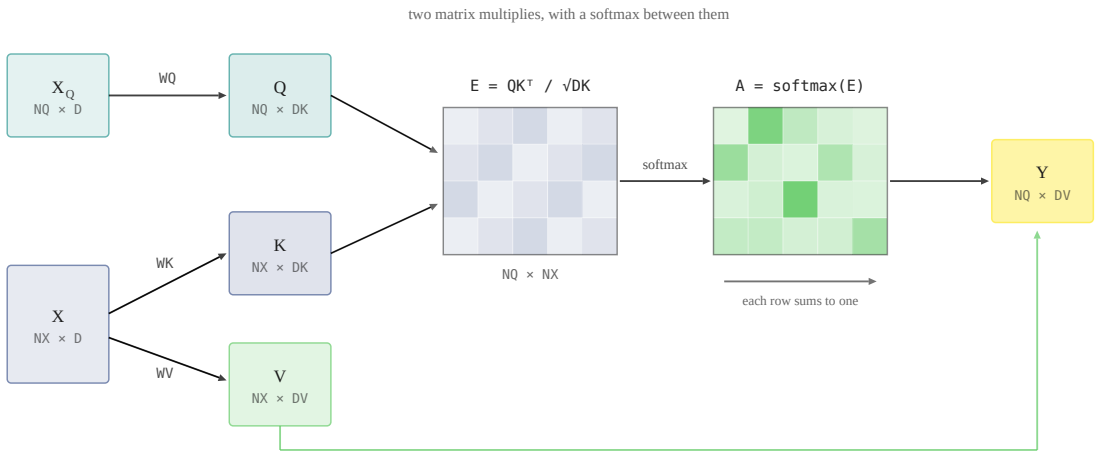


Figure 9.4.: The attention layer of Equation 9.9, with every shape marked. Queries and inputs enter on the left, the key and value projections produce K and V, the outer product $QK^\top$ gives an $N_Q \times N_X$ grid of scaled scores, the softmax normalizes each query's row into a distribution over inputs, and multiplying by V returns one output vector per query. *Drawn for these notes.*

words, and soft is what makes it differentiable — a hard lookup has no useful gradient with respect to the query, and this one does.

Notice what the layer does *not* assume. It takes two sets of vectors and returns one, and nothing in Equation 9.9 refers to order, adjacency, or time. It works on the words of a sentence, the patches of an image, the nodes of a graph, or an unordered pile of vectors with no structure at all. That generality is the reason the rest of the course keeps meeting this equation.

9.6. One set, attending to itself

Equation 9.9 has two inputs, X_Q and X, and translation supplies them naturally: the decoder asks, the encoder answers. Attention used this way, with queries from one collection and keys and values from another, is **cross-attention**, and it is the right tool whenever a problem genuinely has two kinds of thing in it — a source sentence and a translation, an image and the caption being written about it.

Many problems have only one. Classifying an image or modelling a sentence involves a single collection of vectors that needs to be processed, with nothing external to consult. For those, take the same layer and feed it the same set three times:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V \tag{9.10}$$

with all three projections applied to the same $N \times D$ matrix of inputs. This is **self-attention**. Every vector emits a query describing what it wants to know, a key advertising what it offers, and a value carrying what it will hand over; then every vector's query is compared against every vector's key and each output is the corresponding blend of values. Nothing else changes. Everything above the projections in Figure 9.4 — the scores, the softmax, the weighted sum — is unmodified, and the layer maps N input vectors to N output vectors, each of which has consulted all the others.

The three projections in Equation 9.10 are usually computed as one matrix multiply rather than three, by concatenating W_Q, W_K and W_V into a single $D \times 3D$ matrix. This changes nothing mathematically and matters in practice: hardware prefers a few large matrix multiplies to many small ones, and the fused projection is what implementations do.

9.6.1. The layer does not know what order the inputs came in

Shuffle the rows of X and follow the consequences through Equation 9.10. The queries, keys and values are computed row by row, so they come out identical but shuffled. The scores $E_{ij} = q_i \cdot k_j / \sqrt{D_K}$ compare the same pairs of vectors as before, so the score matrix has the same entries with its rows and columns permuted the same way. Softmax normalizes each row independently and does not care what order the rows arrive in. The weighted sums pair the same weights with the same values. So the outputs are the same vectors, permuted exactly as the inputs were:

$$F(\sigma(X)) = \sigma(F(X)) \tag{9.11}$$

for any permutation σ. Self-attention is **permutation equivariant**, which is the same relation Section 6.11 established for convolution and translation, with a different group acting.

The consequence is worth stating bluntly, because it is easy to slide past. Self-attention does not operate on sequences. It operates on *sets*. The vectors happen to be packed into a matrix and a matrix has rows in an order, but the layer's computation is blind to that order, and any apparent sensitivity to sequence in a model built from these layers has to come from somewhere else.

Sometimes that blindness is exactly right: a set of object proposals or a bag of image patches has no intrinsic order to respect. Usually it is not. "The dog bit the man" and "the man bit the dog" contain the same words, so a model whose only mixing operation is self-attention cannot distinguish them.

The fix is to put the order into the inputs, since it cannot live in the layer. Attach to each input vector a **positional encoding**, a vector that is a fixed function of the index, either added to the input or concatenated with it. The original transformer used sinusoids of geometrically spaced frequencies,

$$p_{t,2i} = \sin\left(\frac{t}{10000^{2i/D}}\right), \qquad p_{t,2i+1} = \cos\left(\frac{t}{10000^{2i/D}}\right) \tag{9.12}$$

where t is the position and i indexes the dimension pair, so that each position gets a distinct pattern and nearby positions get similar ones. A learned lookup table indexed by position works about as well and is common; more recent models mostly use rotary embeddings, which encode position by rotating queries and keys so that scores depend on relative rather than absolute position. All of them do the same job: they make two otherwise identical vectors distinguishable by where they sit.

9.6.2. Blocking some of the connections

Full self-attention lets every position see every other, and there are problems where that is wrong rather than merely wasteful. Language modelling is the clear case. If the model is trained to predict the next token at every position at once, then the output at position t must not depend on the tokens at positions after t — otherwise it can read the answer it is being asked to produce, and the loss goes to zero without anything being learned.

Preventing that requires no new machinery. Set the disallowed scores to $-\infty$ before the softmax:

$$E_{ij} \leftarrow -\infty \quad \text{for } j > i \tag{9.13}$$

Then $\exp(-\infty) = 0$, those weights come out exactly zero, and the output at position i is a blend of the values at positions 1 through i only. This is **masked self-attention**, and the pattern above is the causal mask; other patterns block other things — a local window, a block-diagonal restriction to within-segment attention — by the same route.

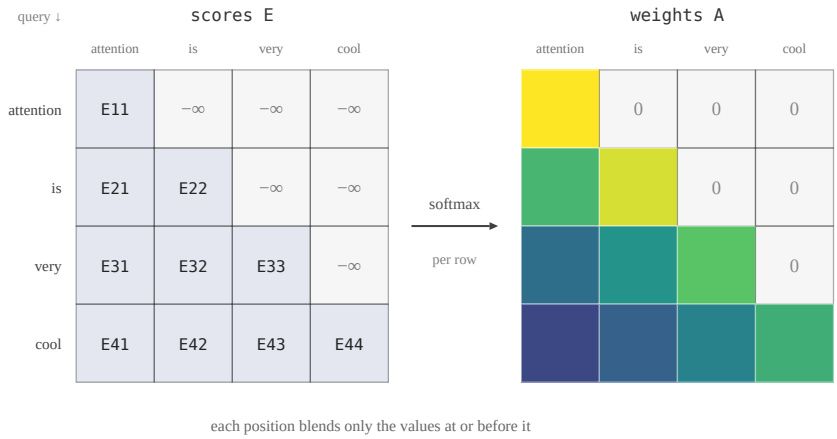


Figure 9.5.: Masked self-attention. Scores above the diagonal are overwritten with $-\infty$ before the softmax, so their weights come out zero and each position blends only the values at or before it. Trained this way on "attention is very cool", the position holding *attention* predicts *is* without ever having seen it. *Drawn for these notes.*

Masking is what lets self-attention take over the language-modelling job of Section 8.6 outright. The recurrent model enforced causality by construction — a state moving forward in time cannot consult the future — and paid for it with a sequential update. The masked attention layer enforces causality by arithmetic and computes every position at once.

9.7. Several heads

A single self-attention layer produces one set of weights per position, which means every position gets one distribution over what to look at. That is a real restriction. A word may need its syntactic governor, its coreferent, and the topic of the paragraph simultaneously, and one distribution cannot concentrate on three unrelated things without diluting all of them.

Run several attention layers side by side instead. Split the model dimension into H **heads**, give each its own W_Q, W_K and W_V projecting into a head dimension D_H, run Equation 9.9 independently in each, concatenate the H outputs back to width HD_H, and mix them with one more learned matrix W_O:

$$O = \text{concat}(Y_1, \dots, Y_H)\, W_O \tag{9.14}$$

Conventionally $D_H = D/H$, so the concatenation returns to the input width and the layer is a map from $N \times D$ to $N \times D$ regardless of how many heads it has. A model with $D = 512$ and $H = 8$ gives each head 64 dimensions to work in.

Note what that convention buys: at $D_H = D/H$ the parameter count and the arithmetic are essentially independent of H, since the heads split the width rather than duplicating it. Eight heads of width 64 cost what one head of width 512 costs. What changes is that eight separate distributions are computed instead of one, and the heads are identical in architecture and differ only in weights — they are initialized differently and specialize because the loss rewards their specializing, not because anyone assigned them roles.

The implementation runs all heads in one batched matrix multiply rather than in a loop, which is why the whole multi-head layer, for all its indices, reduces to four matrix multiplies: the fused QKV projection, the batched $QK^\top$, the batched weighting of V, and the output projection W_O. Multi-head is the form in which self-attention is essentially always used.

9.8. What it costs

Those four matrix multiplies are worth pricing, because two of them scale differently from the others and the difference decides what the layer can be used for.

The projections at either end — step one and step four — are applied to each vector independently, so they cost $O(ND^2)$ for N vectors of width D: linear in the number of vectors. The two in the middle are not. Computing $QK^\top$ compares every query against every key, which is N^2 scores per head, and weighting V by those scores costs the same again. Attention is $O(N^2D)$, quadratic in the number of inputs, and no rearrangement of the algebra changes that, because the layer's defining property is that every output consults every input.

The memory is the sharper constraint, because a naïve implementation *materializes* the score matrix. That is H matrices of $N \times N$ held at once, and the numbers get out of hand quickly: at $N = 100{,}000$ tokens with $H = 64$ heads, the attention weights alone are 6.4×10^{11} entries, which even at two bytes each is 1.28 terabytes. No accelerator has that. Long-context attention was for several years limited not by arithmetic but by the refusal of that matrix to fit.

FlashAttention removed the constraint by never forming the matrix. It fuses the two middle multiplies, walking over the inputs in tiles small enough to sit in on-chip memory, accumulating each output as it goes with an online softmax that never needs a whole row at once. The result is exact — this is not an approximation of attention — and the memory drops from $O(N^2)$ to $O(N)$ while the arithmetic stays quadratic. Nearly every long-context model in use depends on it.

The quadratic compute that remains is usually presented as attention's flaw, and the presentation deserves an objection. In ordinary algorithm design a quadratic term is a defect to be engineered away. Here the term buys something: more compute spent on a sequence is more computation performed on it, and a model that does more work per input has more opportunity to arrive at

a good answer. Attention's cost is not overhead attached to the useful work; within limits it *is* the useful work. Expensive and useless would be a defect. Expensive and proportionately more capable is a price.

9.9. Three ways to process a sequence

The course has now built three mechanisms that turn a collection of inputs into a collection of outputs, and they are worth putting side by side, because the transformer's design is a bet about which of the three properties matters.

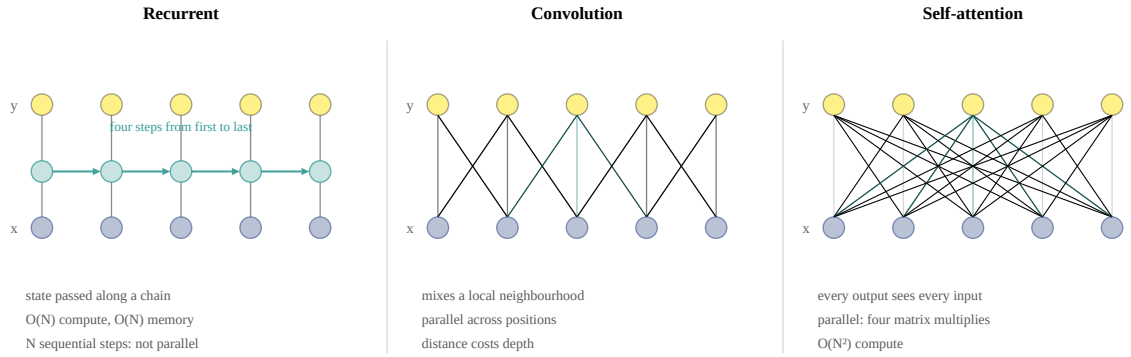


Figure 9.6.: The three primitives compared on how information travels between distant positions. The recurrent network passes it along a chain, so distance costs sequential steps. The convolution mixes locally, so distance costs depth. Self-attention connects every position to every other in one layer, so distance costs nothing — and every connection is drawn, which is also the picture of why it costs $O(N^2)$. *Drawn for these notes.*

A **recurrent network** works on one-dimensional ordered sequences and processes them in $O(N)$ compute and memory, which is the best of the three. Its defect is that the recurrence is inherently serial: h_t cannot be computed before h_{t-1}, so a sequence of length N takes N sequential steps no matter how much hardware is available.

A **convolution** works on grids of any dimension and is embarrassingly parallel — every output position is independent of every other. Its defect is reach. A single layer mixes information only within its kernel, so relating two distant positions requires either enormous kernels or a deep stack, and Section 6.8 showed exactly how slowly the receptive field grows.

Self-attention works on sets, with no assumption of order or grid. Every output depends directly on every input after one layer, so distance costs nothing, and the whole operation is four matrix multiplies, which parallelize as well as anything in computing. Its defect is the quadratic cost just described.

The reason the third property is decisive is a fact about hardware rather than about sequences. Single processors stopped getting appreciably faster some time ago; what has continued to grow, by orders of magnitude, is how many processors can be pointed at one problem. An algorithm that can only use one fast processor is stuck waiting on a trend that has ended, while an algorithm that can spread over a thousand of them rides the one that has not. Recurrence is the first kind and attention is the second, and that — more than any argument about which mechanism is a better model of language — is why the field went the way it did.

9.10. The transformer block

Vaswani et al. put self-attention at the centre of an architecture in 2017 and used the title to say how far they were prepared to take the idea. The block they defined is short enough to state completely and has barely changed since.

Take a set of vectors x. Pass them through multi-head self-attention, which is the only operation in the whole block where vectors see each other. Wrap it in a residual connection, for the reasons Section 7.8 gave: a path that skips the transformation keeps the gradient intact through a deep stack. Normalize, with layer normalization — the variant of Section 7.2 that standardizes each vector across its own features, using no statistics from other vectors and therefore none from other positions, which is what makes it safe here.

Then do something quite different. Self-attention mixes information between vectors and computes nothing much *within* one; the weighted sum of values is a linear operation, and stacking such layers would compose into another linear map on the set. So the block adds a second primitive: a small MLP, applied independently and identically to each vector, conventionally two layers widening from D to $4D$ and back with a nonlinearity between. It is also wrapped in a residual connection and followed by a second normalization.

$$\begin{aligned} z &= \text{LayerNorm}\left(x + \text{MultiHeadAttn}(x)\right) \\ y &= \text{LayerNorm}\left(z + \text{MLP}(z)\right) \end{aligned} \tag{9.15}$$

The division of labour is the design. Every position gets a chance to gather what it needs from everywhere else, then a chance to think about what it gathered, alternating for as many blocks as the model has. Because the block maps $N \times D$ to $N \times D$, blocks compose without any bookkeeping: a transformer is a stack of them, identical in shape, each with its own weights. Six matrix multiplies dominate the cost — four from attention, two from the MLP — and every one of them is large and batched, which is the property Section 9.9 argued was decisive.

What has changed since 2017 is scale, not structure. The original model in its larger configuration stacked twelve blocks at $D = 1024$ with sixteen heads, 213 million parameters in all. GPT-2 two years later ran 48 blocks at $D = 1600$ with 25 heads for 1.5 billion parameters, and GPT-3 the year after that ran 96 blocks at $D = 12288$ with 96 heads for 175 billion. Across three orders of magnitude in parameters, Equation 9.15 is what was being stacked.

9.11. Stacking blocks into a language model

The blocks handle vectors, and words are not vectors, so a language model needs a layer at each end.

At the input, a learned embedding matrix of shape $V \times D$ for a vocabulary of size V: one row per token, and turning a token into a vector is a lookup. At the output, a projection of shape $D \times V$ turning each block's output vector back into a score per vocabulary entry, which a softmax and a cross-entropy loss compare against the token that actually came next. In between, blocks whose attention is masked as in Equation 9.13, so position t sees only positions up to t.

That is the entire architecture, and the training objective is the one Section 8.6 already introduced for recurrent networks: predict the next token, with the text as its own supervision. What changes is that every position's prediction is computed in parallel in one pass, and that the path from any token to any earlier token is one attention step rather than a chain of state updates.

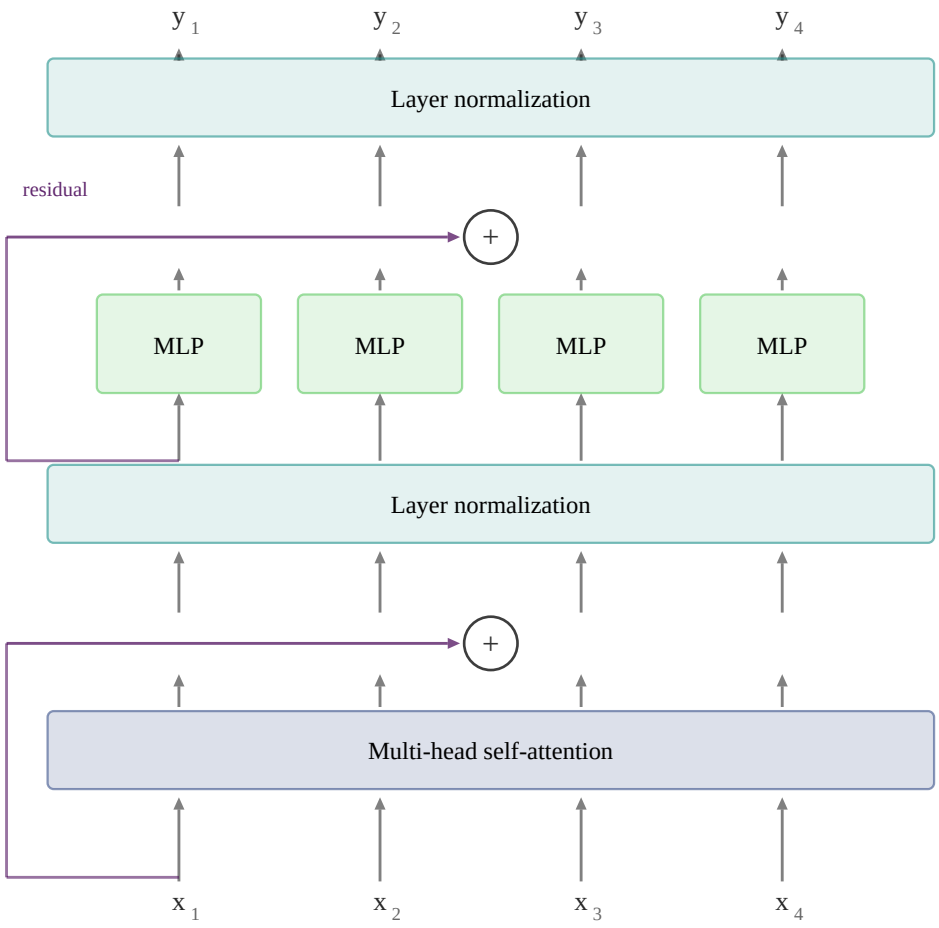


Figure 9.7.: The transformer block of Equation 9.15. Self-attention is the only operation in which vectors interact; layer normalization and the MLP act on each vector alone. Both sublayers sit inside residual connections. A transformer is this block repeated, with new weights each time and no change of shape. *Drawn for these notes.*

9.12. The same block, on images

Nothing in Equation 9.15 mentions language. It consumes a set of vectors, so applying it to images is a question of how to turn an image into a set of vectors — and Dosovitskiy et al. answered it about as directly as possible.

Cut the image into a grid of non-overlapping square patches, typically 16×16 pixels. A 224×224 image gives $14 \times 14 = 196$ of them. Flatten each patch into a vector of $16 \cdot 16 \cdot 3 = 768$ numbers and apply one learned linear map to width D. Add a positional encoding, since Equation 9.11 means the block would otherwise not know where any patch sat. Feed the 196 vectors to a stack of transformer blocks — unmasked, since there is nothing to predict in sequence and every patch may see every other. Average the outputs and apply a linear layer to get class scores.

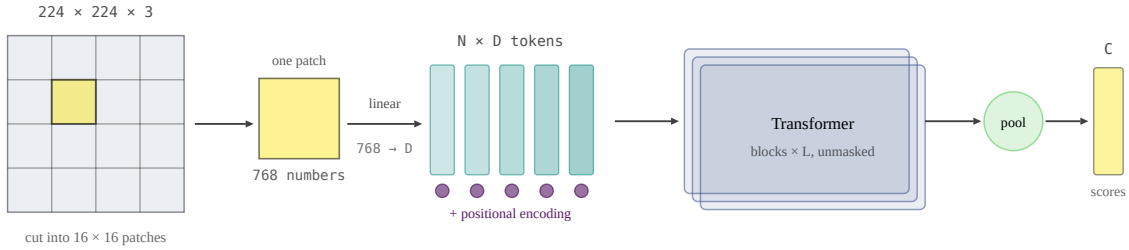


Figure 9.8.: The Vision Transformer. The image is cut into fixed patches, each flattened and linearly projected to width D; positional encodings supply the geometry the transformer cannot infer; the blocks are the same ones Figure 9.7 draws, unmasked; and the outputs are pooled and classified. The patch projection is exactly a 16×16 convolution with stride 16, so the only convolution in the model is the one that builds its input. *Drawn for these notes.*

The patch step deserves a second look, because it is a familiar operation in unfamiliar dress. Cutting an image into 16×16 tiles and applying the same linear map to each is a convolution with a 16×16 kernel, stride 16, three input channels and D output channels. The architecture that was presented as an alternative to convolutional networks begins with a convolution, and then contains no others — which locates the actual disagreement precisely. It is not about whether local linear filtering is useful at the very first step. It is about whether everything above that step should also be local.

9.13. What changed after 2017

Equation 9.15 is still recognisably what current models run, but four modifications have become close to standard, and a reader meeting a modern implementation will see all four. The lecture ran out of time before reaching them and left them as reading; they are short.

Pre-norm. In Equation 9.15 the normalization sits *outside* the residual addition, which has an awkward consequence: there is no setting of the weights for which the block computes the identity, since everything passing through is normalized on the way out. Moving each normalization to the front of its sublayer, inside the residual, restores it —

$$z = x + \text{MultiHeadAttn}\,(\,\text{Norm}(x)) \tag{9.16}$$

— and leaves a clean path from input to output. Deep stacks train more stably this way, without the learning-rate warmup the original arrangement needed, and it is now the default.

RMSNorm. Layer normalization subtracts each vector's mean and divides by its standard deviation. RMSNorm drops the centering and divides by the root mean square alone,

$$y_i = \frac{x_i}{\sqrt{\varepsilon + \frac{1}{D}\sum_j x_j^2}}\,\gamma_i \tag{9.17}$$

which is cheaper, loses nothing measurable, and is what most recent models use.

SwiGLU. The classic MLP computes $\sigma(XW_1)W_2$ with a widening of $4D$. The gated variant computes two projections and lets one gate the other elementwise,

$$Y = (\sigma(XW_1) \odot XW_2)W_3 \tag{9.18}$$

with the hidden width set to $\frac{8}{3}D$ so that three matrices cost what two did. It works better, and the paper that introduced it declined to explain why, attributing the result "as all else, to divine benevolence."

Mixture of experts. Learn E separate MLPs per block instead of one, and route each token to $A < E$ of them, chosen by a small learned gate. Parameters scale with E while the compute per token scales with A, which decouples how much a model knows from how much it spends per token — the reason the idea is now behind essentially every frontier model, though those models publish few enough details that this is inference rather than fact.

9.14. What this leaves

The chapter started with a single vector too narrow to carry a sentence, and every step after that was a removal. Removing the bottleneck gave a decoder that rebuilt its context at each step. Removing the decoder gave a layer taking queries against inputs. Removing the learned comparison gave a scaled dot product and with it the ability to run everything as a matrix multiply. Removing the distinction between the two input sets gave self-attention, which needed positional encodings put back precisely because the removals had gone as far as discarding order itself.

What is left is an operation on sets of vectors, parameterized by three projections, in which every element consults every other in one step. It carries no assumption about grids, adjacency, or time, which is why the same layer serves words, patches, and — in later chapters — point clouds, video frames, and pairs of modalities that have nothing structural in common. The transformer is the small amount of scaffolding needed to use that operation repeatedly: mix across positions, compute within each, wrap both in residual connections, stack.

Two things are worth carrying forward past the architecture. The first is the trade that keeps recurring: a mechanism that is weaker per unit of computation but expressible as a large matrix multiply beats a stronger one that is not, because hardware scales in width and not in speed. The second is that attention's cost is quadratic and that this has been survivable — better arithmetic on the same operation, as in FlashAttention, has bought more than any of the many attempts to replace it with something cheaper.

The next chapter turns from architectures to tasks: detection, segmentation, and the question of what a trained network has actually learned to look at.

References

- J. L. Ba, J. R. Kiros and G. E. Hinton, "Layer Normalization," 2016. arXiv:1607.06450
- A. Baevski and M. Auli, "Adaptive Input Representations for Neural Language Modeling," *ICLR*, 2019. arXiv:1809.10853
- D. Bahdanau, K. Cho and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *ICLR*, 2015. arXiv:1409.0473
- T. Brown et al., "Language Models Are Few-Shot Learners," *NeurIPS*, 2020. arXiv:2005.14165
- T. Dao, D. Y. Fu, S. Ermon, A. Rudra and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," *NeurIPS*, 2022. arXiv:2205.14135
- A. Dosovitskiy et al., "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ICLR*, 2021. arXiv:2010.11929
- M.-T. Luong, H. Pham and C. D. Manning, "Effective Approaches to Attention-Based Neural Machine Translation," *EMNLP*, 2015. arXiv:1508.04025
- N. Shazeer, "GLU Variants Improve Transformer," 2020. arXiv:2002.05202
- N. Shazeer et al., "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," *ICLR*, 2017. arXiv:1701.06538
- J. Su et al., "RoFormer: Enhanced Transformer with Rotary Position Embedding," 2021. arXiv:2104.09864
- I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *NeurIPS*, 2014. arXiv:1409.3215
- A. Vaswani et al., "Attention Is All You Need," *NeurIPS*, 2017. arXiv:1706.03762
- B. Zhang and R. Sennrich, "Root Mean Square Layer Normalization," *NeurIPS*, 2019. arXiv:1910.07467

Part II.

Vision Tasks

10. Object Detection and Segmentation

Lecture 9

Based on Lecture 9 of CS231n, Stanford University, Spring 2025.

10.1. Past a single label

Every architecture so far has ended the same way. Whatever the network did in the middle — convolve, recur, attend — the last layer produced C numbers, one score per class, and the image was reduced to the index of the largest. That output shape is a strong assumption, and it is wrong for most of what a vision system is asked to do. A photograph of a street does not have *a* label. It has a road, two pedestrians, a parked car, and a sky, each occupying a particular region, some of them appearing more than once.

Figure 2.9 laid out the four tasks that fall out of relaxing that assumption in different directions, and it is worth restating them in terms of what leaves the network. Classification emits one label for the whole image and keeps no spatial information at all. Semantic segmentation emits a label for every pixel and so retains full spatial detail, but has no notion of objects: two overlapping cats produce one connected region marked *cat*, and nothing in the output says there were two. Object detection emits a variable-length list of boxes, each with a class, which recovers the count and the rough position but not the shape. Instance segmentation emits a mask per object, which is the union of the last two: it knows there were two cats and it knows which pixels belong to each.

What differs across those four is the shape of the output and the loss that supervises it. What does not differ much is the middle. All four are built on a backbone that turns an image into a stack of feature maps, and by this point in the course that backbone is a settled question — a ResNet, or a Vision Transformer, or whatever the current best classifier happens to be, usually pretrained on a classification task and then adapted. The interesting design work has moved to the ends: how to shape an output that does not have a fixed length, and how to define a loss over it.

That framing also explains why the chapter closes on visualization rather than on another architecture. Once a network is producing spatially structured answers, the question of *why* it produced them stops being academic. In a medical setting, a model that reports a tumour without indicating where it found the evidence is close to useless, because the radiologist's job is not to receive the verdict but to check it.

10.2. Labelling every pixel

Take semantic segmentation first, because it is the task whose output shape is least like a classification vector and most like the input. Training data is an image paired with a second image of the same size in which each pixel carries a category index. At test time the network sees a photograph and must produce that second image.

The most direct approach reuses the classifier unchanged. To label a pixel, extract a small patch centred on it, run the patch through a classification network, and take the answer as the label of the centre pixel. The patch is necessary rather than incidental: a single pixel is a colour, and no classifier can say whether a particular brown pixel belongs to a cow or to a fencepost. Only the surrounding region carries the information, so the patch size is a choice about how much context the decision is allowed to consult.

This works and nobody does it, because the cost is prohibitive. One forward pass per pixel means around fifty thousand passes for a modest 224×224 image, and almost all of that work is redundant — adjacent patches overlap in all but a strip of pixels, so the network recomputes nearly identical features tens of thousands of times.

The redundancy points to its own fix, and the fix is the one convolution was designed for. A convolutional layer already applies the same filter at every spatial position, sharing computation between overlapping receptive fields by construction. If the network contains only convolutional layers — no flattening, no fully connected layers, nothing that destroys the spatial axes — then a $3 \times H \times W$ image passes through to give a $C \times H \times W$ block of scores, one length-C vector per pixel, in a single pass. Taking the argmax over the channel axis at every position yields the $H \times W$ map of predictions.

Training needs no new machinery either. Every pixel is performing a classification, so the loss is the softmax cross-entropy of Section 3.6 evaluated at every position and averaged,

$$L = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} -\log \frac{\exp(s_{ij,\,y_{ij}})}{\sum_{c=1}^{C} \exp(s_{ij,\,c})} \tag{10.1}$$

where $s_{ij,c}$ is the score the network assigns class c at position (i, j) and y_{ij} is that pixel's ground-truth category. Backpropagation proceeds unchanged. The cost of this arrangement is not in the loss but in the supervision it assumes: Equation 10.1 needs a category for every pixel of every training image, which for years meant somebody tracing object boundaries by hand.

This is the fully convolutional network, and stated that way it looks like the end of the story. It is not, for a reason that is purely about arithmetic. Every layer now operates at the full resolution of the input. A classification network can afford wide layers precisely because it has already thrown away spatial extent by the time the channel count gets large; a 512-channel feature map at 7×7 is 25,000 values, while the same 512 channels at 224×224 is 25 million. Holding full resolution through a deep stack multiplies the cost of every layer by a factor of a thousand or more, and the receptive field grows only as fast as the filters stack, so a network deep enough to see context is also one that cannot be run.

The resolution is the same one the classification architectures reached: downsample. Let the network reduce spatial size as it goes, through pooling or strided convolution, so that the deep layers are cheap and their receptive fields cover most of the image. The difference is that the output must come back out at full resolution, so having gone down, the network has to go back up. That gives the shape almost every dense-prediction model has used since — an encoder that trades resolution for channels and semantic depth, then a decoder that trades them back.

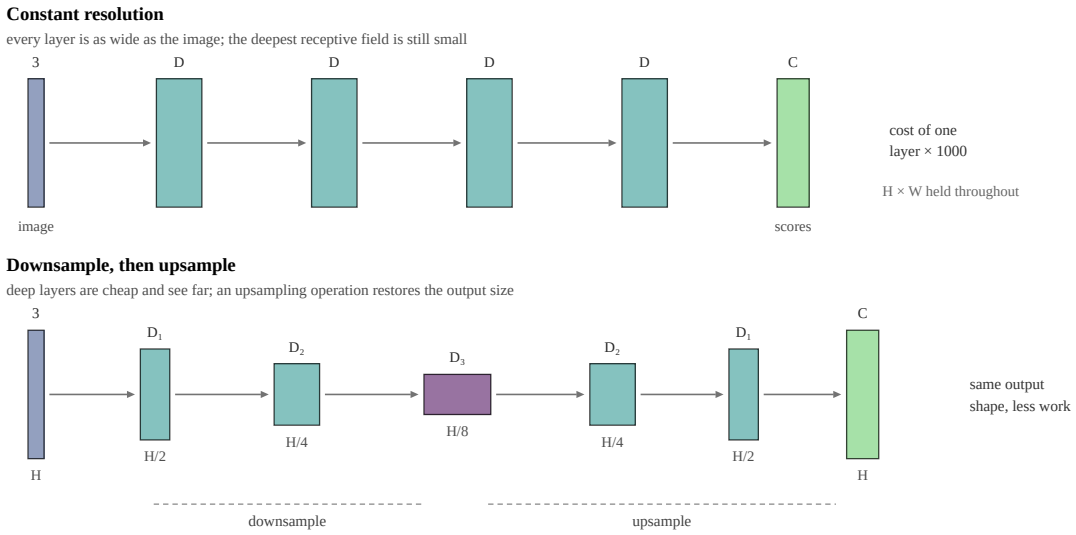


Figure 10.1.: The two ways to build a fully convolutional network. Holding the input resolution throughout (top) keeps every prediction aligned with its pixel but makes every layer cost as much as the first. Downsampling and then upsampling (bottom) makes the deep layers cheap and their receptive fields large, at the price of needing an operation that increases spatial size. *Drawn for these notes.*

Half of that shape is already familiar. Downsampling is pooling and strided convolution, both covered in Section 6.10. The other half has no counterpart yet: nothing so far in the course takes a small feature map and produces a larger one.

10.3. Putting the resolution back

The simplest upsampling operations have no parameters at all and are worth stating first, because the learnable one is best understood as their generalisation. To double a 2×2 map to 4×4, *nearest-neighbour unpooling* copies each input value into all four output cells that it covers. *Bed of nails* instead writes each value into one designated cell of its block and leaves the other three at zero, on the argument that subsequent convolutions will spread the value out anyway and the network should choose how rather than have averaging imposed on it.

Both discard something the encoder knew. Max pooling selected a specific position in each window — the location of the largest activation — and then threw that location away. *Max unpooling* keeps it: each pooling layer records which of its inputs was the maximum, and the corresponding unpooling layer writes its input back to exactly that position, zeroing the rest. Because the indices come from a particular pooling layer, this pairs the encoder and decoder position by position, and the resulting decoder is a mirror image of the encoder rather than an independent stack. What it buys is spatial alignment: an edge that was detected at a specific pixel is restored to that pixel rather than to the corner of a block.

None of the three has a parameter. The learnable version comes from reading strided convolution backwards. In a stride-2 convolution the filter advances two positions in the input for each one position in the output, which is what makes the output half the size; the stride is a ratio between movement in the input and movement in the output. Swap the two. Let the filter advance two positions in the *output* for each one in the *input*, and the output comes out twice as large.

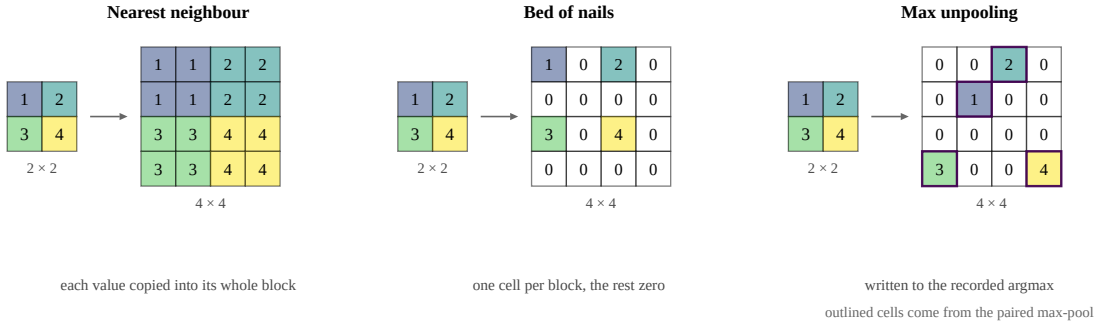


Figure 10.2.: Three parameter-free ways to double the spatial size of a feature map. Nearest-neighbour replicates, bed of nails writes one cell per block and zeroes the rest, and max unpooling writes to the position that the paired pooling layer recorded as its maximum — the only one of the three whose output stays aligned with the features that produced it. *Drawn for these notes.*

The operation that ratio describes is a *transposed convolution*, and its arithmetic is the reverse of the usual one in a specific sense. An ordinary convolution takes a dot product between the filter and a patch of the input, producing one output number from many input numbers. A transposed convolution takes one input number, multiplies the whole filter by it, and adds the result into a patch of the output — one input number scattering into many. Where the scattered patches overlap, as they must when the filter is wider than the stride, the contributions are summed.

A one-dimensional example makes the whole operation visible. Take an input of two values a and b, a filter (x, y, z), and stride 2. The first input contributes $a \cdot (x, y, z)$ starting at output position 0; the second contributes $b \cdot (x, y, z)$ starting at position 2. The output is

$$(ax,\ ay,\ az + bx,\ by,\ bz) \tag{10.2}$$

with the sum appearing at the one position both filters reached. The filter weights are learned exactly as any convolution's are.

The name is the last piece, and it comes from writing convolution as the linear map it always was. A convolution with kernel (x, y, z), stride 1 and one cell of zero padding, applied to a length-4 input, is a matrix product:

$$X\mathbf{a} = \begin{bmatrix} y & z & 0 & 0 \\ x & y & z & 0 \\ 0 & x & y & z \\ 0 & 0 & x & y \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.3}$$

Each row of X is the filter shifted to one output position, which is what makes the matrix banded and its entries tied together. Transposed convolution multiplies by X^{T} instead of X — hence the name, and hence also the older and worse name *deconvolution*, which the operation does not deserve since it does not invert the convolution, only its shape. The transpose is exactly the matrix that backpropagation through a convolution already uses to send gradients from output to input, so the forward pass of a transposed convolution is the backward pass of an ordinary one.

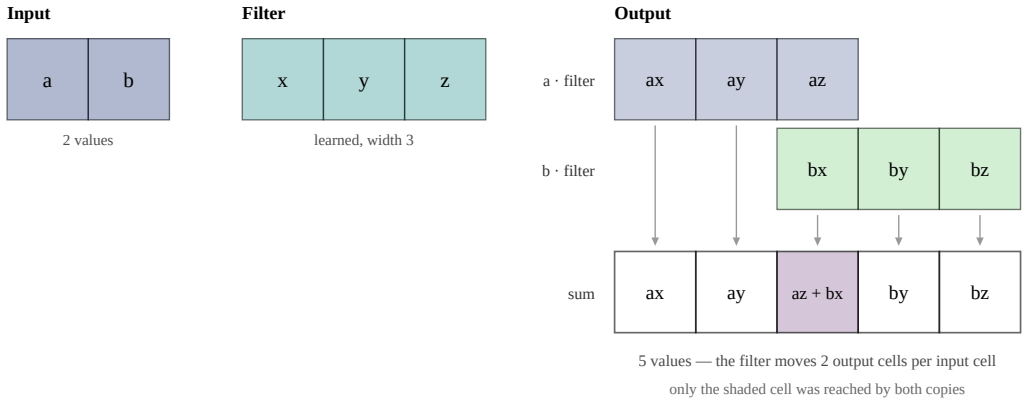


Figure 10.3.: A stride-2 transposed convolution in one dimension. Each input value scales a copy of the filter, the copies are laid down two output positions apart, and the single position covered by both copies receives their sum, giving Equation 10.2. *Drawn for these notes.*

One consequence of Equation 10.2 is worth knowing before using it. When the filter width is not a multiple of the stride, the output positions receive unequal numbers of contributions — in the example above, position 2 gets two terms and its neighbours one — and a network trained this way tends to produce a regular grid of brighter and darker cells. These checkerboard artifacts are common enough that many current models avoid transposed convolution altogether, upsampling by interpolation and following it with an ordinary stride-1 convolution to do the learning.

10.4. The shape that keeps coming back

Downsampling and upsampling in one network has an unwanted property: the information the decoder needs most is precisely what the encoder was built to discard. The deep layers know *what* is in the image, because their receptive fields are large and their features abstract. They do not know *where* very precisely, because eight-fold downsampling has left one feature vector to speak for a 8×8 block of pixels. Segmentation needs both, and the boundary between a cat and the grass is exactly the place where the coarse answer is worst.

U-Net fixes this by not routing all the information through the bottleneck. At each resolution the encoder's feature map is copied across and concatenated onto the decoder's feature map at the same resolution, so the layer that produces the final full-resolution output sees both the upsampled deep features and the original shallow ones, which never lost their spatial precision. The deep path supplies the category and the skip connection supplies the boundary.

The skip connections are the same device Section 7.8 used, applied for a different reason. In a residual block the shortcut exists so that gradients reach early layers and the identity is easy to represent; here it exists so that high-resolution detail reaches the output without having to survive a round trip through the bottleneck. In both cases the lesson is that a strictly sequential stack forces information through a channel that need not be the only one.

U-Net was published for biomedical segmentation and remains the strongest thing to reach for on a small medical dataset without a foundation model. Its shape also outlived its original purpose entirely: the denoising network at the centre of a diffusion model is a U-Net, for the same

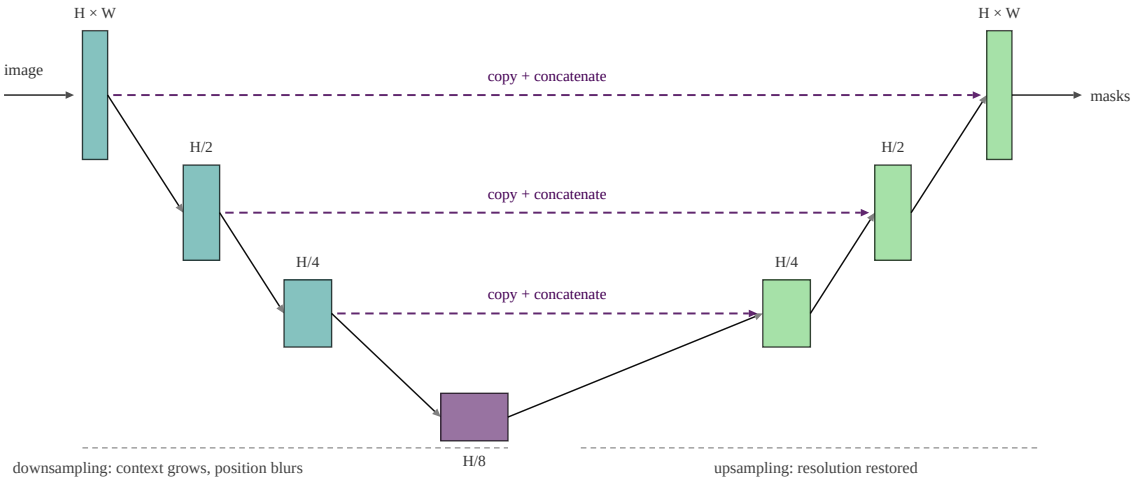


Figure 10.4.: U-Net. The left branch downsamples and widens, the right branch upsamples and narrows, and at each level the encoder's feature map is concatenated onto the decoder's before the next pair of convolutions. Depth carries semantics down the left side; the horizontal copies carry localisation across. *Drawn for these notes.*

structural reason, since predicting noise at every pixel is a dense-prediction problem needing both global context and pixel-accurate output.

What semantic segmentation cannot do, at any level of accuracy, is count. Equation 10.1 labels pixels, and two cats sitting together produce one region marked *cat* with nothing to indicate that two animals are present. Recovering objects, as opposed to categories, requires an output that is not a map.

10.5. Detection is not a regression problem

Start with the easy case, because it is genuinely easy and it fails in an instructive way. Suppose every image contains exactly one object. Then the output is a class and a box, the box being four numbers (x, y, w, h), and the network needs two heads on a shared backbone: a classification head producing C scores, and a regression head producing four coordinates.

Each head brings its own loss. The class scores take softmax cross-entropy; the coordinates take a regression loss, the squared distance from the ground-truth box. Both are differentiable, so both can be trained at once by adding them,

$$L = L_{\text{cls}}(s,\, y) + \lambda \left\| \mathbf{b} - \mathbf{b}^{*} \right\|^{2} \tag{10.4}$$

with s the predicted scores, y the true class, $\mathbf{b}$ the predicted box, $\mathbf{b}^{*}$ the true one, and λ a hyperparameter setting the exchange rate between the two. This is a *multi-task loss*, and the pattern — one backbone, several heads, one weighted sum — is how essentially every model in this chapter is trained. The awkward part is λ: the two terms are measured in different units, so it cannot be reasoned about from first principles and has to be tuned against the metric that is actually cared about.

Now allow more than one object, and the architecture breaks at a level below the loss. An image with one cat needs four numbers plus a class; an image with three needs twelve; a crowd scene needs hundreds. A fully connected output layer has a width fixed when the network is built.

There is no value of that width which is right, because the required width is a property of the image, and the network does not know the image when it is compiled.

The obvious workaround is to change what a single prediction is about. Rather than asking the network to describe the image, ask it about a *region*: given a crop, is this a cat, a dog, or background? That question has a fixed-size answer regardless of how many objects the image contains, and running it over many crops recovers a variable-length result — one detection per crop that came back as a non-background class.

The difficulty moves to which crops. Objects appear at any position, any scale, and any aspect ratio, so an exhaustive sweep is a four-dimensional search whose size grows as roughly the square of the number of pixels. On a 640×480 image, running a full network on every plausible box is not slow but infeasible, and the sliding-window trick that rescued segmentation does not apply, because the crops here differ in size and must each be resized to the classifier's fixed input.

10.6. Proposing regions

The way out, and the one the field took for most of a decade, is to stop searching exhaustively and get the candidates from somewhere cheaper. Before deep learning was involved at all, there were algorithms for finding image regions that look like they contain *some* object without knowing which — grouping pixels into blobs by colour and texture, merging them hierarchically, and returning the merged regions as candidates. Selective search is the best known, and it produces roughly two thousand proposals per image in a few seconds on a CPU. That is a reduction of several orders of magnitude from the exhaustive sweep, and it has high recall: the true objects are almost always among the two thousand, buried in a great many false ones the classifier can reject.

R-CNN is the direct application of that idea. Run selective search, warp each proposed region to the fixed 224×224 input, push it through an ImageNet-pretrained convolutional network, classify the resulting features, and — since the proposal's box was produced by a colour-and-texture heuristic and will not be tight — also regress four corrections $(\mathrm{d}x, \mathrm{d}y, \mathrm{d}w, \mathrm{d}h)$ that nudge the box onto the object. Using a pretrained classifier as a feature extractor for a task it was not trained for is the transfer-learning argument of Section 7.12, and R-CNN is where it first paid off spectacularly on detection.

It is also unusably slow, for a reason that should look familiar. Two thousand proposals means two thousand independent forward passes per image, and those proposals overlap heavily, so the network recomputes nearly the same features many times over — the same redundancy that made per-pixel classification hopeless in Section 10.2, in a new setting.

And the fix is the same fix. Convolution preserves spatial correspondence: a position in a conv5 feature map corresponds to a known region of the input image. So there is no need to crop the *image* and then run the network. Run the network once on the whole image, then crop the *features*. Fast R-CNN does exactly that — one backbone pass per image, a projection of each proposal onto the feature map, a crop, and a small per-region head producing the class and the box offsets. The backbone's cost is paid once and shared across all two thousand regions; only the small head runs per region. The paper reports training VGG16 nine times faster than R-CNN and testing 213 times faster, at higher accuracy.

Cropping features is less trivial than cropping pixels, because the proposal's box is in image coordinates and the feature map is on a coarser grid — a stride-32 backbone means one feature cell per 32×32 pixel block. *RoI pooling* handles this by snapping the projected box outward to whole feature cells, dividing the snapped region into a fixed 7×7 grid of roughly equal

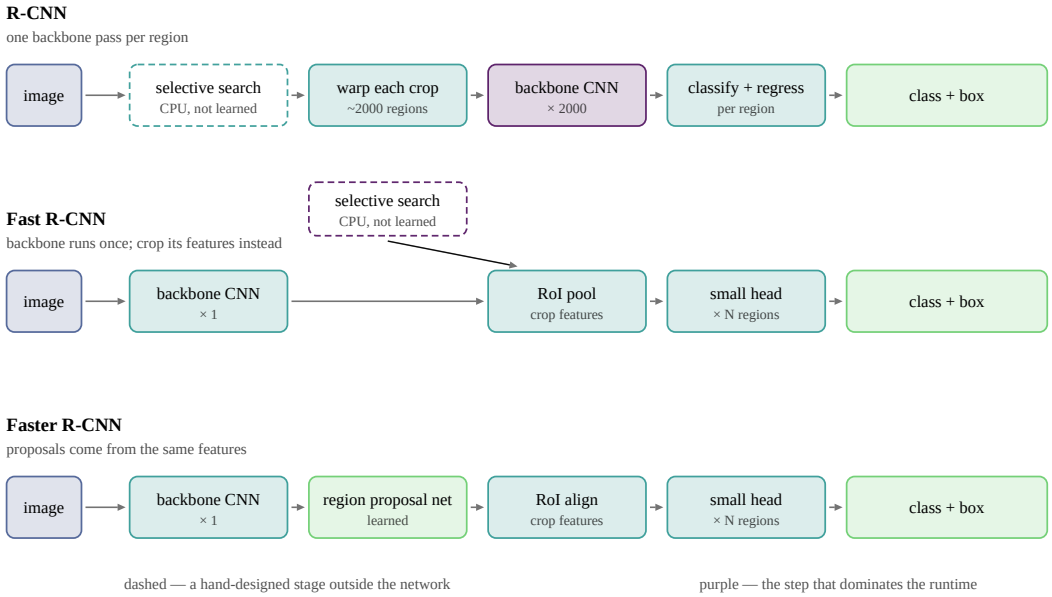


Figure 10.5.: Three generations of proposal-based detection. R-CNN crops the image and runs a full backbone per region. Fast R-CNN runs the backbone once and crops its features, leaving only a small head per region. Faster R-CNN replaces the external proposal algorithm with a network reading the same features, so nothing outside the model remains. *Drawn for these notes.*

subregions, and max-pooling within each. The output is $C \times 7 \times 7$ regardless of the proposal's size, which is what lets a fixed-width head consume regions of any shape.

The snapping is a real defect, not a rounding detail. A box that should start at feature coordinate 3.4 is treated as starting at 3, displacing the cropped features by up to half a cell — several pixels in the original image — and the subregion boundaries inherit the same error. For a class label this hardly matters. For anything that has to be pixel-accurate, it matters a great deal, which is why it was fixed only when masks arrived. *RoI align* does not snap. It keeps the box's real-valued coordinates, samples at regularly spaced points inside each subregion, and computes each sample by bilinear interpolation of the four surrounding feature cells,

$$f_{xy} = \sum_{i,j \in \{1,2\}} f_{ij}\,(1 - |x - x_i|)(1 - |y - y_j|) \tag{10.5}$$

where f_{ij} are the features at the four neighbouring grid positions and (x_i, y_j) their coordinates. Every term is differentiable in x and y, so gradients flow back through the sampling positions as well as the features.

10.7. Learning the proposals

Fast R-CNN made the network cheap enough that the proposal step became the bottleneck. Selective search runs on the CPU and takes about two seconds per image, against a fraction of a second for everything else; the model was now waiting on a hand-designed algorithm from

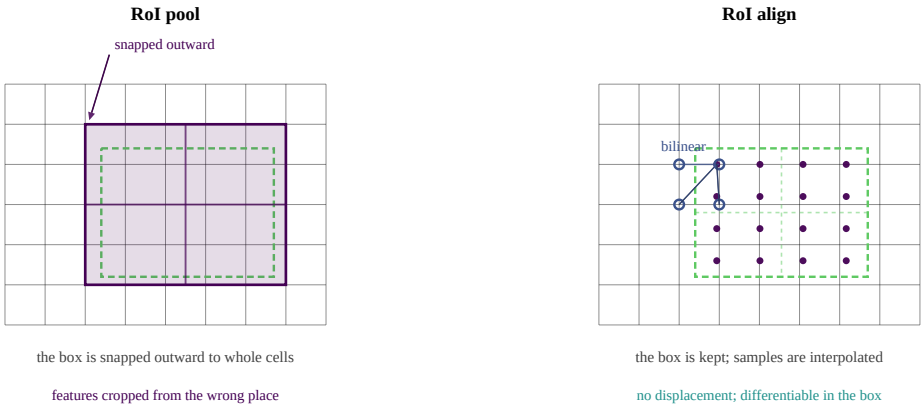


Figure 10.6.: RoI pool against RoI align. Pooling snaps the projected box to feature-cell boundaries, so the cropped region is displaced from the object by up to half a cell; align keeps the real-valued box and reads each sample point by bilinear interpolation from its four neighbours, as in Equation 10.5. *Drawn for these notes.*

before the deep-learning era, one that cannot be trained and does not improve when the rest of the system does.

Faster R-CNN removes it by predicting proposals from the features the backbone has already computed. The *region proposal network* is a small convolutional head over the feature map. At each of its spatial positions it considers K *anchor boxes* — reference rectangles of fixed sizes and aspect ratios, centred on that position — and predicts two things per anchor: an objectness score saying whether the anchor contains an object of any class, and four corrections transforming the anchor towards the true box if it does. For a $512 \times 20 \times 15$ feature map this is a $K \times 20 \times 15$ tensor of scores and a $4K \times 20 \times 15$ tensor of offsets, both produced by ordinary convolutions in one pass on the GPU. Sorting the resulting boxes by objectness and taking the top few hundred gives the proposals, and the paper needs only 300 of them where selective search supplied two thousand.

Anchors deserve a moment, because the device recurs throughout detection. The network is not asked to produce boxes from nothing, which is a hard unconstrained regression; it is asked to classify a fixed set of guesses and to correct the good ones slightly. Regressing a small offset from a nearby reference is a far better-conditioned problem than regressing absolute coordinates, and using K anchors of different shapes at each position means at least one reference is usually close to any given object.

With the RPN in place, Faster R-CNN trains end to end against four losses at once: objectness and box regression in the proposal network, class scores and box regression in the per-region head. It runs at about five frames per second including every step, and the structure has a name — a *two-stage detector*, in which the first stage runs once per image and the second once per surviving region.

One piece of the pipeline is deliberately not learned. Many anchors near the same object survive, so the raw output contains clusters of near-duplicate boxes for a single thing. *Non-maximum suppression* removes them: sort detections by score, keep the highest, discard any remaining box overlapping it by more than a threshold in intersection-over-union, repeat. It works, it is universally used, and it is a hand-written procedure with a hyperparameter sitting outside the network — which is exactly the sort of component the rest of this chapter is about absorbing into learning.

10.8. One pass

Two stages means two passes, and the second one runs per region. If the goal is a detector that keeps up with a camera, the natural question is whether the second stage is needed at all — whether the classification could not be done in the same pass that produced the boxes.

YOLO answers that it can, by fixing the set of candidate boxes in advance and predicting everything about them at once. Divide the image into an $S \times S$ grid, 7×7 in the original. Attach B base boxes to every cell. Then a single fully convolutional pass produces, for each cell, five numbers per base box — four coordinate corrections and a confidence that this box contains an object — and one distribution over the C classes. The output of the whole network is one tensor,

$$S \times S \times (5B + C) \tag{10.6}$$

and detection is done. There is no proposal stage, no per-region head, and no variable-length intermediate: the network's output shape is fixed at compile time, and the variable-length answer is recovered afterwards by thresholding on confidence and running non-maximum suppression over what survives.

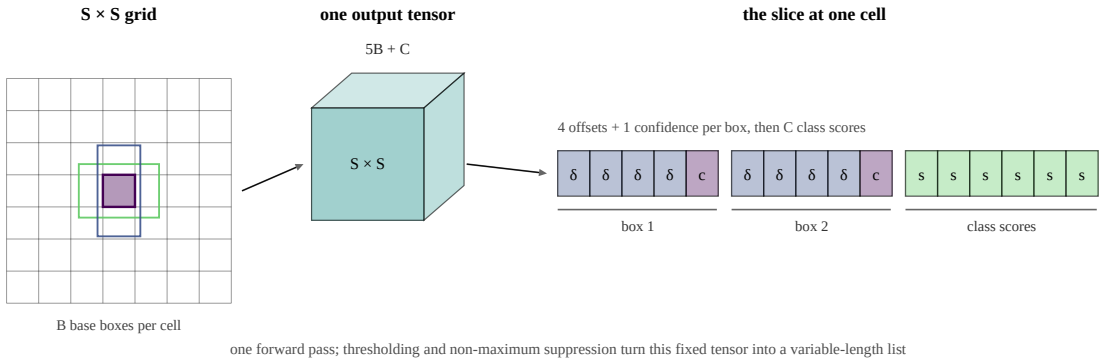


Figure 10.7.: The YOLO output tensor of Equation 10.6. Every grid cell predicts B boxes as offsets from its base boxes, each with a confidence, plus one distribution over classes for the cell. The whole prediction is one forward pass; thresholding and non-maximum suppression turn the fixed-size tensor into a variable-length list of detections. *Drawn for these notes.*

Structurally this is a region proposal network that also predicts categories, with the anchors laid out on a grid instead of being scored and re-cropped. What it buys is speed: the original model ran at 45 frames per second and a reduced version at 155, against Faster R-CNN's five. What it costs, in that first version, is localisation accuracy — the class distribution is per cell rather than per box, so two objects of different classes in one cell cannot both be reported correctly, and the coarse grid limits how precisely small objects can be placed. Later single-stage detectors (SSD, RetinaNet) closed most of that gap by predicting classes per box and by fixing the class imbalance that dense prediction creates, and the two-stage advantage in accuracy has largely evaporated.

The trade this sets up outlived all the specific architectures. Two-stage detectors spend computation refining a small set of promising regions and are more accurate per unit of accuracy-critical work; single-stage detectors spend a fixed budget everywhere and are faster. YOLO is now many

versions past the one described here and remains what most industrial deployments reach for, because in practice latency is a hard constraint and the last point of average precision is not.

10.9. Detection as set prediction

Every detector so far shares two hand-designed components. Anchors encode a prior about what shapes objects take; non-maximum suppression encodes a rule for what counts as a duplicate. Both work, both have hyperparameters, and neither is learned. DETR removes both by changing what the network is asked to produce: not a dense field of scored boxes to be filtered, but a *set* of exactly N predictions, once, with the network itself responsible for not repeating.

The architecture is the transformer of Section 9.10 with almost nothing added. A convolutional backbone reduces the image to a feature map; the map's cells become tokens; positional encodings supply the geometry; a transformer encoder mixes them. The decoder is where the new idea sits. Its inputs are N *object queries* — learned vectors, one per output slot, that are parameters of the model rather than functions of the image. Each query passes through self-attention against the other queries and cross-attention against the encoder output, and the resulting vector goes through a small feed-forward network producing a class and a box. N is fixed and chosen larger than the number of objects any image is expected to contain; the original model used $N = 100$.

Nothing in that description tells a query *where* to look. The queries begin as arbitrary learned vectors, and what stops all hundred from converging on the same large object in the middle of the image is the self-attention among them: each query sees what the others are proposing and can differentiate itself. Specialisation is a consequence of training rather than a constraint imposed by construction, and after training the individual queries do settle into rough preferences for regions and scales.

The loss is the part that makes this work, and it has to solve a problem the previous detectors never faced. The network emits N predictions in some order; the ground truth is a set of objects in no particular order. There is no natural correspondence between slot 7 and any particular cat, so the loss must first decide which prediction is responsible for which object. DETR does this by finding the cheapest one-to-one assignment — a *bipartite matching* between the N predictions and the ground-truth objects, padded with a special $\emptyset$ class meaning *no object* so that both sides have N elements:

$$\hat{\sigma} = \arg\min_{\sigma \in \mathfrak{S}_N} \sum_{i=1}^{N} \mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}) \tag{10.7}$$

where σ ranges over permutations of the N slots, y_i is the i-th ground-truth element and $\hat{y}_{\sigma(i)}$ the prediction assigned to it, and $\mathcal{L}_{\text{match}}$ scores a pairing by its class probability and box overlap. The minimum is found by the Hungarian algorithm in cubic time, which is negligible for $N = 100$. Having fixed the assignment, the actual training loss is the usual multi-task combination of Equation 10.4 applied to the matched pairs, with unmatched slots supervised towards $\emptyset$.

The one-to-one constraint is doing the work that non-maximum suppression used to do. If two slots both predict the same cat, at most one of them can be matched to it and the other is trained to say $\emptyset$ — so producing duplicates is directly penalised, and the model learns not to. Suppression stops being a post-process and becomes a property of the objective.

Two limitations follow from the design and are worth stating. N is a ceiling: an image with more objects than there are queries will have some of them go unreported, and the fix is to train with a larger N. And the class list is fixed at training time, as it is for everything else in this chapter

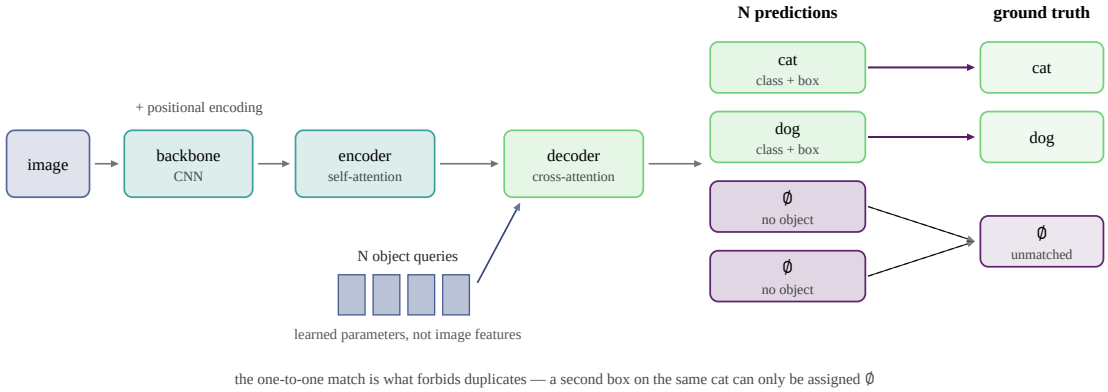


Figure 10.8.: DETR. A backbone and transformer encoder turn the image into a set of tokens; N learned object queries attend to those tokens and to one another, each emitting a class and a box. Training matches the N predictions to the ground-truth objects one-to-one via Equation 10.7, with the unmatched slots supervised towards the no-object class. *Drawn for these notes.*

— the output is a distribution over known categories plus $\emptyset$, with no mechanism for naming something never seen. Open-vocabulary detection exists, but it works by tying the output to a language embedding rather than to a fixed index, which is a later chapter's subject.

DETR is five years old and has largely been superseded, its slow convergence in particular having prompted a long line of successors. What it established is durable: detection can be posed as set prediction, and posing it that way removes the last hand-designed stages from the pipeline.

10.10. Instance segmentation for almost free

The fourth task of Section 10.1 is now nearly solved by accident. A detector already isolates individual objects and crops the features belonging to each; a segmentation network already turns features into a per-pixel map. Instance segmentation is what you get by putting the second inside the first.

Mask R-CNN is exactly that. Take Faster R-CNN unchanged, and add a third branch to the per-region head: a few convolutional layers over the RoI features that emit a small binary mask, 28×28 in the paper, saying which pixels inside the box belong to the object. The multi-task loss of Equation 10.4 gains a third term, a per-pixel binary cross-entropy over the mask.

One detail in that branch is a real design decision rather than a formality. The branch predicts C masks per region — one for every class — and the loss is applied only to the mask for the class the region actually belongs to. The alternative, one mask with a softmax over classes at each pixel, would make the mask branch compete with the classification branch: each pixel would have to decide *which* class it belongs to as well as whether it is foreground. Decoupling them means the mask branch answers only "is this pixel part of the object", which is a much easier question, and the paper attributes a substantial part of its accuracy to that choice.

This is also where the misalignment of RoI pooling stops being tolerable. A half-cell shift is invisible in a class label and obvious in a 28×28 mask stretched back over the object, which is why Equation 10.5 and RoI align were introduced by this paper rather than by Fast R-CNN, whose outputs were coarse enough not to care.

The same branch generalises without modification. Replacing the binary mask with K one-hot maps, one per body joint, and training each with a cross-entropy over its 28×28 positions turns Mask R-CNN into a pose estimator — the joint is a single pixel that is on, and everything else is off. No part of the architecture changes; only the shape of the target does, which is the chapter's argument in miniature.

10.11. What has the network learned

Detection and segmentation are what to ask a network for. The rest of the chapter is about what to ask *of* it: given a trained model and a prediction, what can be said about how it arrived there.

The linear classifier of Section 3.4 was interpretable for free. Each class's row of W has one weight per pixel, so reshaping the row to the image's dimensions gives a picture — a template the class score computes an inner product against, showing a blurry front-facing car for the car class and a two-headed smear for horse. That readability came from the model's weakness: a single linear map from pixels to scores has nowhere to hide.

The first convolutional layer of a deep network keeps that property, for the same reason. Its filters take three input channels, so a $64 \times 3 \times 11 \times 11$ weight tensor is 64 small RGB images and can simply be displayed. Every network trained on natural images produces the same thing there: oriented edges at a range of angles and scales, and blobs of opposing colour. AlexNet, ResNet-18, ResNet-101 and DenseNet learn first layers that are hard to tell apart, which is a strong statement about how little choice there is at that level — and the same family of oriented filters that Section 2.3 found in the visual cortex.

Every layer above the first ends there. A conv2 filter takes 64 input channels, and there is no way to render a $64 \times 5 \times 5$ tensor as something a person can look at. The weights of a deep network are, past the first layer, simply not viewable, and this is the fact that every technique in the remaining sections is working around. What can be examined instead is not the weights but the *behaviour*: what the network does with a particular image.

10.12. Which pixels mattered

The simplest such question has an answer already sitting in the machinery. Training computes $\partial L / \partial W$, the gradient of a loss with respect to the weights, holding the image fixed. Nothing stops the same backward pass from being run the other way: fix the weights, and compute the gradient of a class score with respect to the *pixels*,

$$M_{ij} = \max_{c \in \{R,G,B\}} \left| \frac{\partial s_y}{\partial I_{ij,c}} \right| \tag{10.8}$$

where s_y is the unnormalized score for the predicted class, I the input image, and the absolute value and maximum over the three colour channels collapse the result to one number per pixel. Simonyan, Vedaldi and Zisserman called the result a *saliency map*, and its meaning is the ordinary meaning of a derivative: it is large where a small change to that pixel would most change the score. Computing it costs one forward and one backward pass.

The same trick localises intermediate units rather than classes. Pick one channel at one position in a middle layer, treat its activation as the quantity to differentiate, and backpropagate to the

image; the result shows what that unit responds to in this particular picture. Plain gradients are noisy for this, so guided backpropagation modifies the backward pass through each ReLU to pass only positive gradients, discarding the negative evidence. The images that come out are dramatically cleaner and the modification is not a derivative of anything — it is a visualization heuristic, and it is worth being clear that this is a different kind of object from Equation 10.8.

That distinction generalises into the main caveat about this whole family of methods. A saliency map looks like an explanation, and the temptation is to accept it as one. Adebayo et al. applied an obvious test — randomise the network's weights and recompute the map — and found that several popular methods, guided backpropagation among them, produce nearly the same picture from a trained network and an untrained one. A method that cannot distinguish a working model from a random one is showing you image structure rather than model behaviour. Plain gradients pass that test; some of the prettier methods do not.

10.13. Where the evidence is

Saliency works in pixel space and inherits its noise. A different approach works in feature space, using the fact that a convolutional feature map is still laid out over the image: position (h, w) of the last conv layer corresponds to a known region of the input, so a map over those positions can be drawn on the photograph.

Class activation mapping exploits a particular architecture to make that map fall out of the arithmetic. Suppose the network ends with a K-channel feature map $f \in \mathbb{R}^{H\times W\times K}$, a global average pool, and one linear layer with weights $w \in \mathbb{R}^{K\times C}$ — the ending Section 7.8 described, which replaced VGG's fully connected stack. Then the score for class c is

$$S_c = \sum_k w_{k,c} F_k = \sum_k w_{k,c} \cdot \frac{1}{HW} \sum_{h,w} f_{h,w,k} = \frac{1}{HW} \sum_{h,w} \underbrace{\sum_k w_{k,c}\, f_{h,w,k}}_{M_{c,h,w}} \tag{10.9}$$

where the only step taken is exchanging the two sums. The bracketed quantity $M_{c,h,w}$ is a number per spatial position whose average over positions *is* the class score, so it decomposes the prediction across the image: it says how much position (h, w) contributed to concluding c. Upsampled to the input's size and drawn as a heatmap, it shows where the evidence for that class was found. Nothing was trained to produce this and no extra computation is needed; it is the score written in a different order.

The catch is the *suppose*. Equation 10.9 holds only when a global average pool is followed directly by one linear layer, and only for the last convolutional layer. Any other head — a couple of fully connected layers, anything at all between the pool and the scores — and the derivation collapses.

Grad-CAM removes the constraint by noticing what $w_{k,c}$ was doing. It is the sensitivity of the class score to channel k, and a sensitivity can always be obtained by differentiation whether or not it happens to be a weight. So pick any layer, with activations $A \in \mathbb{R}^{H\times W\times K}$; compute $\partial S_c / \partial A$ by backpropagation; average it over space to get one number per channel,

$$\alpha_k = \frac{1}{HW} \sum_{h,w} \frac{\partial S_c}{\partial A_{h,w,k}} \tag{10.10}$$

and use those in place of the weights:

$$M^c_{h,w} = \mathrm{ReLU}\left(\sum_k \alpha_k A_{h,w,k}\right) \tag{10.11}$$

The ReLU is there because only positive contributions are wanted — evidence *for* class c, not against it. On the architecture CAM requires, Equation 10.10 recovers exactly the weights CAM uses, so Grad-CAM is a strict generalisation: any layer, any architecture, including detection and captioning models whose output is not a class score at all.

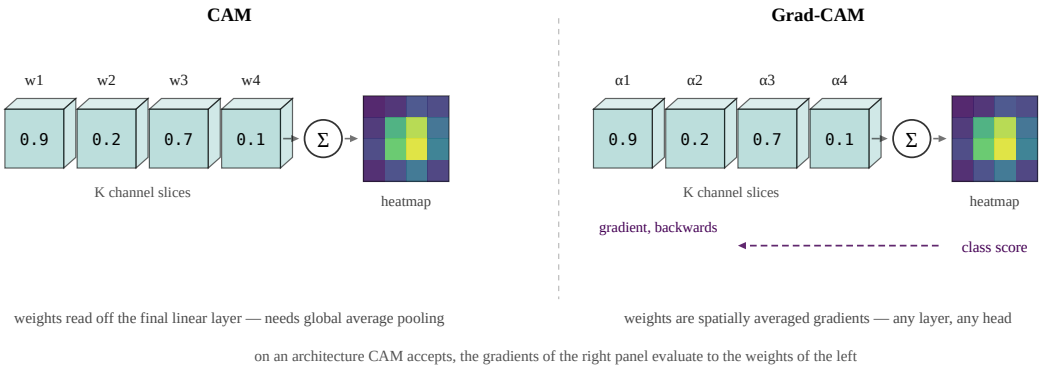


Figure 10.9.: CAM and Grad-CAM. Both weight the channels of a feature map and sum them into a spatial heatmap; they differ only in where the weights come from. CAM reads them off the final linear layer, which requires that layer to exist; Grad-CAM computes them as spatially averaged gradients of the class score, which requires nothing. *Drawn for these notes.*

Transformers make the whole exercise easier, and that is worth noting as a genuine architectural advantage rather than a footnote. A ViT computes attention weights as part of its forward pass, and those weights are already a distribution over image patches — Section 9.3 showed the same matrices doing the same job for translation. Reading which patches a token attended to needs no gradient, no auxiliary construction, and no architectural assumption; the model reports it. Self-supervised ViTs go further: the attention maps of a DINO-trained model segment the main object of a scene without ever having been shown a segmentation mask, which is a stronger claim about what the model represents than any post-hoc method could establish.

10.14. What this leaves

The chapter's through-line is that the interesting variable stopped being the architecture. Every model here runs the same backbone — a convolutional network or a ViT, pretrained on classification — and differs in the shape of what leaves it and the loss that supervises that shape. Segmentation makes the output a map and pays for it with upsampling. Detection makes the output a variable-length list and pays for it by predicting against a fixed set of references, whether those are proposals, anchors, grid cells, or queries. Instance segmentation is the two composed, and the composition needed one new operation and no new ideas.

The second thread is what happened to the hand-designed parts, and the sequence is unusually clean. R-CNN learned features but took its regions from selective search. Faster R-CNN learned the regions but kept anchors and non-maximum suppression. DETR removed both, at the cost of needing a loss that solves an assignment problem before it can be evaluated. At each step something that a person had specified was replaced by something the objective specifies instead,

and the pattern is worth recognising because it is not finished — the fixed class list is the next such component, and removing it is what vision–language models are for.

The visualization half is a different kind of subject and does not resolve as neatly. The honest summary is that the first layer can be looked at, nothing above it can, and everything else in Section 10.12 and Section 10.13 is an indirect measurement whose validity has to be argued rather than assumed. Grad-CAM is the most useful of them because it makes the fewest architectural demands and because the quantity it computes has a plain reading. The strongest result in the section is not a visualization method at all: it is that attention hands you the map for free, and that a model trained without labels puts it in the right place.

The next chapter adds a dimension. Everything so far has taken a single image; video introduces time, and with it the question of whether motion is a fourth axis to convolve over or something that needs its own machinery.

References

- J. Adebayo et al., "Sanity Checks for Saliency Maps," *NeurIPS*, 2018. arXiv:1810.03292
- M. Caron et al., "Emerging Properties in Self-Supervised Vision Transformers," *ICCV*, 2021. arXiv:2104.14294
- N. Carion et al., "End-to-End Object Detection with Transformers," *ECCV*, 2020. arXiv:2005.12872
- R. Girshick, "Fast R-CNN," *ICCV*, 2015. arXiv:1504.08083
- R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," *CVPR*, 2014. arXiv:1311.2524
- K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN," *ICCV*, 2017. arXiv:1703.06870
- T.-Y. Lin et al., "Focal Loss for Dense Object Detection," *ICCV*, 2017. arXiv:1708.02002
- W. Liu et al., "SSD: Single Shot MultiBox Detector," *ECCV*, 2016. arXiv:1512.02325
- J. Long, E. Shelhamer and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *CVPR*, 2015. arXiv:1411.4038
- A. Odena, V. Dumoulin and C. Olah, "Deconvolution and Checkerboard Artifacts," *Distill*, 2016. doi:10.23915/distill.00003
- J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *CVPR*, 2016. arXiv:1506.02640
- S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *NeurIPS*, 2015. arXiv:1506.01497
- O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *MICCAI*, 2015. arXiv:1505.04597
- R. R. Selvaraju et al., "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," *ICCV*, 2017. arXiv:1610.02391
- K. Simonyan, A. Vedaldi and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," *ICLR Workshop*, 2014. arXiv:1312.6034
- J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *ICLR Workshop*, 2015. arXiv:1412.6806
- J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers and A. W. M. Smeulders, "Selective Search for Object Recognition," *IJCV* 104(2):154–171, 2013. doi:10.1007/s11263-013-0620-5
- M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," *ECCV*, 2014. arXiv:1311.2901

- B. Zhou, A. Khosla, A. Lapedriza, A. Oliva and A. Torralba, "Learning Deep Features for Discriminative Localization," *CVPR*, 2016. arXiv:1512.04150

11. Video Understanding

Lecture 10

Based on Lecture 10 of CS231n, Stanford University, Spring 2025.

11.1. The axis that was missing

Every input so far has been a $3 \times H \times W$ array. A video adds one index, and the tensor becomes $T \times 3 \times H \times W$ — a stack of T frames, each an ordinary image. Nothing about that description is difficult. The difficulty is that the new axis is not like the other two, and treating it as though it were is the mistake most of this chapter is organised around avoiding.

Start with what changes in the task. Image classification asks what is in the picture, and the answer is a noun: a dog, a truck, a stethoscope. Video classification usually asks what is *happening*, and the answer is a verb — swimming, running, jumping, eating. That is not a cosmetic relabelling of the output vocabulary. A noun is a property of a single frame and can in principle be read off any one of them; a verb is a property of how frames relate, and there are pairs of actions that no single frame distinguishes. Opening a door and closing a door produce the same set of frames in opposite orders.

The loss does not change. A video classifier still emits C scores and is still trained with the softmax cross-entropy of Section 3.6 against a single action label. The entire question is how to produce those scores — how to turn a $T \times 3 \times H \times W$ tensor into a feature vector — and every architecture in this chapter is an answer to it.

11.1.1. Videos are big

Before any of those answers, an arithmetic problem that constrains all of them. Video is captured at around thirty frames per second, and at three bytes per pixel a minute of uncompressed standard-definition footage at 640×480 comes to roughly 1.5 GB. At 1920×1080 the same minute is about 10 GB. These are not numbers a training pipeline absorbs. A single ImageNet image is 150 KB; one minute of HD video is seventy thousand of them, and a dataset is measured in years of footage rather than minutes.

The memory problem is worse than the storage problem, because the input is the smallest thing on the GPU. A convolutional network holds activations for every layer of every example in the batch in order to backpropagate through them, so a tensor that is already too large to load comfortably becomes a multiple of itself once it is inside the network.

There is no clever compression that resolves this; the resolution is to give up on processing whole videos and shrink the input along both of its expensive axes at once. Temporally, sample a handful of frames per second rather than thirty — adjacent frames at 30 fps are nearly identical,

so most of that data is redundant by construction. Spatially, downscale to something like 112×112. A clip of $T = 16$ frames at 112×112 covers 3.2 seconds at 5 fps and occupies 588 KB, which is four ImageNet images and entirely manageable.

That reduction defines the unit of work for the rest of the chapter. **The model is trained to classify short clips, not videos.** Training samples clips from long videos with a sliding window and treats each as an independent labelled example. At test time the model is run on several clips drawn from the video — ten is a common choice — and their predicted probabilities are averaged into one answer. The clip is what the architecture sees; the video is assembled from clips outside the network.

The consequence worth holding onto is that this arrangement builds in a horizon. Whatever an architecture does with time, it does within a few seconds, and everything longer is handled by averaging, which is to say by an operation with no notion of order at all. Section 11.7 is about the attempts to push that horizon out.

11.1.2. Which frames

Sampling introduces a question that has no clean answer: which frames. The default is uniform or random sampling — take one frame every so often and hope the informative moments are among them. For a one-hour video of which four seconds matter, that is a poor bet, and the failure is silent, because the model still produces a confident average over the frames it happened to see.

Better strategies make sampling adaptive: look at one frame, use what it shows to decide where to look next, and spend the frame budget where the content warrants it. This is an active research area rather than a settled technique, and it returns in Section 11.13.1, where the sampler is itself a learned network.

11.2. The baseline that refuses to model time

The simplest thing that could possibly work is to ignore the fourth axis entirely. Take a 2D CNN, run it independently on each sampled frame as though the frame were a photograph, and average the predicted probabilities across frames. There is no temporal modelling anywhere in this pipeline. The network is an image classifier and does not know it is being shown a video.

This is worth stating carefully because the natural reaction — that a method which discards the defining property of the data cannot be competitive — is wrong, and reliably wrong. The single-frame CNN is a very strong baseline. Most action classes in most datasets are heavily correlated with appearance: a frame containing a swimming pool, goggles and water is enough to identify swimming without observing a single stroke. If a person is running, every frame shows a person in running posture on a track, and an image classifier will answer *running* for all of them.

The practical instruction that follows is the one to carry out of this section. Any new video architecture must be measured against the single-frame CNN, and it should be the first thing run rather than the last, because a method that beats it by a point after tripling the compute has not established that it is modelling motion — it has established that the dataset does not require motion to be modelled.

The baseline also makes the failure mode legible. What per-frame averaging cannot represent is anything that lives in the ordering of frames. It sees the same evidence for a door opening and a

door closing, for a weight being lifted and lowered, for a ball approaching and receding. Every architecture that follows is an argument about where in the network the temporal axis should be consumed, and each one buys a different amount of that ordering information at a different price.

11.3. Where to fuse

If the frames must be combined somewhere, the first design question is where. The single-frame CNN combines them after the softmax, which is as late as possible. Moving that point earlier is the axis along which the classical video architectures — all of them from Karpathy et al.'s 2014 study — differ from one another.

11.3.1. Late fusion

Run a 2D CNN on each frame, but stop before the classifier and keep the features. Each frame yields a feature map of shape $D \times H' \times W'$, and T frames give a $T \times D \times H' \times W'$ stack. Flatten that stack into one long vector of length $TDH'W'$ and feed it to an MLP, which maps it to the C class scores.

This is late fusion, and it is a genuine improvement on averaging, because the MLP sees all frames at once and can in principle learn that a particular feature appearing early and a different one appearing late means a specific action. What it costs is parameters. A concatenated vector over sixteen frames is sixteen times as long as a single frame's, and the first fully connected layer's weight matrix is proportional to that length. This is the same objection that removed fully connected layers from image classifiers, arriving in a more acute form.

The cheaper variant replaces concatenation with pooling. Average-pool the frame features over space *and* time to get a single D-dimensional clip feature, then apply one linear layer of shape $D \times C$. The parameter count no longer depends on T at all, which is the whole point. The cost is that pooling is order-blind in exactly the way averaging probabilities was: the mean of a set of feature vectors does not record which came first.

Both variants share a deeper limitation that the name is meant to signal. *Late* means the fusion happens after each frame has been through the full depth of a 2D CNN, and by that depth the features are semantic. A late-layer feature map answers questions like *is there a person here* and *is this a track surface*; it has long since discarded the pixel-level detail from which motion could be computed. The information that a runner's feet are moving up and down between adjacent frames exists only in the raw frames and the earliest feature maps. Fusing late means comparing representations from which the thing you wanted to compare has already been removed.

11.3.2. Early fusion

Push the fusion to the other extreme. Reshape the input from $T \times 3 \times H \times W$ to $3T \times H \times W$ — that is, reinterpret time as additional colour channels — and apply an ordinary 2D convolution whose input has $3T$ channels and whose output has D. Every filter in that first layer spans all T frames at once, so it can respond to a pattern that changes across them. After that layer the temporal axis is gone and the remainder of the network is a standard 2D CNN.

Early fusion has the property late fusion lacked: the comparison between frames happens on raw pixels, where motion is still visible. Its problem is the mirror image. A single convolutional layer, followed by a pointwise nonlinearity, is all the temporal processing the architecture will

ever do. One layer of linear combination across time is a weak model of how frames relate — it can detect that a region brightened, but it cannot compose such detections into anything hierarchical, because after that layer there is no time axis left to compose over.

Note also what "reinterpret time as channels" throws away. Convolution treats the channel axis as unordered: nothing in the layer knows that channel 7 comes one frame after channel 4 and one frame before channel 10. The temporal ordering survives only in whatever the learned weights happen to encode, and it must be learned separately for every filter.

So the two extremes fail for opposite reasons. Late fusion has plenty of depth for temporal reasoning but no low-level signal left to reason about; early fusion has the signal but only one layer in which to use it. What is wanted is fusion that is neither early nor late but distributed through the network — and that is 3D convolution.

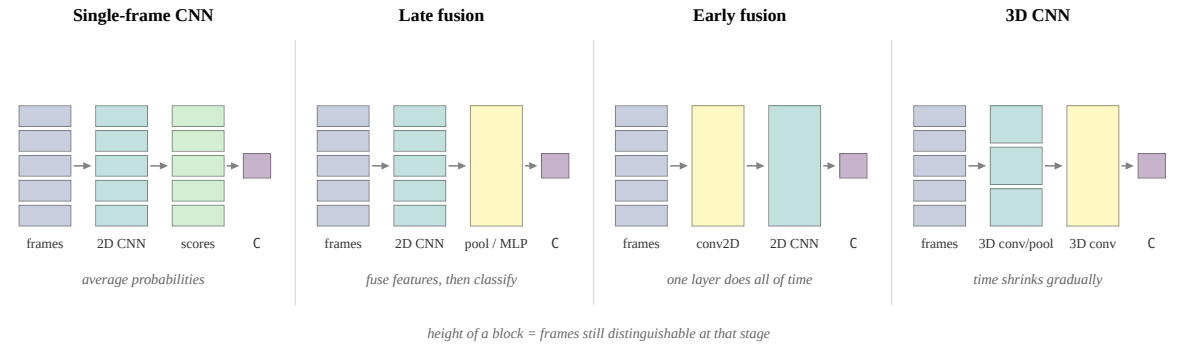


Figure 11.1.: Where each architecture consumes the temporal axis. Time is drawn as the vertical extent of each block; a block that is one frame tall has already collapsed it. The single-frame CNN never combines frames at all, late fusion combines them after the full 2D stack, early fusion at the first layer, and the 3D CNN reduces the axis gradually. Drawn for this project.

11.4. Slow fusion: the 3D convolution

Recall what a 2D convolution does. A 5×5 filter applied to a $3 \times 32 \times 32$ image has weights of shape $3 \times 5 \times 5$: it is local in the two spatial dimensions and spans the channel axis completely. Sliding it over all spatial positions produces one 28×28 activation map, and C_{out} such filters give C_{out} maps.

A 3D convolution changes exactly one thing. The input is $C_{\text{in}} \times T \times H \times W$ and the filter has shape $C_{\text{in}} \times 3 \times 3 \times 3$: local in *three* dimensions — time, height, width — and spanning the channel axis completely as before. It slides over all three, so the output is $C_{\text{out}} \times T \times H \times W$ and the temporal axis survives the layer. That is the crucial difference. A stack of such layers, interleaved with 3D pooling, shrinks T gradually rather than eliminating it, so temporal structure is built up hierarchically the way spatial structure already was. Karpathy's paper calls this **slow fusion**, and the name is the argument.

11.4.1. The three architectures side by side

The distinction is easiest to see in the receptive field, so take a toy network on a $3 \times 20 \times 64 \times 64$ clip and track both the tensor shape and the region of the input each output unit depends on.

Table 11.1.: Shapes and receptive fields for the three fusion strategies on a $3 \times 20 \times 64 \times 64$ clip. All three end with the same full-clip receptive field; they differ entirely in how they get there.

	Layer	Size ($C \times T \times H \times W$)	Receptive field ($T \times H \times W$)
Late	Conv2D(3×3, $3 \to 12$)	$12 \times 20 \times 64 \times 64$	$1 \times 3 \times 3$
	Pool2D(4×4)	$12 \times 20 \times 16 \times 16$	$1 \times 6 \times 6$
	Conv2D(3×3, $12 \to 24$)	$24 \times 20 \times 16 \times 16$	$1 \times 14 \times 14$
	GlobalAvgPool	$24 \times 1 \times 1 \times 1$	$20 \times 64 \times 64$
Early	Conv2D(3×3, $3{\cdot}20 \to 12$)	$12 \times 64 \times 64$	$20 \times 3 \times 3$
	Pool2D(4×4)	$12 \times 16 \times 16$	$20 \times 6 \times 6$
	Conv2D(3×3, $12 \to 24$)	$24 \times 16 \times 16$	$20 \times 14 \times 14$
	GlobalAvgPool	$24 \times 1 \times 1$	$20 \times 64 \times 64$
3D CNN	Conv3D($3 \times 3 \times 3$, $3 \to 12$)	$12 \times 20 \times 64 \times 64$	$3 \times 3 \times 3$
	Pool3D($4 \times 4 \times 4$)	$12 \times 5 \times 16 \times 16$	$6 \times 6 \times 6$
	Conv3D($3 \times 3 \times 3$, $12 \to 24$)	$24 \times 5 \times 16 \times 16$	$14 \times 14 \times 14$
	GlobalAvgPool	$24 \times 1 \times 1$	$20 \times 64 \times 64$

Read the temporal column of Table 11.1 downward and the three strategies separate cleanly. Late fusion holds the temporal receptive field at 1 through every layer and then jumps to 20 in a single step at the end: it builds slowly in space and all at once in time, at the end. Early fusion jumps to 20 in the first layer and holds it: slowly in space, all at once in time, at the start. The 3D CNN goes $3 \to 6 \to 14 \to 20$, growing in time at the same rate it grows in space. All three finish with the same $20 \times 64 \times 64$ receptive field, which is why the receptive field alone does not explain why one works better — the *path* does.

11.4.2. Why not just stack the frames

Early fusion and the 3D CNN both build a temporal receptive field, so it is fair to ask what is really different. The answer is the property that made convolution worth using in the first place.

Consider what early fusion's first layer looks like from the 3D point of view. Its weight tensor has shape $C_{\text{out}} \times C_{\text{in}} \times T \times 3 \times 3$: local in x and y, full extent in t. It slides over space and does not slide over time, because there is nothing left to slide over. So a filter that responds to a blue-to-orange transition between frames 4 and 5 has that timing baked into its weights. The same transition occurring between frames 14 and 15 excites a different set of weight positions entirely, and to detect it the network must learn a *second* filter that is a time-shifted copy of the first.

This is exactly the argument that produced convolution in Section 6.4, moved to a new axis. A fully connected layer on images has to learn a separate detector for a cat in each corner; convolution shares one detector across all positions and gets spatial shift invariance for free. Early fusion is, along the time axis, that fully connected layer. It has no temporal shift invariance,

and it pays for the lack in parameters and in sample efficiency, because every temporal pattern must be learned once per moment it can occur.

The 3D convolution's filter is $C_{\text{in}} \times 3 \times 3 \times 3$ — a local window in time as well as in space — and it slides over t. One filter detects the blue-to-orange transition wherever in the clip it happens. That is the whole benefit, and it is the reason 3D convolution is not merely early fusion with extra steps.

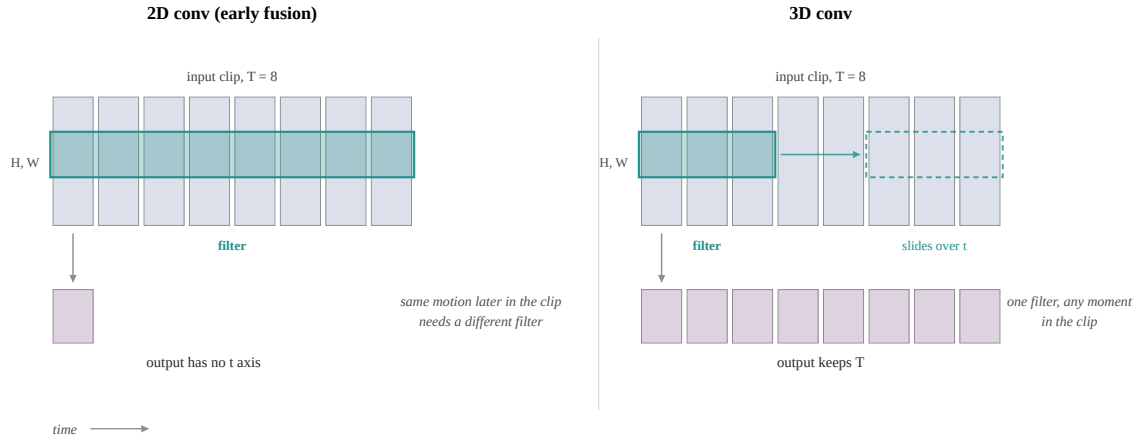


Figure 11.2.: Early fusion's first layer against a 3D convolution. Both see the same $C_{\text{in}} \times T \times H \times W$ input. The 2D filter spans all T frames and slides only over x and y, so its output has no temporal axis and the same motion at a different moment needs a different filter. The 3D filter spans three frames and slides over t as well, so one filter detects that motion anywhere in the clip. Drawn for this project.

The trade is compute, and it is steep. A 3D convolution keeps a T-fold larger activation tensor at every layer and multiplies the work per output by the kernel's temporal extent. Where a 2D network is expensive, the 3D version of it is roughly an order of magnitude worse, and that cost is what Section 11.9 and Section 11.13.1 spend their time trying to recover.

11.4.3. What the filters learn

Because a first-layer 3D filter spans both space and time — shape $3 \times 4 \times 5 \times 5$ for RGB over four frames — it can be displayed the way its input is displayed: as a short video clip rather than a still image. Doing so shows two populations. Some filters are the oriented edges and colour-opponent blobs familiar from 2D first layers, effectively static across their four frames; the network has rediscovered that appearance matters. Others are genuinely temporal — an edge pattern that translates from one side of the patch to the other over the four frames, which is a motion detector tuned to a direction and a speed. The mixture is informative: a 3D CNN does not become a motion model by construction, it allocates some of its capacity to motion and the rest to appearance, and how it splits that budget is decided by the dataset.

11.5. What the comparison actually showed

Architectures are arguments until somebody measures them, and the measurement that settled this one was run on **Sports-1M**: a million YouTube videos labelled with 487 sports categories. The label set is deliberately fine-grained — marathon and ultramarathon are separate classes — which is why results are usually quoted as top-5 accuracy rather than top-1.

The dataset also illustrates a problem peculiar to video. ImageNet can be distributed as images; a million videos cannot, so Sports-1M was released as a list of YouTube URLs. Users delete and edit their uploads, so the dataset has been decaying since publication and no two research groups have downloaded quite the same thing. Reproducibility in video benchmarking is worse than in image benchmarking for reasons that have nothing to do with the models.

Figure 11.3 gives the video-level top-5 accuracies for the four architectures, all trained by Karpathy et al. under matched conditions, with C3D added from the following year.

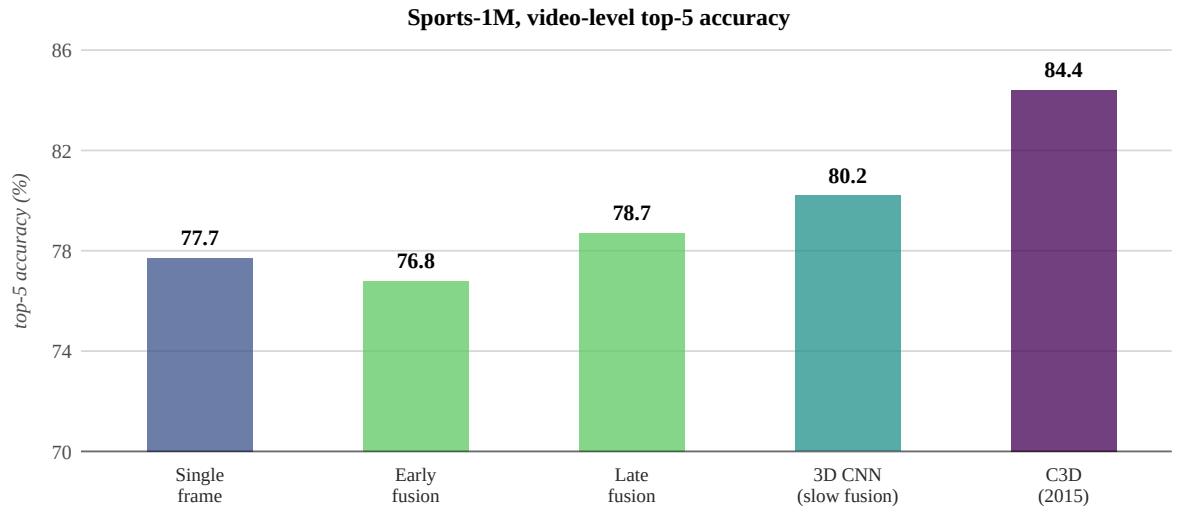


Figure 11.3.: Video-level top-5 accuracy on Sports-1M. The first four bars are from Karpathy et al., CVPR 2014 under matched training; C3D is from Tran et al., ICCV 2015. Replotted for this project from the published numbers.

Two things in that chart are worth sitting with. The first is that the single-frame CNN reaches 77.7%, and early fusion — an architecture built specifically to model motion — comes in *below* it at 76.8%. Adding a temporal mechanism made the model worse. Late fusion recovers a point over the baseline and slow fusion adds two and a half more, so the ranking does eventually favour temporal modelling, but the margin over an image classifier that has never heard of time is about 2.5 points for roughly an order of magnitude more computation.

The second is what that says about the benchmark rather than about the models. If appearance alone gets you to within 2.5 points of the best method available, the dataset is largely testing scene recognition — the pool, the track, the gym — and a genuinely better motion model has very little room in which to demonstrate that it is better. This is a recurring hazard in video research and one reason later benchmarks were built specifically to defeat single-frame classifiers.

11.5.1. C3D

The obvious next move after slow fusion is to stop designing bespoke video architectures and simply port a good image architecture across. C3D does exactly that with VGG: every 3×3 convolution becomes $3 \times 3 \times 3$, every 2×2 pool becomes $2 \times 2 \times 2$, and the whole stack is applied to a $3 \times 16 \times 112 \times 112$ clip. The one exception is the first pooling layer, which is $1 \times 2 \times 2$ — it downsamples space but not time, so that the sixteen frames survive intact into the second block rather than being halved before any temporal filter has looked at them.

C3D mattered less for its design than for its distribution. Training it on Sports-1M in 2014 required compute that almost no academic group had, so when Facebook released the trained weights, C3D became the video equivalent of a pretrained ImageNet network: extract a 4096-dimensional clip feature from FC6, fit a linear classifier on your own task, and you have a video model without owning a cluster. That is why it appears in so many papers of the period, and it took top-5 accuracy on Sports-1M to 84.4%.

The cost is the part to remember. A forward pass through AlexNet is 0.7 GFLOP and through VGG-16 is 13.6 GFLOP; C3D is 39.5 GFLOP, or 2.9× VGG-16, for a clip that is 112×112 rather than 224×224. Converting a 2D architecture to 3D by brute force multiplies its cost by roughly the temporal extent, and every subsequent development in this chapter is in some way an attempt to avoid paying that in full.

11.6. Motion as its own signal

There is a different objection to the 3D CNN, and it is not about cost. Space and time are being processed by the same undifferentiated machinery, on the assumption that a filter which learns to detect motion is as easy to learn as one that detects an edge. Human vision suggests that motion deserves its own treatment.

The evidence is Johansson's 1973 point-light experiment. Attach small lights to a person's joints, film them in the dark, and show a viewer only the moving dots. There is no appearance information whatsoever — no faces, no bodies, no scene — and viewers nonetheless identify walking, sitting, dancing, and often the actor's sex and mood, within a fraction of a second. Freeze a single frame and the percept collapses to a meaningless scatter of dots. Whatever is being recognised exists purely in how the dots move.

If motion alone suffices for humans, it is worth extracting motion explicitly rather than hoping a network discovers it.

11.6.1. Optical flow

The explicit representation is **optical flow**: a displacement field F between consecutive frames I_t and I_{t+1} that says, for every pixel, where that pixel went. Writing $F(x, y) = (dx, dy)$, the defining relation is

$$I_{t+1}(x + dx,\ y + dy) = I_t(x, y) \tag{11.1}$$

so F is a two-channel image the same size as the frame, one channel of horizontal displacement and one of vertical. Equation 11.1 is an assumption as much as a definition — it asserts that a pixel's brightness is preserved as it moves, which is the *brightness constancy* assumption, and it is what makes the problem solvable at all. It is also badly underdetermined on its own, since one equation per pixel cannot fix two unknowns, so every flow algorithm adds a smoothness prior of some kind. Computing flow well is its own literature; for present purposes it is a black box that turns two frames into a motion field.

Visualising the two channels separately makes clear what has been isolated. The horizontal channel lights up where things move sideways, the vertical channel where they move up and down, and both are essentially blank over static background regardless of how textured that background is. Flow is a representation from which appearance has been deleted.

11.6.2. Two-stream networks

Simonyan and Zisserman's two-stream network takes that separation literally and builds one CNN per signal. The spatial stream is an ordinary 2D CNN over a single RGB frame — the single-frame baseline of Section 11.2, unchanged. The temporal stream takes the optical flow computed between every adjacent pair in a T-frame clip, stacks the $2(T-1)$ flow channels into one tensor of shape $2(T-1) \times H \times W$, and runs a 2D CNN over it. Both streams produce class scores, and the scores are combined — by averaging, or by a linear SVM fitted on the concatenated softmax outputs.

Note that the temporal stream is early fusion. It collapses all of time into channels at the first layer, which Section 11.3.2 argued was a weakness. It is much less of a weakness here, because the input is already a motion representation: the first layer is no longer being asked to discover motion from raw pixels, only to combine motion measurements that were handed to it.

The results on **UCF-101**, a 101-class action dataset, are the reason this architecture was influential.

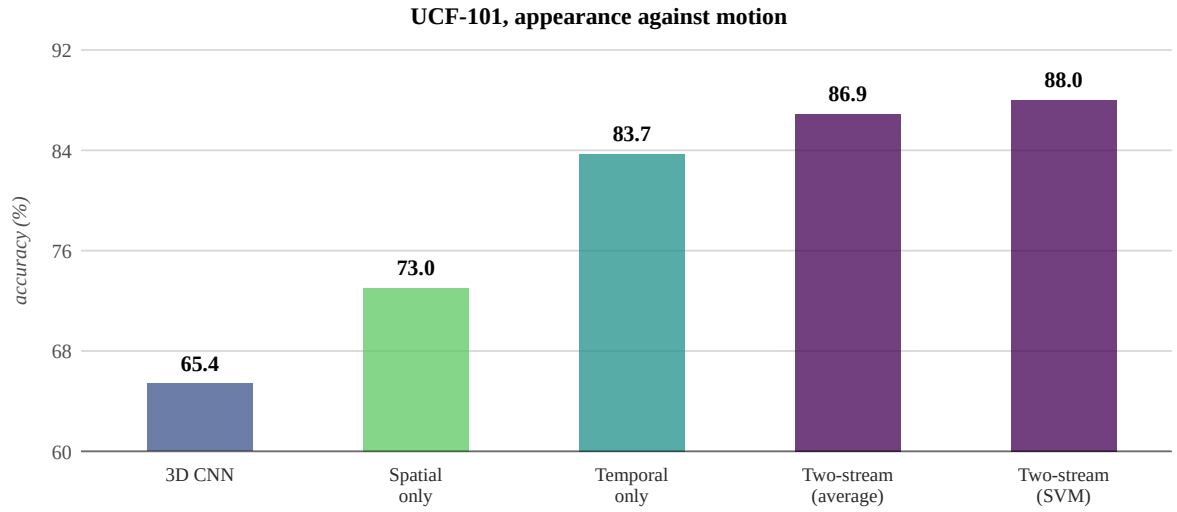


Figure 11.4.: Accuracy on UCF-101. The 3D CNN bar is Karpathy et al.'s slow-fusion network transferred to this dataset; the remaining four are from Simonyan and Zisserman, NeurIPS 2014. Replotted for this project from the published numbers.

The temporal stream on its own reaches 83.7%, more than ten points above the spatial stream's 73.0%. A network that has seen no colours, no textures and no objects — only displacement fields — beats a network that has seen the frames. That inverts the ordering Sports-1M suggested, and the likely reason is overfitting: appearance offers an enormous amount of information correlated with the label through the background scene, and with only 101 classes and 13,000 clips a spatial network will happily memorise gyms. Flow discards all of that and leaves the network only the part of the signal that generalises.

Combining the two adds another three to four points, ending at 88.0%, which is a straightforward statement that the streams are not redundant. Appearance and motion each carry information the other lacks.

The catch, and it is the reason two-stream networks eventually fell out of favour, is that optical flow is not free. It is computed by a separate non-learned algorithm as a preprocessing step, it is expensive, and it is not differentiable with respect to anything the network is training. The

pipeline is not end-to-end, and the flow algorithm's assumptions are baked in whether or not they suit the task.

11.7. Reaching past the clip

Every method so far operates inside a clip of two to five seconds, because that is what Section 11.1.1 established the memory budget allows. That horizon is not merely a practical limit; it is a modelling limit, and it excludes an entire category of content. Consider a video in which someone places a kettle on a hob, and two minutes later pours a cup of tea. A clip-level classifier sees each event but has no representation in which they are the same episode. Anything requiring that two events be related — narrative, causation, intention — is beyond a 3D CNN by construction, and averaging clip predictions at test time does not help, because averaging has no notion of before and after.

The apparatus for long sequences already exists. Section 8.2 built recurrent networks precisely to carry information across arbitrary distances, and the composition is immediate: run a CNN — 2D on frames or 3D on short clips, it does not matter — to produce a sequence of feature vectors, then run an LSTM over that sequence. If a single video-level label is wanted, take the prediction at the final timestep, a many-to-one arrangement. If per-frame output is wanted, read a prediction at every step. The idea predates AlexNet — Baccouche et al. had it in 2011 — but became standard with long-term recurrent convolutional networks in 2015.

Training this composite is awkward in practice. Backpropagating through a CNN unrolled over hundreds of timesteps exceeds memory long before it exceeds patience, so the usual approach is to pretrain the CNN — on ImageNet, or on clip classification — freeze it, precompute the features, and train only the recurrent part. That works, but it means the visual representation is never adapted to the temporal task.

11.7.1. The two ways of covering time

Comparing this design against 3D convolution sharpens what each mechanism contributes, and the comparison is cleanest as two independent properties.

A CNN over time is **convolutional but finite**: each output depends on a local window of input, the same weights are applied at every position, and no output can depend on anything outside its receptive field. An RNN is the opposite — **fully connected but infinite**: the hidden state in principle summarises everything seen so far, with no fixed horizon, but the state is updated by dense matrix multiplication that treats the feature vector as an unstructured list of numbers, discarding the spatial layout the CNN built.

Written that way, the missing quadrant is obvious. What is wanted is infinite temporal extent *and* convolutional structure — a recurrence whose state is a feature map rather than a vector, updated by convolution rather than by matrix multiplication.

11.7.2. Recurrent convolutional networks

That is the recurrent convolutional network, and the construction takes one line. The vanilla RNN update of Section 8.2 is

$$h_t = \tanh(W_h h_{t-1} + W_x x_t) \tag{11.2}$$

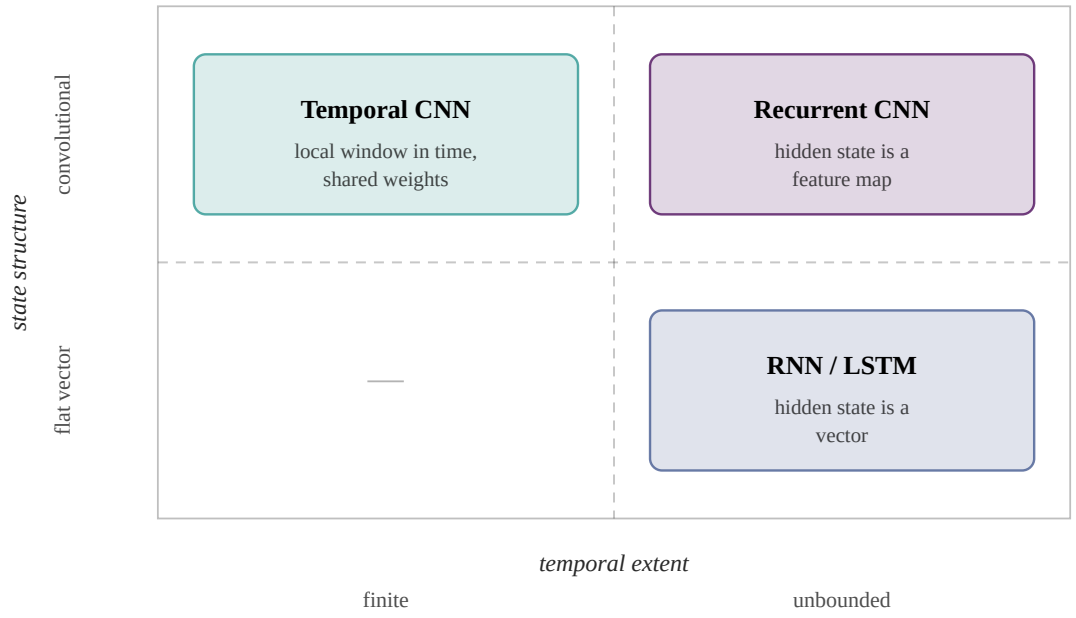


Figure 11.5.: The two properties a temporal mechanism can have, and the three architectures that occupy the corners. A temporal CNN is convolutional but sees a bounded window; an RNN sees everything but updates a flat vector by dense multiplication; the recurrent CNN takes one property from each. Drawn for this project.

where h_t and x_t are vectors and W_h, W_x are matrices. Replace every quantity with a feature map and every matrix multiplication with a 2D convolution:

$$h_t = \tanh(W_h * h_{t-1} + W_x * x_t) \tag{11.3}$$

Now h_t and x_t are tensors of shape $C \times H \times W$, W_h and W_x are convolution kernels, and $*$ is 2D convolution. The hidden state retains spatial structure throughout, and the recurrence is applied identically at every spatial position, so the layer is translation-equivariant in space and unbounded in time. Nothing about the substitution is specific to the vanilla RNN — the same replacement inside an LSTM's gates gives a convolutional LSTM, which is what is usually used in practice, for the gradient-flow reasons of Section 8.11.

Stacking these layers gives a grid indexed by depth and time, in which every feature map depends on two neighbours: the same layer at the previous timestep, and the previous layer at the same timestep. Spatial and temporal fusion happen together, everywhere in the network.

This is an elegant architecture and it is not widely used, for a reason that has nothing to do with what it represents. The recurrence is sequential: timestep t cannot be computed until $t-1$ is finished. Video sequences are long, GPUs are wide, and a computation that refuses to parallelise across its longest axis is the wrong shape for the hardware. The same objection retired recurrent networks from language modelling, and it retired them here.

11.8. Attention over space and time

The replacement is the one Section 9.6 already introduced. Self-attention relates every position to every other position in a single parallel operation: unbounded range like a recurrence, but with no sequential dependency at all. Applying it to video means treating the $T \times H \times W$ grid of feature vectors as the set of positions, so that a location in one frame can attend directly to a location in any other frame, at any spatial offset.

The non-local block packages this as a drop-in module. Its input is a feature map of shape $C \times T \times H \times W$ from whatever produced it, and it proceeds exactly as self-attention does, with $1 \times 1 \times 1$ convolutions playing the role of the query, key and value projections — a $1 \times 1 \times 1$ convolution is a per-position linear map on the channel axis, which is precisely what those projections are.

Reshape queries and keys to $C' \times THW$, multiply to get a $THW \times THW$ matrix of affinities, softmax it, and use the result to take a weighted combination of the values. A final $1 \times 1 \times 1$ convolution maps the C' channels back to C so the output can be added to the input as a residual. That residual wrapping is what makes the block droppable into an existing 3D CNN: initialise the last convolution's weights to zero and the block starts as the identity, so inserting it into a trained network cannot make things worse before training has adjusted it.

The name is a nod to the non-local means algorithm in image denoising, and it captures the point. A convolution is local by definition; this block is not, and interleaving a few of them into a 3D CNN gives a network whose early layers build local spatiotemporal features cheaply and whose non-local blocks relate those features across the whole clip.

The cost is the one Section 9.8 identified, now with a larger index set. The affinity matrix is $THW \times THW$, so memory grows as the square of the number of space-time positions. A $16 \times 14 \times 14$ feature map has 3,136 positions and a comfortable ten million affinities; the same clip at 56×56 has 50,176 positions and 2.5 billion. This is why non-local blocks are inserted in the deep, low-resolution stages of a network and never near the input.

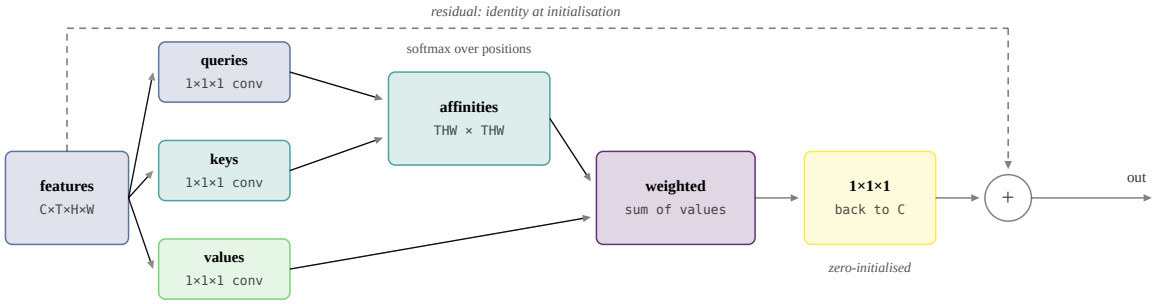


Figure 11.6.: The non-local block. Three $1 \times 1 \times 1$ convolutions project the input into queries, keys and values; the affinity matrix relates every one of the THW space-time positions to every other; a fourth $1 \times 1 \times 1$ convolution restores the channel count so the result can be added back as a residual. Drawn for this project.

11.9. Recycling the image networks

A question has been deferred through all of this: when an architecture calls for "a 3D CNN", which one? The image side of the field spent years discovering that depth matters, that residual connections make depth trainable, and that Inception's parallel branches are an efficient way to spend parameters. Rediscovering all of it on video — where the datasets are smaller, the training runs longer, and the compute per experiment is an order of magnitude higher — would be an expensive way to arrive at the same answers.

Inflation is the shortcut. Take a 2D architecture and replace every $K_h \times K_w$ convolution and pooling layer with a $K_t \times K_h \times K_w$ version, leaving the topology, the channel counts and the branching structure untouched. Carreira and Zisserman did this to Inception and called the result **I3D**, for Inflated 3D ConvNet. The inflated Inception block is the original block with every filter given a temporal extent.

That transfers the architecture. The more useful trick transfers the *weights*. A trained 2D kernel has shape $C_{\text{in}} \times K_h \times K_w$ and its inflated counterpart needs shape $C_{\text{in}} \times K_t \times K_h \times K_w$, so copy the 2D kernel K_t times along the new axis and divide every copy by K_t. The scaling is what makes this exact rather than approximate. Feed the inflated kernel a *constant* video — the same frame repeated K_t times — and each temporal slice contributes $\frac{1}{K_t}$ of the original 2D response, so the K_t contributions sum to precisely the 2D output. **Inflation initialises the 3D network at a point where it computes exactly what the 2D network computed on a still image.**

That is a much better starting point than random initialisation, because it means the video network begins training already able to recognise objects, and the gradient's job is to teach it about motion rather than about vision from scratch. Everything ImageNet paid for is inherited. Fine-tuning on video proceeds normally from there.

The measurements, on **Kinetics-400**, show how much of the improvement comes from the transfer rather than from the architecture:

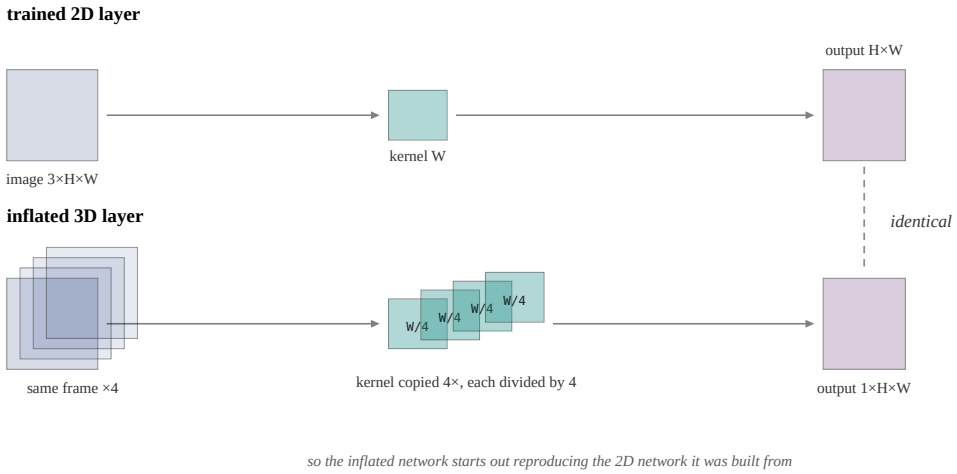


Figure 11.7.: Why the $1/K_t$ scaling makes inflation exact. Copying the trained 2D kernel along the new axis and dividing each copy by K_t means that on a constant video — one frame repeated K_t times — the K_t temporal contributions sum to precisely the original 2D response. Drawn for this project.

Table 11.2.: Top-1 accuracy on Kinetics-400, all built on the same Inception backbone, from Carreira and Zisserman, CVPR 2017.

Model	From scratch	ImageNet-pretrained
Per-frame CNN	57.9	62.2
CNN + LSTM	53.9	63.3
Two-stream CNN	62.8	65.6
Inflated CNN (I3D, RGB)	68.4	71.1
Two-stream inflated CNN	71.6	74.2

Every row in Table 11.2 improves under pretraining, and the CNN+LSTM improves by nearly ten points — it is the model with the most parameters downstream of the visual features and therefore the most starved of data. Inflation adds five to six points over the two-stream network at either level of pretraining, and adding the flow stream on top of inflation still helps by three points, which says the two ideas are addressing different deficits. Note also that the top row is again the single-frame baseline, still within twelve points of the best result on a dataset built specifically to require temporal reasoning.

11.9.1. Transformers, and where the numbers went

Inflation applies just as well to Vision Transformers, and the video transformer literature is largely a set of answers to the cost problem that Section 11.8 identified: full attention over $T \times H \times W$ tokens is quadratic in a very large number. TimeSformer and ViViT factorise the attention, alternating a layer that attends only within a frame with one that attends only across frames at fixed spatial position, which turns $O((THW)^2)$ into $O(T^2HW + T(HW)^2)$. Multiscale Vision Transformers instead pool tokens as depth increases, reintroducing the resolution-for-channels trade that convolutional networks always had. And VideoMAE sidesteps the labelling bottleneck by pretraining on masked reconstruction, which needs no labels at all and so can consume far more video than Kinetics contains.

Tracking top-1 accuracy on Kinetics-400 across that decade gives the shape of the progress: 62.2 for the per-frame CNN, 63.3 with an LSTM, 65.6 for two-stream, 71.1 for I3D, 74.2 for two-stream I3D, 79.8 for SlowFast with non-local blocks, 86.1 for MViTv2-L, and 90.0 for VideoMAE V2-g. The single-frame baseline that opened this chapter has been beaten by nearly thirty points — but it took ten years, and most of the last fifteen points came from pretraining strategy rather than from a better model of motion.

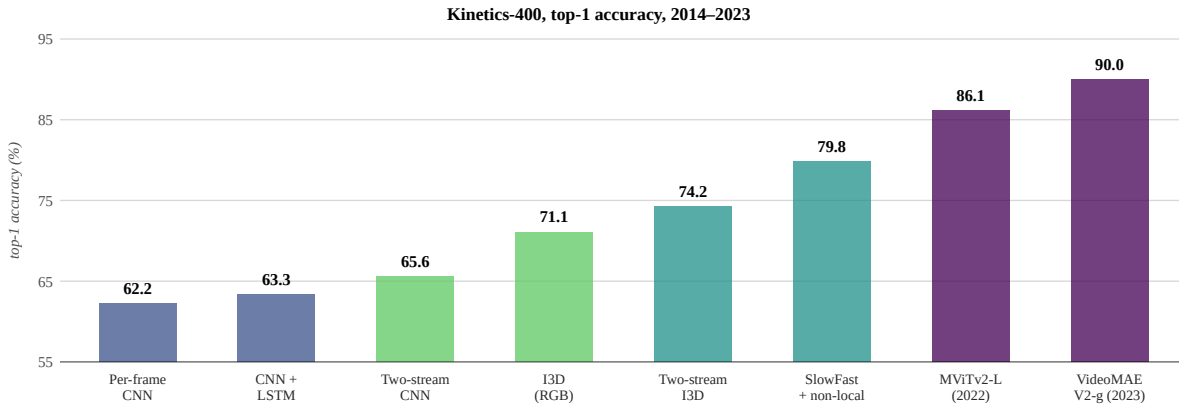


Figure 11.8.: Top-1 accuracy on Kinetics-400, 2014 to 2023, replotted for this project from the published numbers of each paper. All figures are for a single model with the evaluation protocol each paper reports.

11.10. What a video model sees

Section 10.11 optimised an input image to maximise a class score, and the same procedure works here with one adjustment. Applied to a two-stream network it produces two things at once: an appearance image and a flow field, each optimised to make the network as confident as possible about a chosen action.

The adjustment is a regulariser on the flow, penalising spatial roughness. Tuning the strength of that penalty selects the timescale of the motion that appears: a strong penalty forces the optimised flow to be spatially smooth and slowly varying, and what emerges is the action's slow component; a weaker one lets fast, localised motion dominate.

Run on *weightlifting*, the three panels are separately interpretable in a way that is unusual for feature visualisation. The appearance image shows gym-like texture. The slow-motion field shows a horizontal oscillation — the bar shaking. The fast-motion field shows a vertical push, the lift overhead. The network has decomposed the action into a static context and two motion components at different timescales, and none of that structure was designed in; it is what maximising the class score produced. That is reasonably strong evidence that the temporal stream is modelling motion rather than exploiting a correlation with the scene.

11.11. Past the clip label

Clip classification assumes the video has been trimmed so that one action fills it. Real footage is untrimmed: an hour of it contains long stretches of nothing in particular and short intervals that matter.

Temporal action localisation asks for those intervals — given a long untrimmed video, output the start and end frames of each action along with its class. The structural analogy to detection in Section 10.5 is exact, with one axis instead of two, and so is the solution: generate temporal proposals, then classify and refine each one, which is Faster R-CNN with 1D intervals in place of 2D boxes.

Spatio-temporal detection asks for both at once: find every person in the video, in space and in time, and say what each of them is doing at each moment. This is the task the AVA dataset annotates, and it is where video understanding stops resembling classification. Several people in one frame may be doing different things; one person's action changes over the course of a shot; the output is a set of box-plus-interval-plus-label triples with no fixed size. Every complication that made detection harder than classification recurs, with a fourth dimension added.

11.12. The modality that was left out

Everything above treats a video as a silent stack of frames, and the omission is larger than it looks. A recording of a baby laughing while a dog barks is not adequately described by its pixels; strip the audio and the content is not merely diminished but in places unrecoverable.

That audio is not a redundant channel — that it changes what is *seen* — is established by the McGurk effect. Play a recording of the syllable /ba/ over video of a speaker's lips forming /ga/, and a listener hears neither: they hear /da/, or /fa/, depending on the stimulus. Closing their eyes restores /ba/ immediately. The percept is a fusion of the two streams that occurs before anything conscious, and it cannot be suppressed by knowing about it. Vision and hearing are not two independent estimates of the world that get compared at the end; they are combined early, the way Section 11.3 asked architectures to combine frames.

The task that shows off the combination most directly is **visually-guided audio source separation**. Given a recording with several overlapping sources — two people talking at once, several instruments playing together — recover each source separately, using the video to say which is which. Audio-only separation is famously ill-posed: a microphone sums the sources and the sum has no marker of what came from where. Vision resolves the ambiguity because it says where the sources are and what they are doing. VisualVoice separates overlapping speakers by attending to their lip motion; co-separation separates instruments by associating each sound component with the visible object that produces it, trained on a hundred thousand unlabelled multi-source clips without ever being told which sound belongs to which instrument.

For classification, audio enters the same way images did. Convert the waveform to a spectrogram, which is a 2D array with time on one axis and frequency on the other, cut it into patches, and feed the patches to a transformer alongside the video patches. Once both modalities are token sequences, everything from Section 9.10 applies unchanged, and the design questions become where to let the streams attend to one another and how much cross-modal capacity to allow. Masked autoencoding extends just as directly: mask patches of both the frames and the spectrogram, and train the model to reconstruct them.

11.13. Where the field is going

Three directions close the lecture, and they share a premise: the clip classifier of the previous sections is a component, not a system.

11.13.1. Efficiency

The first is forced by the arithmetic of Section 11.1.1. An hour of video contains hundreds of clips and running a 3D CNN on every one is not viable at scale, so the work splits along two lines. One makes the per-clip model cheaper — X3D does this by starting from a small 2D network and expanding it along one axis at a time (frames, resolution, width, depth), keeping whichever expansion buys the most accuracy per FLOP. The other declines to process most clips at all: SCSampler learns to predict which clips are worth looking at and runs the expensive classifier only on those, which is the adaptive sampling promised back in Section 11.1.2, and Listen to Look uses audio as the preview signal, on the reasoning that a spectrogram is orders of magnitude cheaper to process than the frames it accompanies and is often enough to tell whether anything is happening.

11.13.2. Egocentric video

The second is a bet on hardware. If head-mounted cameras become common, the dominant source of video will not be edited third-person footage but continuous first-person streams from a camera that moves with the wearer's head and arrives with a microphone array rather than a single channel. That changes the problems worth solving — inferring who is speaking to whom and who is attending to whom is a natural question for a wearable device and an odd one for a video-search engine — and it makes multi-channel audio a first-class input, since a microphone array carries direction and direction is what disambiguates a conversation.

11.13.3. Video and language models

The third is currently the most active, and the move is the one that produced vision-language models generally: tokenise the video, project the tokens into a language model's embedding space, and let the model answer questions about what it was shown in text. That reframes video understanding away from a fixed label set and towards open-ended description — the difference between a network that emits one of 400 Kinetics classes and one that can be asked what the person on the left is doing and why.

11.14. What this leaves

Which returns to where the chapter began. A fourth axis was added to the input tensor, and the entire progression from single-frame averaging through fusion, 3D convolution, optical flow, recurrence, attention and inflation was a sequence of proposals about what to do with it. The proposals that survived share a property worth naming: none of them treats time as just another spatial dimension, and none treats it as entirely separate either. Slow fusion grows the temporal receptive field alongside the spatial one but with its own kernel size; the two-stream network gives motion its own network but fuses the scores; factorised attention attends over space and time in alternating layers. Time is like the other axes, but not enough like them to be ignored — and locating that difference precisely is what the last decade of video architectures has been doing.

Part III.

Systems

12. Large-Scale Distributed Training

Lecture 11

Based on Lecture 11 of CS231n, Stanford University, Spring 2025.

12.1. Twenty-five thousand computers, one model

Every chapter up to this one has described a computation that fits on a single device. A network is a function, the gradient of its loss is computed by backpropagation, and an optimizer applies that gradient to the weights. Nothing in that account mentions where the weights live, because for the first decade of the field the answer never had to be interesting: they lived in the memory of one GPU, and if they did not fit you trained a smaller model.

That assumption has quietly become false for every model at the frontier, and undoing it is not a matter of engineering hygiene. The gradient of a loss is defined over a batch; if the batch is spread across sixteen thousand devices, then computing that gradient is a communication problem before it is an arithmetic one, and the algorithms in this chapter exist because the communication is what runs out first.

The running example throughout is Llama 3 405B, and the choice is made on grounds of disclosure rather than quality. GPT-4 marked the point at which the industry stopped publishing: its technical report states outright that it "contains no further details about the architecture (including model size), hardware, training compute, dataset construction, training method, or similar," and essentially every frontier lab has followed. Meta's Llama 3 paper is the exception that makes a chapter like this possible — it describes the cluster, the parallelism configuration, the utilisation achieved, and even the failure rate.

Four numbers set the problem. The model has 405 billion parameters. It was pre-trained on 15.6 trillion text tokens. The pre-training compute budget was 3.8×10^{25} floating-point operations. And it ran on up to 16,384 H100 GPUs, drawn from a cluster of about 24,000.

Those numbers are not independent, and the relation between them is the single most useful piece of arithmetic in the chapter. For a dense transformer, a forward pass costs roughly two FLOPs per parameter per token — one multiply and one add for each weight the token passes through — and the backward pass costs about twice the forward. The total is

$$C \approx 6ND \tag{12.1}$$

where N is the parameter count and D the number of training tokens. Substituting Llama 3's figures gives $6 \times 4.05 \times 10^{11} \times 1.56 \times 10^{13} = 3.79 \times 10^{25}$, which is the published budget to two significant figures. The rule is crude — it ignores attention's quadratic term, normalisation, and the embedding layers — and it is accurate enough that it is how training runs are actually planned.

Equation 12.1 also says what the rest of this chapter is for. Divide the budget by what the hardware can deliver: 16,384 GPUs sustaining 400 TFLOP/s each is 6.6×10^{18} FLOP/s, so 3.8×10^{25} operations take about 5.8 million seconds, or roughly two months of wall-clock time at full utilisation. That figure is consistent with the paper, which reports statistics over a 54-day snapshot of pre-training. The entire difficulty is in the phrase *at full utilisation*, because the naive way of using sixteen thousand GPUs achieves a small fraction of it, and every technique below is a way of clawing some of that fraction back.

It is worth registering what a run of that length means physically before treating the cluster as an abstraction. Over those 54 days Meta recorded 466 job interruptions, of which 419 were unexpected; about 78% were traced to hardware, and GPU failures alone accounted for 58.7% of them. That is roughly eight unplanned failures a day on a machine that has to behave as one computer for two months. Only three of them required manual intervention. The rest is the price of building something out of twenty-five thousand physical objects.

12.2. Inside one GPU

A GPU is not a fast CPU. It is a device whose entire design is organised around doing the same arithmetic to a great many operands at once, and the two facts that matter for training — the memory hierarchy and the tensor cores — both follow from that.

Take the H100, which is the workhorse of the current generation. At the coarsest level it is a die of compute surrounded by 80 GB of high-bandwidth memory, connected by a bus that moves about 3,352 GB/s. That number sounds enormous until it is compared against the compute it feeds. Inside the compute die sits 50 MB of L2 cache, three orders of magnitude smaller than the HBM but far closer to the arithmetic units, and beneath that, 132 streaming multiprocessors, each with 256 KB of L1 cache and 256 KB of registers.

The pattern is the point, and it recurs at every scale in this chapter: **memory that is larger is further away, and memory that is closer is faster and smaller.** Writing a fast kernel is largely the discipline of staging data down that hierarchy and reusing it while it is close, and — as Section 12.3 shows — writing a fast distributed training job is the same discipline applied to a hierarchy that extends past the boundary of the chip.

A detail of manufacturing is worth a sentence because it explains a number that otherwise looks arbitrary. The H100 die physically contains 144 streaming multiprocessors, but only 132 are enabled. Chips of this transistor count never come off the line perfect, so vendors *bin* them: the design targets 144, the product promises 132, and the dies with a handful of defective units are sold rather than discarded. Nearly every published core count in this field is a yield-adjusted floor, not a description of the silicon.

12.2.1. Where the throughput actually comes from

Zoom into one streaming multiprocessor and there are two kinds of arithmetic unit, with wildly different capabilities.

The first is a bank of 128 FP32 cores. Each computes $ax + b$ for scalars a, x, b in one clock cycle — a fused multiply-add, counted as two floating-point operations. One SM running its FP32 cores therefore retires $128 \times 2 = 256$ FLOP per cycle.

The second is four *tensor cores*, and the name undersells them: they are matrix cores. Each performs one small matrix multiply-accumulate per clock, on the H100 taking a 16×4 input

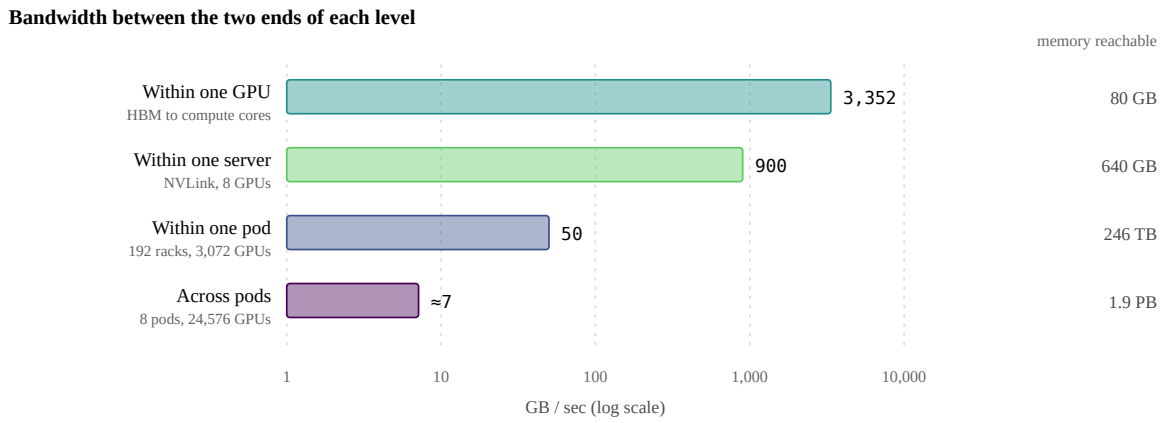


Figure 12.1.: The four levels of the hierarchy for which the Llama 3 cluster publishes a bandwidth, with the memory reachable at each printed at the right. Every step outward multiplies the capacity available and divides the rate at which it can be reached. The cross-pod figure is derived rather than published: the paper gives a 1:7 oversubscription ratio at the aggregation layer, applied here to the pod bandwidth. Drawn for this project from the paper's network description.

against a 4×8 input and adding a 16×8 accumulator. Counting a multiply and an add for each of the $16 \times 4 \times 8$ scalar products gives 1,024 FLOP per tensor core per cycle, so four of them retire 4,096 FLOP per cycle:

$$\underbrace{16 \times 4 \times 8 \times 2}_{\text{1024 FLOP per tensor core}} \times 4 = 4096 \quad \text{versus} \quad \underbrace{128 \times 2}_{\text{FP32 cores}} = 256 \tag{12.2}$$

A factor of sixteen, on the same silicon, in the same cycle. It is not an exaggeration to say that the tensor cores *are* the device and the FP32 cores are what runs when you have made a mistake.

The catch is precision. Tensor cores operate in mixed precision: the inputs arrive as 16-bit floats, the multiplications happen at 16 bits, and the accumulation is carried at 32 bits so that summing many products does not destroy the result. This is why a model that is not cast to a 16-bit dtype silently runs on the FP32 path instead — and why the symptom is not an error but training that is inexplicably twenty times slower than expected. Mixed precision is not an optimisation applied to a working training script; it is the condition under which the hardware is being used at all.

12.2.2. A thousandfold, with an asterisk

The generational numbers make the last decade legible. The K40 of 2013 delivered 5 TFLOP/s of FP32. The P100 reached 10.6. The V100 of 2017 was the first device with tensor cores and offered 125 TFLOP/s on them against 15.7 in FP32. The A100 followed with 312 and 19.5, the H100 with 989 and 67, and the B200 with roughly 5,000 and 83.3.

The headline conclusion drawn from that series — a thousandfold speedup in twelve years — deserves an asterisk, because it compares the B200's tensor cores against the K40's FP32 cores. Measured like against like, general-purpose floating-point throughput grew by about seventeen times, from 5 to 83.3 TFLOP/s, which is roughly what Moore's law would predict over that

span. The remaining factor of sixty came from two decisions rather than from transistor density: spending die area on units that do nothing but matrix multiply-accumulate, and halving the precision of the operands.

That is a more interesting fact than the headline, because it says the speedup was bought by *specialising* — by building hardware that is fast only for a narrow computation and then reorganising deep learning to consist of that computation. Architectures that reduce to large dense matrix multiplies ride the curve; architectures that do not are stranded on the seventeenfold line. The dominance of the transformer, which is very nearly nothing but matrix multiplies, is not unrelated.

12.3. From a chip to a cluster

Set the single device aside and consider the machine that Llama 3 was actually trained on, because the shape of the parallelism strategies in the second half of this chapter is dictated almost entirely by its wiring.

Inside one H100, memory reaches the cores at 3,352 GB/s. Eight of those GPUs sit in one server, connected to each other by NVLink at about 900 GB/s — roughly a quarter of on-device bandwidth, and the first cliff. Two servers make a rack of 16 GPUs. 192 racks are joined by cluster switches into a *pod* of 3,072 GPUs with full bisection bandwidth, at which point any GPU can reach any other at about 50 GB/s. That is another factor of eighteen down. Eight pods make the full cluster of 24,576 GPUs, and here the network is deliberately oversubscribed at a ratio of 1:7, so cross-pod bandwidth is materially worse than 50 GB/s.

Read as one object, that cluster has 1.875 PB of GPU memory, 415 million FP32 cores, 13 million tensor cores, and a peak of 24.3 EFLOP/s — 2.43×10^{19} floating-point operations per second. The productive way to think about it is not as twenty-five thousand computers but as one computer with a four-level memory hierarchy bolted onto the four levels already inside each chip, and with the same rule holding across all eight: **further away means larger and slower.**

The consequence is the organising principle of everything that follows. A computer does two things — it computes, and it moves bits from one place to another. Compute is what you want; communication is overhead. There are only two ways to reduce the damage that communication does. The first is to *hide* it, by arranging for bits to be in flight during a stretch of arithmetic that does not need them yet. The second is to *place* it, by assigning the communication-hungry parts of the algorithm to the links that are fast and the communication-light parts to the links that are slow. Every technique in the remainder of this chapter is one or both of those moves.

Two footnotes on the hardware landscape. NVIDIA is dominant but not alone: Google's TPUs are in their sixth generation, with the v5p offering 459 TFLOP/s of BF16 per chip and 95 GB of memory, arranged into pods of up to 8,960 chips, and Google's own frontier models are almost certainly trained on them. The difference is commercial rather than technical — TPUs cannot be bought, only rented. AMD's MI325X (1,300 TFLOP/s BF16, 256 GB) and AWS's Trainium2 (667 TFLOP/s BF16, 96 GB, packaged 64 to an UltraServer) have competitive datasheets and much less software gravity behind them. The second footnote is that the interconnect numbers above are the Llama 3 cluster's specifically; the topology generalises, the exact bandwidths do not.

12.4. Four axes and a budget

The rest of the chapter is about splitting a transformer across that machine, and it helps to know in advance that there are only four places to cut.

A transformer is a stack of L layers, each operating on a tensor of shape (B, S, D) — a minibatch of B sequences, each of S tokens, each token a vector of dimension D. That is four indices, and each one is an axis of parallelism with a name:

Axis	Name	Abbrev.
batch B	data parallelism	DP
sequence S	context parallelism	CP
feature D	tensor parallelism	TP
layers L	pipeline parallelism	PP

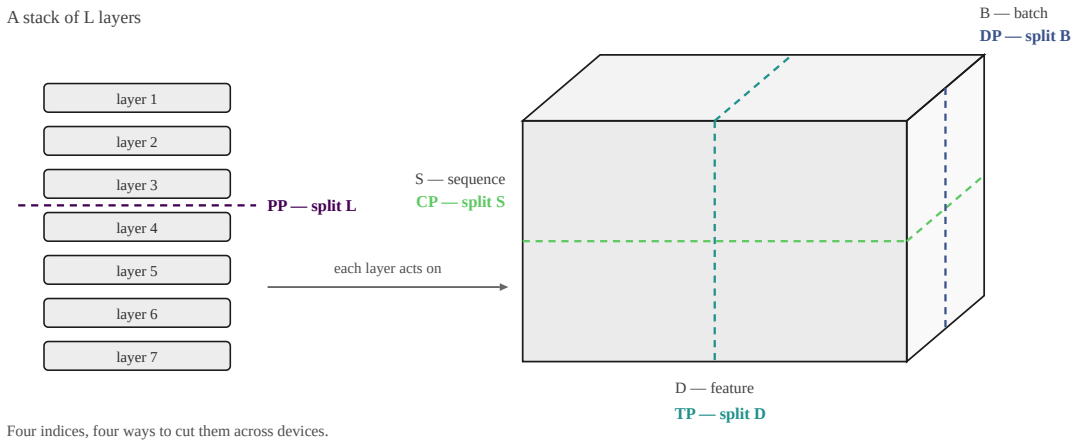


Figure 12.2.: The four axes of a transformer's computation and the parallelism strategy that splits each. A layer's activations form a (B, S, D) tensor and the model is a stack of L of them; every distributed training scheme in this chapter is a choice of how many ways to cut along each of these four indices. Drawn for this project.

The names are opaque and the underlying idea is not: they are four ways of cutting the same block of computation, and a real training run cuts along all of them at once. What decides how much of each is the memory budget, so it is worth doing that arithmetic now.

Storing a trained model requires one number per parameter. *Training* one requires four or five. There is the weight itself; the gradient with respect to it; and, for Adam, the first and second moment estimates — one exponentially weighted average of the gradient and one of its square. Some setups additionally keep an exponential moving average of the weights for evaluation. At 16-bit precision each of these is two bytes, so a lower bound is

$$\text{bytes per parameter} = \underbrace{4}_{\text{weight, grad, } m,v} \times \underbrace{2}_{\text{bytes at 16-bit}} = 8 \tag{12.3}$$

A billion parameters therefore occupies 8 GB before a single activation is stored, and an 80 GB H100 is full at around ten billion. Llama 3 405B needs some 3.2 TB by this accounting — forty H100s' worth of memory to hold the state of a model that has not yet been given any data to process.

This is the wall. Every technique from here on is a response to some part of it: Section 12.5 accepts it and scales the batch instead, Section 12.6 attacks the eight bytes directly, Section 12.8 attacks the activations that Equation 12.3 ignores, and Section 12.10 through Section 12.12 attack the assumption that one layer's computation has to happen on one device.

12.5. Data parallelism

The easiest axis to cut is the batch, and the reason is one line of algebra.

Training minimises a loss that is an average over the minibatch, and the gradient of an average is the average of the gradients. Write the macro-batch as MN samples $x_{i,j}$, indexed so that i runs over M devices and j over the N samples each device holds. Then

$$\frac{\partial}{\partial W}\left[\frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} L(x_{i,j}, W)\right] = \frac{1}{M}\sum_{i=1}^{M} \underbrace{\frac{1}{N}\sum_{j=1}^{N}\frac{\partial L(x_{i,j}, W)}{\partial W}}_{\text{an ordinary forward–backward on } N \text{ samples}} \tag{12.4}$$

The bracketed inner term is exactly what a single GPU already computes when handed a minibatch of N examples. The outer sum is an average of M such quantities. Linearity lets the summation and the differentiation be reordered at no cost, and the reordering assigns the inner term to a device and the outer term to the network.

The emphasis belongs on *at no cost*. This is not an approximation of single-device training and not a variant of SGD with different convergence behaviour — it is the same computation, rearranged. Given deterministic arithmetic, M GPUs running Equation 12.4 produce bit-for-bit what one impossibly large GPU would have produced on the macro-batch.

The procedure follows directly. Every GPU holds its own complete copy of the weights, the gradients and the optimizer state. Every GPU loads a *different* minibatch — accidentally loading the same data on all devices is a real and easily made bug, and it turns an M-fold increase in cost into no increase in information. Every GPU runs forward on its own data to a local loss, and backward to a local gradient. Then, and only then, communication: an **all-reduce** in which each device simultaneously broadcasts its gradient to every other device and accumulates everyone else's, ending with all M devices holding an identical copy of the averaged gradient. A good implementation does this in time logarithmic in M rather than linear. Each device applies that gradient to its own weights, and because they began the step with identical weights and applied an identical update, they end it with identical weights, and the invariant is restored.

12.5.1. Hiding the all-reduce

Written as five sequential steps, this leaves the interconnect idle during the backward pass and the tensor cores idle during the all-reduce. Neither is acceptable, and the fix is the first instance of the hiding move from Section 12.3.

Backpropagation produces gradients one layer at a time, starting from the last layer. The gradient of the final layer is complete and ready to be communicated at a moment when the device still has the entire rest of the backward pass ahead of it. So launch the all-reduce for layer $\ell + 1$ as the backward pass for layer ℓ begins, and let the two proceed concurrently. In the steady state the network is always reducing the gradient of the layer above the one the tensor cores are currently differentiating, and if the interconnect can keep up, the last all-reduce

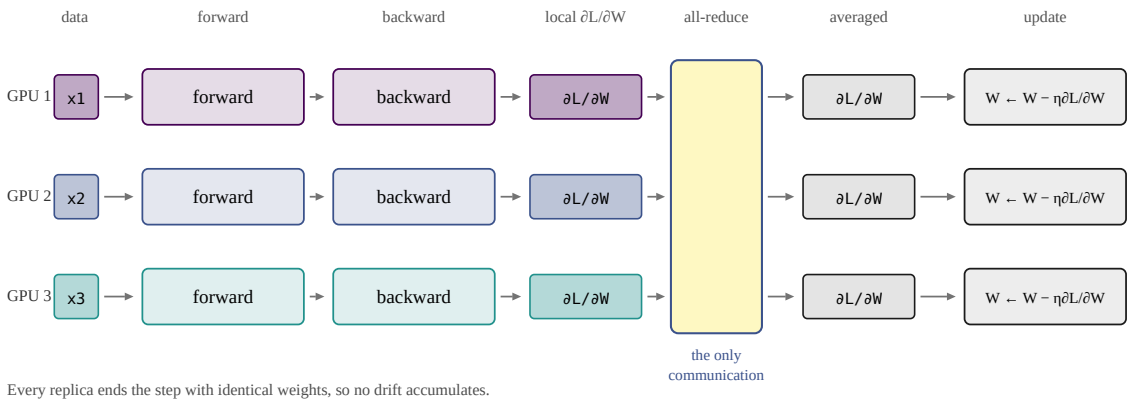


Figure 12.3.: Data parallelism across three devices. The forward and backward passes are entirely local — each device sees only its own minibatch and computes only its own local gradient — and the single point of communication is the all-reduce that averages those gradients. After it, every device holds the same gradient and applies the same update, so the replicas never drift apart. Drawn for this project.

completes at almost the same instant as the last local gradient. The optimizer step then runs with no waiting at all.

Whether the interconnect *can* keep up is a question with no general answer — it depends on the model size, the local batch size, the depth, and the link bandwidth, and the only honest way to find out is to measure the specific configuration. What is general is that none of this scheduling is automatic. Within a single GPU, CUDA overlaps a great deal of memory movement with computation in hardware; across the cluster, nothing does, and the interleaving has to be expressed in software. In PyTorch that software is `DistributedDataParallel`, which registers hooks on the backward pass and issues the collectives at the right moments, so ordinary single-device training code inherits the overlap without being rewritten.

It is worth naming the alternative that this design rejects. One could let each replica take several optimizer steps independently and reconcile the weights occasionally — *asynchronous SGD*, which Google used for some early-2010s models before the TPU pods existed. It communicates far less. It is also less stable, harder to reproduce, and harder to debug, and it tends to reach worse solutions, so the field settled on synchronous updates wherever they are affordable. The tradeoff is not permanent: as clusters grow and interconnects fail to keep pace, asynchronous methods may become attractive again for exactly the reason they were abandoned.

12.5.2. The ceiling

Data parallelism scales the batch beautifully and does nothing whatsoever about the model. Every device still holds a full copy of the weights, gradients and optimizer state, so Equation 12.3 applies unchanged and the largest trainable model is capped at around ten billion parameters on an 80 GB device no matter how many devices there are. Adding GPUs buys throughput and buys no capacity.

That is the wrong tradeoff for the models people actually want, and the fix is to notice that the replication is pure waste: M devices are storing M identical copies of the same optimizer state and using each copy once per step.

12.6. Fully sharded data parallelism

Give every weight matrix an owner. Assign each layer's parameters to exactly one of the M GPUs, and make that GPU responsible for the layer's gradient and optimizer state as well. Memory for model state now falls by a factor of M, because nothing is stored twice. This is **fully sharded data parallelism**, and the sharding is by layer, not by scalar — W_i throughout means an entire layer's weight matrix.

Everything else about data parallelism survives. Each GPU still loads its own minibatch, still runs a full forward and backward pass over the whole network on that minibatch, and still ends up contributing to a gradient averaged over the macro-batch. What changes is that a device no longer possesses the weights it needs, so they have to be fetched.

The forward pass becomes a repeated cycle. Before layer 1 can run, the GPU that owns W_1 **all-gathers** it to every other GPU. All devices now hold W_1, and all run layer 1 on their own local activations. Immediately afterwards, every device that does not own W_1 deletes its copy, returning memory to the sharded state — but keeping the layer-1 activations, which are needed for backward. Then the owner of W_2 broadcasts, and so on to the end of the network. The same hiding move applies: while layer i computes, the weights for layer $i+1$ are already in flight, so in the steady state the gather for the next layer is free.

The backward pass has three things to overlap rather than two, and the extra one is the return trip for gradients. For each layer, in reverse order: the owner gathers the weights out to everyone; every device computes its local gradient for that layer; and every device sends that local gradient back to the owner, which sums them. This last collective is a **reduce-scatter** rather than an all-reduce — the reduction lands on one device instead of on all of them, which is both cheaper and exactly what is wanted, since only the owner will apply the update. Once the owner has the full macro-batch gradient it takes the optimizer step for that layer alone. Notably, it does not then have to broadcast the updated weights, because the next forward pass will gather them anyway.

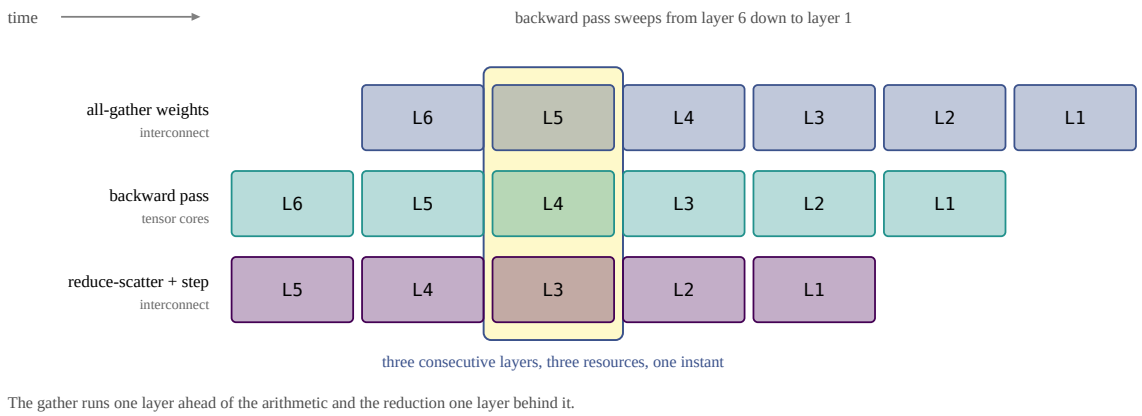


Figure 12.4.: The steady state of an FSDP backward pass. Three things happen concurrently on three different resources: the tensor cores compute the backward pass for layer ℓ, the interconnect gathers the weights for layer $\ell-1$, and the reduce-scatter plus optimizer step for layer $\ell+1$ completes. Communication is only free while all three stay in step. Drawn for this project.

In the steady state of a deep network, then, three consecutive layers are in play at once: layer $\ell+1$ is having its gradients reduced and its weights updated, layer ℓ is being differentiated, and layer $\ell-1$ is having its weights prefetched. Add asynchronous data loading on the host CPUs

and the next batch is ready before the current one finishes. The whole system is a machine for keeping the tensor cores fed.

One optimisation is worth stating because it generalises. At the end of the forward pass every device is holding the last layer's weights, and the very next thing that happens is the backward pass for the last layer, which needs those same weights. So do not delete them. It is a small saving on its own, and it is an instance of the rule that governs all of these schemes: a gather you can skip is worth more than a gather you can hide, because hiding it still consumes bandwidth that something else wanted.

The cost of FSDP is that it communicates far more than plain data parallelism. Over one forward-backward pass, the weights cross the network once during forward and again during backward, and the gradients cross once more — roughly **three times the model size**, against one for plain data parallelism, which moves only the gradients. That ratio is the whole reason the next section exists.

12.7. Hybrid sharding

FSDP has one axis, and a cluster has a hierarchy. Putting a single sharding group across the full cluster means paying the three-model-sizes communication cost over the slowest links in the machine, which is precisely backwards: the most communication-hungry strategy has been assigned the worst wires.

Hybrid sharded data parallelism fixes this by arranging the GPUs in a two-dimensional grid. Along one axis, groups of K GPUs each run ordinary FSDP among themselves, with the model sharded K ways inside the group. Along the other axis, M such groups run ordinary data parallelism against each other — each group holds a complete copy of the model, processes its own share of the macro-batch, and at the end of the backward pass the groups all-reduce their gradients with one another.

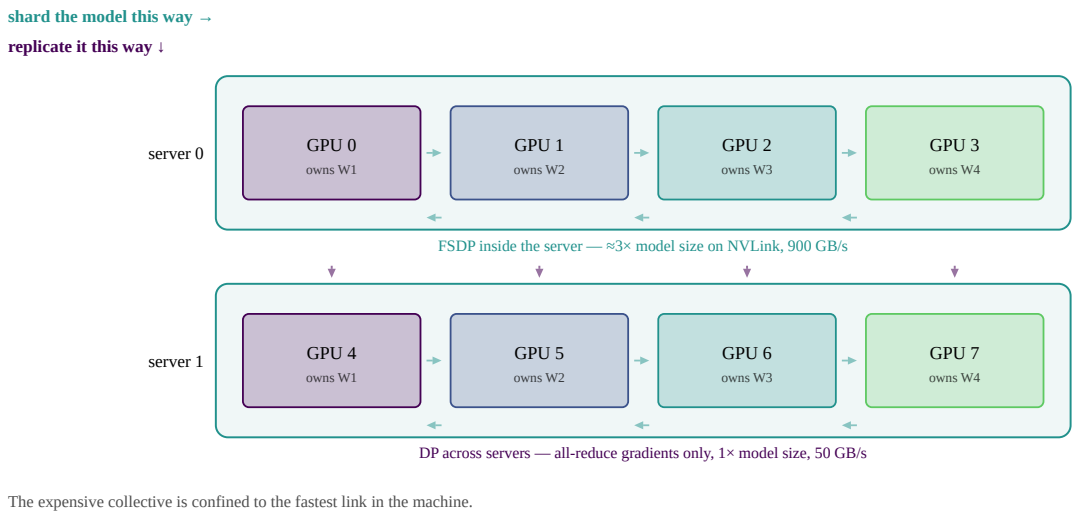


Figure 12.5.: Hybrid sharding as a two-dimensional grid. Within a group the model is sharded and the traffic is roughly three model sizes per step, so groups are made to coincide with the fast domain — the eight GPUs of a server, joined by NVLink. Across groups the model is replicated and only gradients move, one model size per step, which is what the slower inter-server links can afford. Drawn for this project.

The point of the arrangement is that the two axes have different appetites, and the cluster has different links. Sharding costs three model sizes of traffic per step; replication costs one. So make the sharding groups coincide with the fast domain — the eight GPUs inside a server, joined by NVLink at 900 GB/s — and let the replication axis span servers, racks and pods, where bandwidth is an order of magnitude worse but only a third as much has to cross.

This is the first algorithm in the chapter designed against a specific network topology rather than against an abstract collection of devices, and it will not be the last. It is also the second move from Section 12.3 in its purest form: not hiding communication, but placing it.

12.7.1. A recipe, and where it stops

The three schemes so far compose into a practical sequence, and it is worth writing down because most readers will never need anything past it.

Start with plain data parallelism. It works well up to roughly 128 GPUs and models of around a billion parameters, and the local batch size should be set as large as GPU memory allows — that is almost always the right call, since it maximises the arithmetic done per unit of communication. Past a billion parameters the model state stops fitting comfortably and it is time for FSDP, which buys an order of magnitude of model size and scales to several hundred devices. Somewhere between 256 and 512 GPUs, depending on the cluster, FSDP's three-model-sizes cost outgrows the interconnect, and the sharding group should be capped and replicated instead — that is, HSDP. Together these reach models of tens of billions of parameters on about a thousand GPUs.

Beyond that — more than a thousand GPUs, more than fifty billion parameters, or sequences longer than ten thousand tokens — the batch axis is exhausted and the remaining three axes of Figure 12.2 have to be brought in. Before they are, there is a second memory problem that none of the above touches, and a metric that decides all the tradeoffs.

12.8. Activations, and paying compute to store fewer of them

Sharding solves the model-state problem completely. A hundred-billion-parameter model needs 800 GB by Equation 12.3, and split across eighty GPUs that is 10 GB each — comfortable. The memory then fills up anyway, because of the thing Equation 12.3 does not count.

Backpropagation requires the activations. Every layer's backward pass needs the input it saw during the forward pass in order to compute the gradient, so the forward pass cannot discard as it goes; it accumulates. And unlike the model state, activation memory scales with the batch and the sequence length as well as with the depth.

The arithmetic for Llama 3 405B is bracing. It has 126 layers and a model dimension of 16,384, and the first pre-training stage used sequences of 8,192 tokens. Storing one tensor per layer for a *single* sequence at 16-bit precision costs

$$126 \times 8192 \times 16384 \times 2 \text{ bytes} \approx 33.8 \text{ GB} \tag{12.5}$$

That is 42% of an 80 GB H100 for one sequence, and it is an underestimate by a wide margin, because a transformer block stores considerably more than one tensor — the attention projections, the softmax output, the two MLP intermediates, and the normalisation inputs are all live. There is no batch size that makes this work.

The escape is that activations are not data; they are *recomputable*. Every one of them is a deterministic function of the input, so any activation that has been thrown away can be reconstructed by running the forward pass again. This trades compute for memory, and compute is the thing the tensor cores have in abundance.

Model a network as N layers, each of which is two functions: a forward $A_{i+1} = F_i^{\rightarrow}(A_i)$ and a backward $G_i = F_i^{\leftarrow}(A_i, G_{i+1})$, each costing one unit of compute and one unit of memory. Ordinary training runs N forward steps, holding on to all N activations, then N backward steps: $O(N)$ compute and $O(N)$ memory.

At the other extreme, keep nothing but the input. The forward pass runs and discards as it goes. The backward pass for the last layer is fine, but the backward pass for layer i needs A_i, which is gone, so it is regenerated by rerunning layers 1 through $i-1$. Summing that over all i gives $N + \sum_{i=1}^{N} i = O(N^2)$ compute for $O(1)$ memory. The memory problem is solved and replaced by a compute problem that is worse, since N is in the hundreds.

The useful regime is between them. Keep C checkpoints, spaced N/C layers apart, and recompute from the nearest one whenever a backward step needs an activation it does not have. Each such recomputation walks at most a segment, so

$$\text{compute} = O\left(\frac{N^2}{C}\right), \qquad \text{memory} = O(C) \tag{12.6}$$

There is no interior optimum here — Equation 12.6 is a pure exchange rate, and C should be pushed as high as the memory budget allows. The conventional balanced point is $C = \sqrt{N}$, giving $O(N\sqrt{N})$ compute for $O(\sqrt{N})$ memory. On a 126-layer network that means holding eleven checkpoints instead of 126, at a little under four times the compute of an unchecked step.

Real implementations do better than Equation 12.6 by recomputing each segment *once* rather than once per layer within it. Hold the segment's activations while backpropagating through it and discard them when the segment is done, and the forward pass is repeated exactly once over the whole network: compute $3N$ against the baseline $2N$, a flat overhead of half a forward pass, for a peak memory of $N/s + s$ where s is the segment length. That is minimised at $s = \sqrt{N}$, so the same $O(\sqrt{N})$ memory costs a constant factor rather than a $\sqrt{N}$ one. This is Chen et al., 2016, and it is what `torch.utils.checkpoint` does.

Activation checkpointing is genuinely a cost, not a free lunch — it makes every step slower, and the honest reason to turn it on is that the alternative is not training the model at all. It is the last of the memory techniques, and once it is enabled there are no more knobs of that kind. The remaining question is how to set the many knobs that now exist.

12.9. The metric that settles the arguments

Global batch size, local batch size, sharding-group size, replication factor, checkpoint interval, and shortly three more axes of parallelism: the configuration space is large, the interactions are not intuitive, and no amount of reasoning from first principles resolves it. What resolves it is a single measured number.

Build up to it in two steps. The first is **hardware FLOPs utilisation**, which asks what fraction of a device's theoretical peak an operation actually achieves. An H100 is rated at 989.4 TFLOP/s of 16-bit matrix multiply on its tensor cores; a plain PyTorch loop multiplying dense square matrices reaches roughly 80% of that once the matrices are around 8192×8192, and

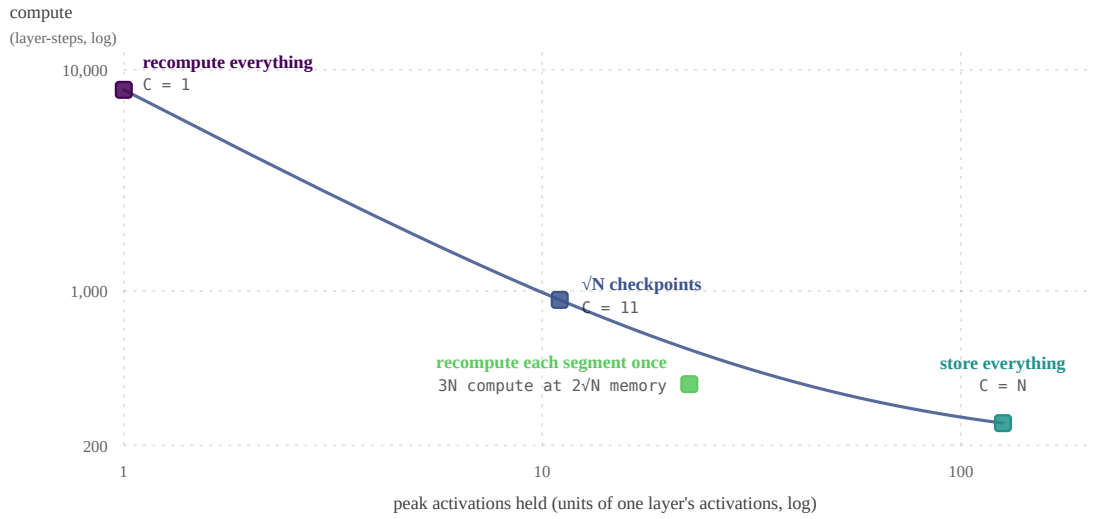


Figure 12.6.: Compute against peak activation memory for a 126-layer network, both on log axes. The curve is Equation 12.6 evaluated for every checkpoint count from one to N: storing everything sits at the cheap end of the compute axis, recomputing everything from the input at the cheap end of the memory axis, and $C = \sqrt{N}$ between them. The isolated point below the curve is the segment-recompute scheme, which reaches the same memory for a constant factor instead. Computed for this project.

considerably less when they are small. HFU is a property of the kernel and the shapes, and it is measured with a few lines of code.

HFU is also not the number that matters, because a training loop does a great deal that is not the matrix multiply: loading data, augmenting it, normalising, applying nonlinearities, communicating gradients, and — after Section 12.8 — recomputing activations it chose to throw away. **Model FLOPs utilisation** asks the sharper question: what fraction of the device's theoretical peak is going into useful model computation?

It is computed as a ratio of times rather than of operations, which is what makes it robust. Count the matrix-multiply FLOPs in one forward and backward pass of your architecture on your batch, approximating the backward as twice the forward and ignoring nonlinearities, normalisation and elementwise operations, since those run on the FP32 cores and are not what the peak figure describes. Divide by the device's rated throughput to get $t_{\text{theoretical}}$, the time the step would take on a perfect machine. Then measure t_{actual}, the wall-clock time of a complete iteration including data loading, forward, backward and the optimizer step. Then

$$\text{MFU} = \frac{t_{\text{theoretical}}}{t_{\text{actual}}} \tag{12.7}$$

Everything the training loop does other than the model's own arithmetic lands in the denominator and nowhere else. Communication that failed to hide behind compute shows up here. So does a pipeline bubble, a straggler, a data loader that cannot keep up, and recomputation from activation checkpointing — which is the sharp consequence of Equation 12.7 worth pausing on. Recomputed FLOPs are real work the tensor cores perform, so they raise HFU; but they are not in the numerator of Equation 12.7, which counts one forward and one backward. Checkpointing

therefore appears as a *drop* in MFU, correctly, because the extra work buys memory rather than progress.

Calibration: above 30% is good and above 40% is excellent. Well below 30% means something is badly wrong rather than slightly suboptimal, and is worth diagnosing before tuning anything.

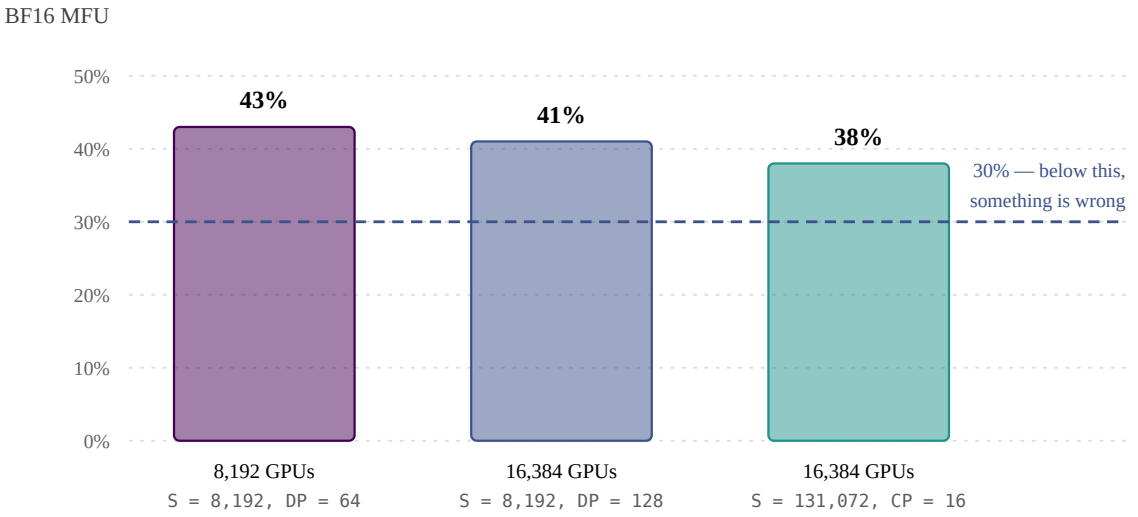


Figure 12.7.: BF16 model FLOPs utilisation for the three stages of Llama 3 405B pre-training, replotted from Table 4 of the Llama 3 paper. The drop from 43% to 41% comes from halving the per-group batch size to hold the global token count fixed while doubling the device count; the further drop to 38% is the cost of the long-context stage, which raises the sequence length sixteenfold and turns on context parallelism. Replotted for this project from published numbers.

Llama 3 405B reached 43% on 8,192 GPUs, 41% on 16,384, and 38% in the final long-context stage. Those are state-of-the-art figures, and the shape of the decline is instructive: the 43% and the 41% differ only in that the second doubles the device count while holding the global tokens per batch constant, which halves the batch each data-parallel group sees and gives the collectives less arithmetic to hide behind.

One counterintuitive trend closes the section. Newer hardware sometimes achieves *worse* MFU than its predecessor. The A100 to H100 transition multiplied peak tensor-core throughput by about 3.2× — 312 to 989 TFLOP/s — while memory bandwidth roughly doubled. Compute is outrunning the ability to feed it, on the chip and between chips alike, so the fraction of a device's peak that a real workload can reach has been drifting downward even as the absolute peak climbs. Every technique in the remainder of this chapter is aimed at that gap, and the gap is widening.

12.10. Tensor parallelism

Split the feature dimension, which means splitting a single weight matrix across devices. This is a different thing from FSDP, where a whole matrix has an owner and is temporarily copied elsewhere; here no device ever holds a complete W, and the matrix multiply itself is distributed.

A linear layer computes $XW = Y$ with X of shape $N \times D$ and W of shape $D \times D$. Cut W into P column blocks $W_1, \dots, W_P$, each $D \times D/P$, and give one to each device. Every device

already has the full X, so device i computes $XW_i = Y_i$, a column slice of the output, with no communication at all:

$$X \begin{bmatrix} W_1 & W_2 & \cdots & W_P \end{bmatrix} = \begin{bmatrix} Y_1 & Y_2 & \cdots & Y_P \end{bmatrix} \tag{12.8}$$

The forward pass of one layer is free. The trouble is the next layer, which appears to need all of Y, and gathering it would cost a full activation tensor per layer.

It does not need all of Y, provided the next layer is cut the other way. Let the second layer compute $YU = Z$ and split U into P *row* blocks $U_1, \ldots, U_P$, each $D/P \times D$, with U_i living on the same device as W_i. Block multiplication then gives

$$Z = YU = \sum_{i=1}^{P} Y_i U_i \tag{12.9}$$

and device i already holds both factors of the i-th term. Each device computes a complete $N \times D$ partial sum from purely local data, and one all-reduce over the P devices produces Z. Two layers, one collective.

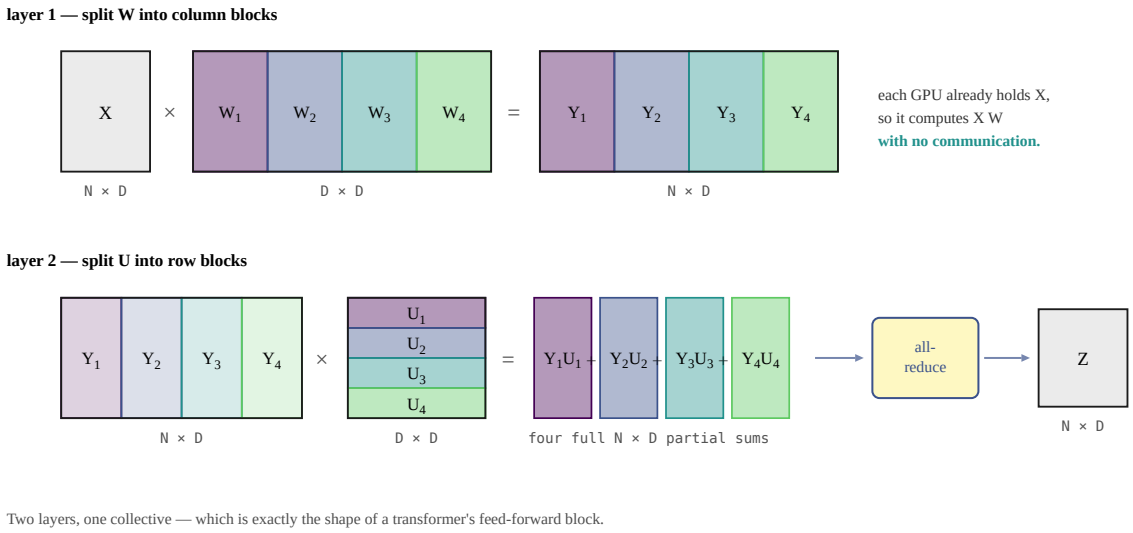


Figure 12.8.: Column-parallel followed by row-parallel. The first weight matrix is cut into column blocks, so each device produces a column slice of Y and nothing needs to be exchanged. The second is cut into row blocks placed on the matching devices, so each computes one full-sized term of Equation 12.9 from data it already has, and a single all-reduce over the partial sums yields Z. Drawn for this project.

The pairing is not a coincidence looking for an application. A transformer's feed-forward block is exactly two linear layers in sequence, so column-parallel then row-parallel drops onto it without modification, and tensor parallelism inside the MLP is standard practice in large transformers.

What tensor parallelism costs is bandwidth, and a lot of it: an all-reduce of a full activation tensor for every pair of layers, at every step, with the tensor cores stalled until it lands — there is no next layer to overlap it with. That places tensor parallelism firmly inside the fastest available domain. In practice the tensor-parallel degree is set to the number of GPUs in one server, so that every one of these collectives stays on NVLink and never touches the network.

12.11. Context parallelism

Split the sequence, so that several devices cooperate on a single long input. This is the axis that matters when the sequence length rather than the model size is what exhausts memory, which is the situation in long-context training and fine-tuning.

Most of a transformer takes to it easily. Normalisation, the feed-forward block and the residual connections all act on each token independently, so cutting the S axis and handing each device a contiguous span of tokens changes nothing about their computation. The feed-forward block has weights, so its gradients need the same all-reduce that data parallelism required, but the forward pass is embarrassingly parallel.

Attention is not, and it is the entire difficulty. The queries, keys and values are per-token projections and split cleanly, but attention proper computes an interaction between every pair of positions, and a device holding tokens 1 through S/P needs keys and values from every other span to compute even one row of it.

Two answers exist. Ring attention keeps the sequence split and moves the data: the attention matrix is decomposed into blocks, and key–value blocks are passed around a ring of devices so that each device eventually sees every block it needs, with the transfers overlapped against the block computations. It is the more general solution and the more intricate one.

Ulysses attention is conceptually simpler and exploits a structure the architecture already has. Multi-head attention computes H independent attention operations, so instead of splitting attention along the sequence, split it along the heads. An all-to-all before the attention block converts the sequence-sharded layout into a head-sharded one, each device computes a few complete heads over the full sequence, and a second all-to-all converts back. Everything outside attention stays split by sequence; only the attention operator changes its notion of what a device owns.

Llama 3 shows the regime where this becomes necessary. The first pre-training stage ran at a sequence length of 8,192 with no context parallelism at all. The final stage raised the sequence length to 131,072 and used 16-way context parallelism, putting sixteen GPUs on a single sequence — with 8,192 tokens each, which is exactly the span one GPU handled alone in stage one. The batch dimension has effectively gone below one example per device.

12.12. Pipeline parallelism

Split by depth: give the first quarter of the layers to GPU 1, the second quarter to GPU 2, and so on, passing activations across device boundaries in the forward pass and gradients back in the reverse. It is the most obvious partition of a deep network and, done naively, the worst.

The problem is the sequential dependency. GPU 2 cannot start until GPU 1's output arrives; GPU 3 waits on GPU 2. The forward pass sweeps down the devices one at a time, GPU P turns around and does its backward, and the gradients sweep back up, again one device at a time. At every instant exactly one device is working. With P stages the utilisation ceiling is $1/P$, so eight-way pipeline parallelism caps MFU at 12.5% — worse than any bottleneck the previous sections were trying to fix. The idle regions have a name: the **bubble**.

The bubble exists because there is only one batch in flight, and the fix is to put several in flight. Split the batch into m microbatches and stream them through the pipeline: GPU 1 runs forward on microbatch 1 and hands it to GPU 2, then immediately runs forward on microbatch 2 while

GPU 2 works on microbatch 1, and so on. This is GPipe, and after a fill period of $P-1$ slots the pipeline is saturated.

Counting the schedule gives the utilisation directly. The forward sweep occupies $m+P-1$ time slots and the backward sweep the same, for $2(m+P-1)$ slots on P devices, while the useful work is $2mP$ device-slots. So

$$\text{utilisation} = \frac{2mP}{2P(m+P-1)} = \frac{m}{m+P-1}, \qquad \text{bubble fraction} = \frac{P-1}{m+P-1} \tag{12.10}$$

With $m=1$ this recovers the $1/P$ ceiling. With $P=4$ and $m=4$ it gives $4/7 \approx 57.1\%$, and the ratio keeps improving as m grows.

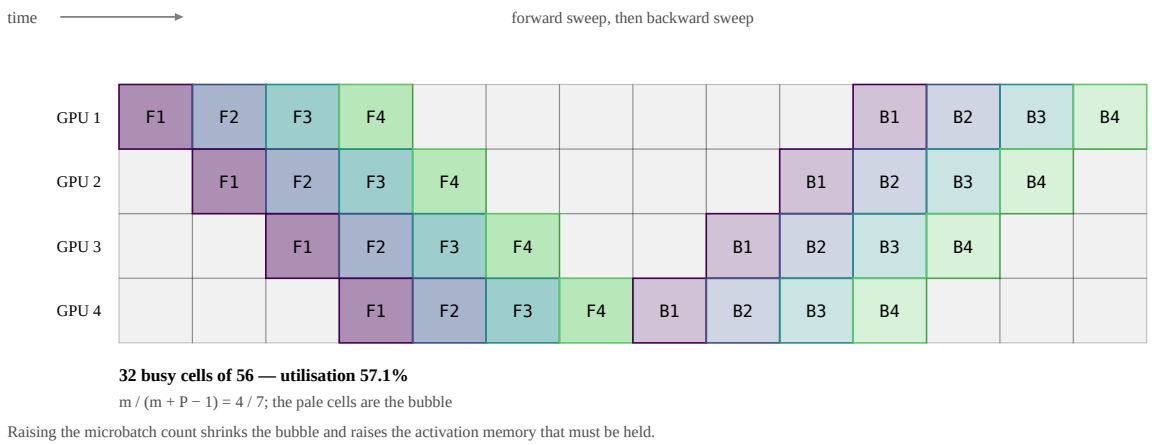


Figure 12.9.: A four-stage pipeline with four microbatches. Each colour is a microbatch; the shaded cells are the bubble. The fill and drain regions are what Equation 12.10 counts, and they shrink relative to the useful work as more microbatches are put in flight — which is why the only way to make pipeline parallelism efficient is to have many more microbatches than stages. Computed for this project.

So take m large and the bubble vanishes. Except that each in-flight microbatch holds its own activations until its backward pass completes, so raising m raises activation memory in proportion — and the response to that is activation checkpointing, which costs compute, which costs MFU. More pipeline stages, fewer microbatches, more aggressive recomputation, more data parallelism layered on top: the knobs are coupled and there is no closed form for the optimum. This is what Section 12.9 is for. Tune whichever knob is available and keep the setting that maximises MFU.

12.13. Four dimensions at once

Which of the five strategies is best is the wrong question. The answer in practice is all of them, applied simultaneously along different axes, with the degree of each chosen to fit the cluster.

Arrange the GPUs conceptually in a grid with one dimension per parallelism strategy. A device's coordinates give its rank along each axis: which shard of the feature dimension it holds, which span of the sequence, which block of layers, and which slice of the batch. HSDP in Section 12.7 was already a two-dimensional instance of this. Current practice runs four.

Llama 3 405B's largest configuration is the concrete case. On 16,384 GPUs it used eight-way tensor parallelism, sixteen-way context parallelism, sixteen-way pipeline parallelism and eight-way data parallelism, and $8 \times 16 \times 16 \times 8 = 16{,}384$ accounts for every device exactly once.

The assignment of degrees to axes is where the topology of Section 12.3 is finally cashed in. Tensor parallelism is the most communication-hungry — an unhidden all-reduce of an activation tensor every two layers — and its degree is eight, which is precisely the number of GPUs in one server sharing an NVLink domain. Pipeline parallelism sends only a boundary activation tensor between adjacent stages and tolerates slower links, so it can span racks. Data parallelism communicates once per step and can span pods. The physical hierarchy that Figure 12.1 draws and the logical hierarchy of the parallelism grid are the same hierarchy, and the job of the configuration is to make them line up.

That, finally, is the discipline. There is no single trick and no schedule that is right in general. There is a machine with a known bandwidth hierarchy, a computation with four separable axes, one memory-for-compute trade, and one number that tells you whether the arrangement you chose is working.

12.14. Closing

A GPU is a parallel processor whose speed comes from a narrow specialisation — units that do nothing but small matrix multiply-accumulate at reduced precision — and whose performance is governed by a memory hierarchy in which everything larger is further and slower. A cluster is the same object at a larger scale, extending that hierarchy four levels past the edge of the chip, and it is best understood as one computer rather than as many.

Training a large model on it means splitting the computation along the four axes a transformer offers. The batch splits into data parallelism, and sharding the model state across the group turns that into FSDP, which trades three times the communication for an M-fold reduction in memory; restricting the sharding to the fast domain and replicating above it gives HSDP. The sequence splits into context parallelism, where everything but attention is trivial and attention needs ring or Ulysses. The feature dimension splits into tensor parallelism, where a column-parallel layer feeding a row-parallel layer needs one collective per pair and therefore belongs inside a server. The depth splits into pipeline parallelism, which is worthless without microbatches and good with them. Activation checkpointing sits underneath all of it, buying memory with recomputation at the $\sqrt{N}$ point where the two costs balance.

None of these choices has a principled optimum, and all of them are settled the same way — by measuring model FLOPs utilisation and keeping what raises it. The frontier sits between 38% and 43%, and it is not obviously going up, because compute is growing faster than the bandwidth that has to feed it.

13. Self-Supervised Learning

Lecture 12

Based on Lecture 12 of CS231n, Stanford University, Spring 2025.

13.1. What a representation costs

There is a result from earlier in the course that is easy to pass over and worth returning to. Take a network trained to classify ImageNet, cut it open at the last hidden layer, and read off the 4096-dimensional vector it produces for an image. Look for that image's nearest neighbours in that vector space under an ordinary L^2 distance, and what comes back are pictures of the same kind of thing — the same animal in a different pose, against a different background, at a different scale. Do the same search in pixel space and what comes back are pictures with the same average colour.

The network was never asked to produce that. It was asked to name the object, and the vector is an artefact of the layer before the naming happens. Yet it is far more useful than the name: a linear classifier trained on those vectors will pick out categories the network was never shown, which is the whole content of the transfer learning discussed in Section 7.12. The representation is the valuable object, and the classifier on top of it is a thin thing that can be replaced.

Which raises the question of what the representation cost. It cost 1.2 million images, each of which a human being looked at and assigned to one of a thousand categories. That is the bill, and it grows steeply with how much detail the task demands. The COCO annotation effort is documented precisely enough to make the shape of the growth visible: deciding which of eighty categories appear in an image consumed about 20,000 aggregate worker hours across 328,000 images; marking where each instance sits added another 10,000; and drawing the actual outlines — 2.5 million of them, at over 22 worker hours per thousand — added roughly 55,000 more. The whole dataset took more than 70,000 hours of human attention, which is about forty years of one person working full time, and the segmentation stage alone is three quarters of it.

Push the density further and it gets worse rather than better. Cityscapes, which labels every pixel of a street scene with one of thirty classes, reports that annotation and quality control together required more than one and a half hours for a single image. The dataset's coarse tier — polygons drawn quickly, deliberately not following boundaries — was capped at under seven minutes per image, and the fact that the authors built a second, worse tier at all is an admission that the good one does not scale.

Meanwhile there are, on the internet, some very large number of images that nobody has labelled and nobody is going to. The asymmetry is total: labelled data is a scarce, expensive, slowly-growing resource, and unlabelled data is effectively free and effectively unbounded. If the representation is the thing worth having, and the label is only the pretext under which it was

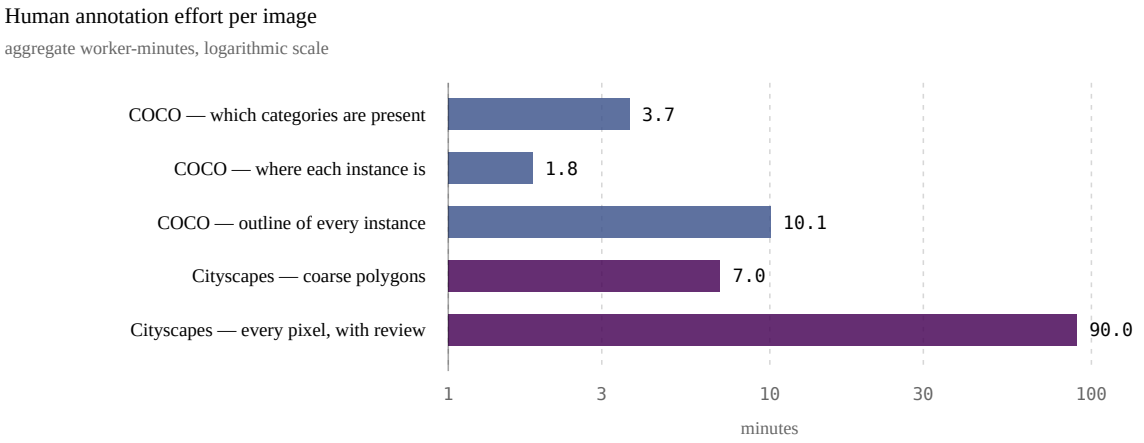


Figure 13.1.: Aggregate human annotation effort per image, from published dataset papers. The first three bars are the three stages of the COCO pipeline and the last two are the two annotation qualities of Cityscapes; both papers report their own figures, and the axis is logarithmic because otherwise only the last bar is visible. The span from "which categories are present" to "every pixel has a class" is a factor of about twenty-five, and it is the reason dense prediction datasets are small.

obtained, then the question is whether some other pretext would do — one whose answers the data already contains.

13.2. An invented task, and a real one

The answer that organises this chapter is to invent a task. Take unlabelled images, corrupt or transform them in a way whose parameter you chose and therefore know, and train a network to recover that parameter. The supervision is genuine — there is a loss, a target, and a gradient, and the training is supervised learning in every mechanical respect — but no human produced the target. It was manufactured by the same code that produced the input. This is the **pretext task**, and the task you actually care about, trained afterwards on whatever small labelled dataset you have, is the **downstream task**.

The structure has two stages and one moving part between them. In the first stage a network is split into an encoder, which maps the image to a representation, and a head — a decoder, a classifier, a regressor, whatever the pretext task requires — which maps that representation to the manufactured target. Both are trained together on the unlabelled corpus. In the second stage the head is thrown away. The encoder is kept, a new and usually very shallow head is attached, and that head is trained on the labelled downstream data.

Discarding the head is not a detail. It is what makes the choice of pretext task a design problem rather than an application: nothing about solving the pretext task is wanted for its own sake, and the entire value of the exercise is in what solving it forces the encoder to encode. A pretext task can be perfectly solvable and completely useless, if there is a cheap surface statistic that solves it. Much of the history in Section 13.3 is the history of discovering such shortcuts and closing them.

That criterion also explains why the two requirements on a pretext task are stated together and are in tension. It must generate its own labels, which pushes towards simple mechanical corruptions; and solving it must require understanding the content of the image, which pushes

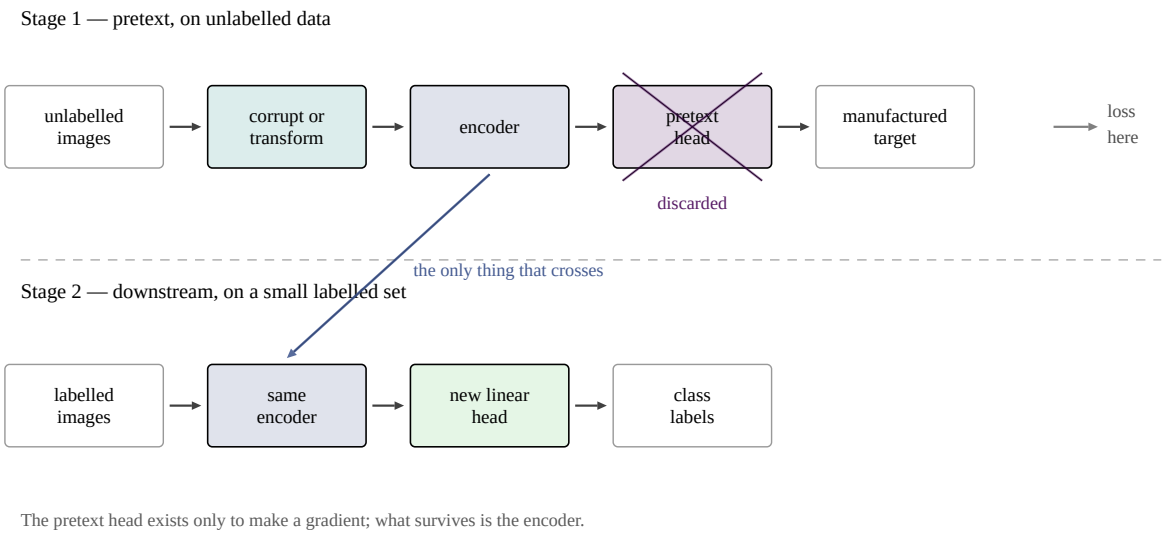


Figure 13.2.: The two stages. Everything above the dashed line is trained on unlabelled data against a target the pipeline generated for itself; everything below is trained on a small labelled dataset. The encoder is the only thing that crosses. The pretext head exists solely to create a gradient and is discarded the moment it has done so, which is why its architecture matters much less than the loss it imposes on the encoder.

towards corruptions no surface statistic can undo. The good pretext tasks sit where those two pressures meet.

Because the pretext task's own accuracy is not the point, evaluating a self-supervised method takes some care, and three protocols recur throughout the rest of this chapter. **Linear probing** freezes the encoder entirely and fits a single linear layer on top using the full labelled training set. It asks a sharp question: are the classes already linearly separable in the representation, without any adaptation at all? **Fine-tuning** unfreezes everything and trains end to end from the pretext initialisation, which measures the representation's value as a starting point rather than as a finished product. **Semi-supervised evaluation** fine-tunes on a deliberately small fraction of the labels — one percent, ten percent — which is the regime the whole enterprise is aimed at, since a method that only helps when you already have a million labels has not solved the problem it was built for.

These three do not agree with each other, and the places where they disagree are informative rather than noisy. A representation can be excellent under fine-tuning and mediocre under a linear probe, which says its information is present but not linearly accessible; Section 13.6 contains the sharpest instance of that gap. It is worth holding all three in mind, because papers tend to report the protocol that flatters them.

One last piece of context before the tasks themselves. Everything here is described for images, because this is a vision course, but self-supervised learning is not a vision technique that happens to work elsewhere. Predicting the next token in a corpus of text is a pretext task in exactly this sense — the label is the next word, which the data already contains — and it is the entire pre-training objective behind every large language model. The same pattern underlies modern speech models and much of what is now done in robotics, where the appeal is identical: a fleet collecting sensor data does not have to annotate it to learn from it. Vision arrived at the idea late and, for a while, awkwardly.

13.3. Tasks made from transformations

The first family of pretext tasks all have the same shape. Sample a transformation from some set, apply it to the image, and ask the network which one you applied. The transformation's identity is the label, and it is free because you drew it yourself.

The cleanest instance is rotation. Rotate each image by one of 0°, 90°, 180° or 270° and train a four-way classifier on the result — this is RotNet, Gidaris et al., 2018. The restriction to multiples of a right angle is deliberate and does more work than it looks: those four rotations can be implemented as transposes and flips of the pixel array, so they introduce no interpolation artefacts and no border padding, and a network cannot solve the task by detecting resampling blur. It is a genuine four-way classification over image content.

The reason it can work at all is a hypothesis about what recognising an upright object requires. To know that a photograph of a dog has been turned on its side, you must know what an upright dog looks like — where the head sits relative to the legs, which way the ground is. There is no local statistic of a natural image that reveals its orientation; the information is in the arrangement of the object's parts, and a network that solves the task has had to represent that arrangement.

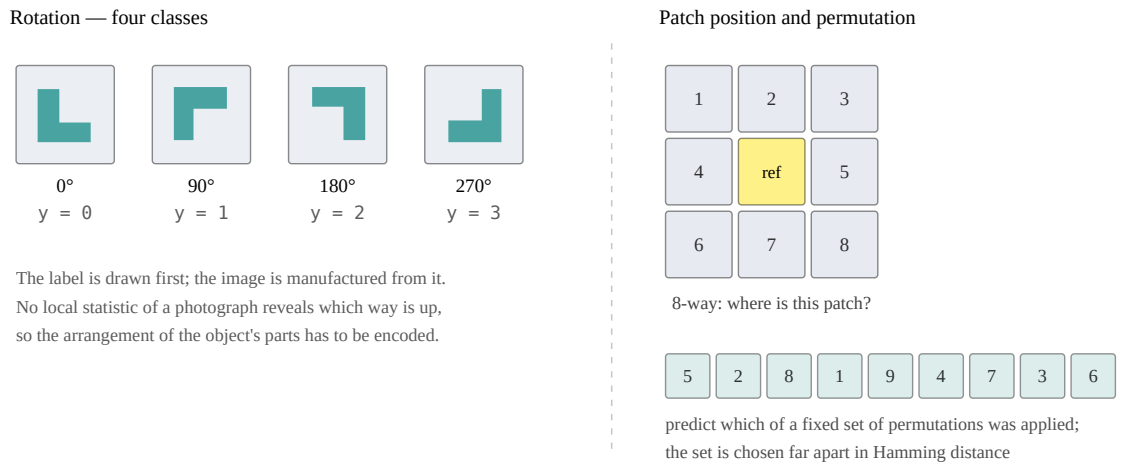


Figure 13.3.: The shared structure of the transformation tasks. On the left, one image and the four rotations it is turned into, each carrying the index of the rotation as its own label. On the right, the 3×3 patch grid: the relative-location task asks for one patch's position among the eight around the centre, and the jigsaw task asks which of a fixed set of permutations was applied to all nine at once. In each case the label is drawn first and the image is manufactured from it, so the arrow between them runs the opposite way to ordinary supervision.

The evidence that it does is the transfer, and RotNet's numbers are worth reading against the two baselines that bracket them. Pre-train an AlexNet on ImageNet *with* its labels, freeze it at conv4, and a classifier on those features reaches 59.7% top-1 on ImageNet; leave the same layers randomly initialised and frozen, and it reaches 27.1%. RotNet, which sees the same images and none of the labels, reaches 50.0%. The gap to the supervised ceiling is real, but the gap to random initialisation is four times larger, which is the finding. On PASCAL VOC 2007 detection the rotation-pretrained model reached 54.4% mAP, the best of any label-free method at the time.

A second finding from the same paper is more suggestive than the accuracies and easier to overread. Visualising which regions each model's activations respond to, the supervised classifier's attention is narrow — it has found the parts that discriminate the thousand ImageNet categories and ignores everything else, which is exactly what its loss asked of it. The rotation model's attention is spread over more of the object. That is not evidence of a better representation, since the model has no idea what the downstream task will be and has hedged accordingly; but a representation that hedges is the one you want when the downstream task is unknown, and this is the first appearance of a tension that runs through the chapter. A specific objective produces specific features. Self-supervision is trying to buy generality, and generality looks like this.

The second instance replaces rotation with spatial arrangement. Cut a 3×3 grid of patches from the image, hand the network the centre patch and one neighbour, and ask which of the eight surrounding positions the neighbour came from (Doersch et al., 2015). Again an eight-way classification, again free, again apparently requiring some idea of what the object is: to know that this patch of fur belongs above that patch of fur, you need to know it is a face.

Except that it does not require that, and the story of how it fails is the best illustration in this chapter of why pretext task design is hard. The first shortcut is obvious enough to be anticipated — patches that were adjacent in the original image have edges and textures that continue across the seam, so the network can match boundary patterns without looking at content. The fix is to leave gaps between the patches and jitter each patch's position by a few pixels, and Doersch et al. did both. The second shortcut was not anticipated by anyone. Camera lenses focus different wavelengths at slightly different distances, so in many cameras the green channel is very slightly shrunk toward the image centre relative to red and blue. This chromatic aberration varies smoothly with distance from the optical axis, which means that measuring the local separation between green and magenta tells a network *where on the lens* a patch was taken from. Knowing each patch's absolute position, the relative position is arithmetic. The network had learned to read the camera, not the scene.

The lesson generalises past this one bug. A pretext task defines a set of images the network must distinguish, and the network will find the cheapest feature that distinguishes them — not the feature you had in mind. Every advance in this family is, in part, the closing of a shortcut, and the eventual fix for chromatic aberration was to project colours away from the green–magenta axis or to drop two of the three channels outright and replace them with noise, both of which are ways of destroying information so the network cannot cheat with it.

Noroozi and Favaro's jigsaw puzzles push the same idea to all nine patches at once: shuffle them and predict the permutation. All nine tiles have $9! = 362{,}880$ arrangements, which is too many classes to be usable and, worse, mostly indistinguishable — permutations differing in a single swap of two tiles pose an almost impossible discrimination. So the paper fixes a much smaller set of permutations in advance and predicts an index into it, which turns the problem back into a classification over a manageable number of classes. The set is not chosen at random: it is selected to maximise the average Hamming distance between its members, so that every permutation displaces most tiles and the network is never asked to split hairs. Chromatic aberration is handled here too, by jittering the colour channels. Solving the puzzle transfers about as well as relative patch prediction — 45.3% at conv4 against Doersch's 45.6% — and the harder task did not buy much.

That flatness across the family is the honest summary. Rotation, relative location and jigsaw are three different inventions with three different shortcut problems, and they land within a few points of each other and well short of supervised pre-training. Each one buys features shaped by the peculiarities of its own task, and there is no principle telling you what to invent next.

13.4. Tasks made from reconstruction

The second family removes information from the image and asks the network to put it back. The label is the part you removed, so it is free in exactly the same sense; what differs is that the target is now a chunk of image rather than a class index, and that difference turns out to cause most of the trouble.

Cut a hole in the image and ask the network to fill it — context encoders, Pathak et al., 2016. The architecture is an encoder mapping the corrupted image to a representation and a decoder painting the missing region back, with the loss computed only where the hole was. Writing M for the binary mask that is 1 inside the hole and 0 outside, and F for the network,

$$L_{\text{rec}}(x) = \left\| M \odot \left(x - F((1 - M) \odot x) \right) \right\|_2^2 \tag{13.1}$$

where $\odot$ is elementwise multiplication. The network sees $(1 - M) \odot x$, the image with its hole blacked out, and is scored only on $M\odot$ its output, the pixels it invented. To fill a hole convincingly the network has to know what object the surrounding context belongs to and what that object looks like where it cannot see, which is a strong demand and is why the task is interesting.

It is also where reconstruction losses first show their flaw. A hole in a photograph has many plausible fillings, and Equation 13.1 rewards the network for predicting their average. The average of many plausible fillings is a blur. Context encoders' reconstructions are recognisably the right *kind* of content in the right place, and they are soft and smeared, and no amount of training fixes this because the smear is what the loss asked for. Pathak et al.'s remedy was to add an adversarial term — a second network trained to tell real images from filled-in ones, whose disapproval the first network must minimise — which converts "be close to the truth on average" into "be indistinguishable from something real". The mechanics belong to Section 13.13's successor lecture on generative models; what matters here is why it was needed.

Colourisation is the same idea applied to channels rather than regions, and it makes the ambiguity problem impossible to ignore. Convert the image to the Lab colour space, which factors a pixel into a lightness L and two chrominance coordinates a and b; hand the network L and ask for a and b. Every colour photograph in existence is a training pair, and the task genuinely needs semantics, since nothing in the grayscale intensities says a banana is yellow.

But an object's colour is not determined by its shape. An apple may be red or green; a car may be any colour at all. Regressing a and b under a squared loss therefore drives the network to the *mean* of the plausible colours, and the mean of red and green is a muddy grey — which is precisely what the early colourisation networks produced. Zhang et al., 2016 diagnosed this and changed the output space rather than the architecture: quantise the ab plane into a grid, keep the 313 cells that occur in natural images, and predict a *distribution* over those 313 classes with a cross-entropy loss. A distribution can be bimodal where a regression cannot, so the network is permitted to say "red or green, not grey". A further reweighting compensates for the fact that most pixels in most photographs are near-neutral and would otherwise dominate the loss. The result is vivid colourisations, and the general lesson is worth extracting: when the target is genuinely ambiguous, the fix is usually to give the model a loss that can express the ambiguity, not to train it harder.

Once the task is stated as "given some channels, predict the others", the restriction to $L \to ab$ looks arbitrary. Why not the reverse? The split-brain autoencoder does both at once: split the input tensor into two disjoint sets of channels X_1 and X_2, train one network F_1 to predict X_2

from X_1 and another network F_2 to predict X_1 from X_2, and use the concatenation of their representations as the feature.

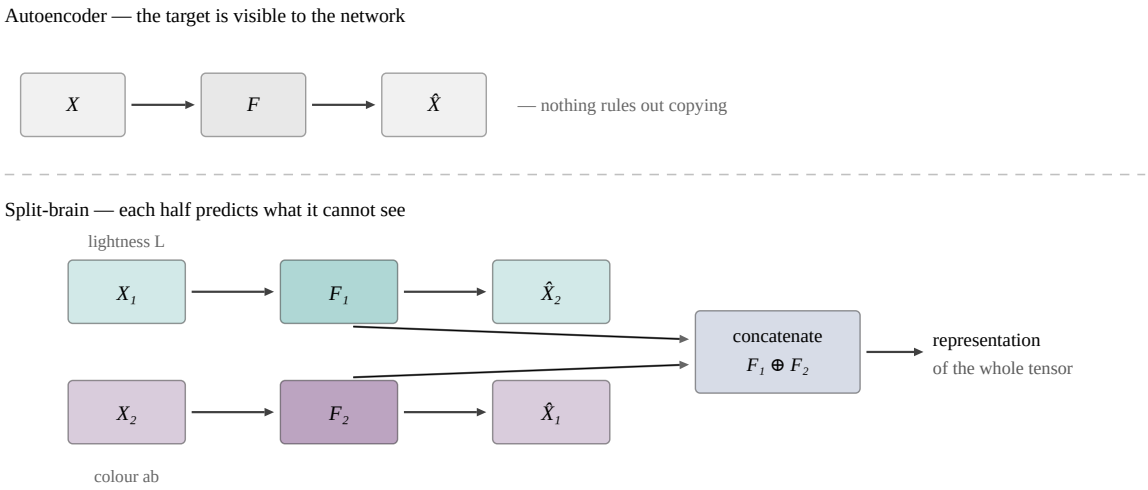


Figure 13.4.: The split-brain construction. A conventional autoencoder passes the whole input through a bottleneck and reconstructs it, which admits the degenerate solution of copying; splitting the channels makes each half a genuine prediction problem, because neither network can see what it is asked to produce. The two representations are concatenated, so the feature is defined on the whole tensor even though each encoder only ever saw part of it.

The split is what makes this more than a rebranded autoencoder. A plain autoencoder sees its own target and can, given enough capacity, learn to copy; every trick in the autoencoder literature is some way of preventing that. Withholding the channels being predicted removes the possibility structurally rather than by penalty. And because the construction refers only to "channels", it applies wherever the input has more than one modality stacked: with an RGB-D sensor, predict depth from colour and colour from depth, and the concatenated feature has learned the relationship between geometry and appearance without anyone labelling either.

Where this family lands, measured on the same ImageNet probe as before, is around 40.7% at conv4 for colourisation against RotNet's 50.0% and the fully supervised 59.7%. Reconstruction was competitive but not leading, and for several years the reasonable conclusion was that predicting pixels is a worse pretext task than predicting a discrete transformation parameter, because too much of the model's capacity goes into reproducing texture that carries no semantic content. Section 13.6 is the story of that conclusion being overturned, and it was overturned by changing the architecture, not the objective.

13.5. Tracking, from nobody's tracking labels

Everything so far uses the encoder afterwards and discards the mechanism that trained it. The most striking result in this lecture does the opposite: the mechanism *is* the useful thing, and it turns out to be a tracker that nobody asked for.

The setup is colourisation moved to video (Vondrick et al., 2018). One frame of a video is given in colour as a reference; the following frames arrive in grayscale, and the task is to colour them. Since objects keep their colour as they move, the answer for any target pixel is sitting somewhere in the reference frame — the problem is finding *where*.

The architectural decision that makes this work is a restriction. The network is not allowed to generate colour. It sees only grayscale, computes an embedding f_i at every spatial location of both frames, and forms a soft correspondence between them by inner-product similarity,

$$A_{ij} = \frac{\exp(f_i^\top f_j)}{\sum_k \exp(f_k^\top f_j)} \tag{13.2}$$

where i indexes reference locations, j indexes target locations, and the softmax over k makes each target location's weights sum to one across the reference frame. The predicted colour is then nothing but a weighted average of reference colours,

$$y_j = \sum_i A_{ij}\, c_i \tag{13.3}$$

with c_i the reference frame's known colour at location i. This is precisely the attention of Section 9.4 — Equation 13.2 is a softmax over dot-product scores and Equation 13.3 is the weighted sum of values — with the reference frame supplying both keys and values and the target frame supplying queries. As in the single-image case the colour targets are quantised rather than regressed, here by k-means into sixteen clusters with a cross-entropy loss, for the same reason: an average of plausible colours is grey.

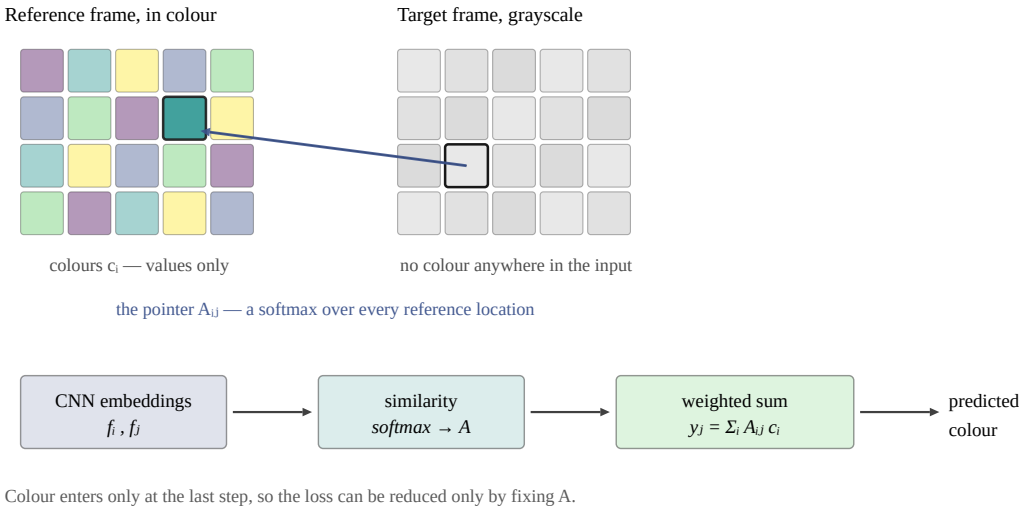


Figure 13.5.: The pointer. Both frames enter as grayscale, so nothing in the network's input carries colour; the only route from the reference colours to the prediction is the weight matrix A, and the loss can only be reduced by making A point at the right place. What the network is forced to learn is therefore correspondence, and colour is merely the signal that reveals whether the correspondence was right.

Because colour enters the prediction only through Equation 13.3, the loss can be reduced in exactly one way: by improving A. There is no capacity anywhere in the model for hallucinating a plausible colour from grayscale texture, which is what a colourisation network would ordinarily do. The gradient has a single path to travel, and it travels into the embedding that determines which reference pixel each target pixel points at. The network is trained on colour and learns correspondence, because correspondence is the only variable it has.

The payoff is that A can carry anything. Take a video with one annotated first frame — a segmentation mask, or a set of pose keypoints — and propagate that annotation forward with

the same Equation 13.3, substituting mask values or keypoint heatmaps for colours. The model has never seen a segmentation mask or a keypoint. It was trained on unlabelled video with a colour loss, and it tracks.

That is a genuinely different kind of result from the rest of this section. Rotation prediction gives you an encoder that you then have to train on something. Video colourisation gives you a working tracker as a side effect of a task that has nothing to do with tracking, which is what people mean when they say a capability *emerges*. The same phrase will recur in Section 13.12, for the same reason: a model trained with no notion of objects turns out to have found them anyway.

13.6. Masked autoencoders

Five years after context encoders concluded that pixel reconstruction makes a mediocre pretext task, masked autoencoders (He et al., 2021) reached the opposite conclusion with the same objective. Nothing changed about the loss — it is still mean squared error on the pixels that were removed. What changed is the architecture, and the change is worth understanding in detail, because it is the current default for pre-training vision models on raw data.

The image is cut into non-overlapping patches, exactly as in the ViT of Section 9.12. A large fraction of them — 75% in the default configuration — is sampled uniformly at random and discarded. What survives is fed to a Transformer encoder, and a decoder reconstructs the full image, with the loss computed only over the patches that were removed.

Two decisions inside that sentence carry the result.

The masking ratio is extreme. BERT masks 15% of tokens; MAE masks 75% of patches. The difference is not a hyperparameter tuned to a different value but a consequence of what the two media contain. A word is a nearly irreducible unit of meaning, and a missing word must be inferred from syntax and context. A 16×16 patch of a photograph is heavily redundant with its neighbours, and if only 15% are missing the task collapses into interpolation — extend the texture, continue the edge, done, with no understanding of the scene involved. Removing three quarters of the image destroys enough that local extrapolation cannot work and the network has to hold some model of what the object is. The difficulty is the point, and this is a general property of masking objectives: their value comes from being *just* solvable.

The encoder never sees a mask token. This is the structural asymmetry, and it is not obvious. The natural implementation would replace each removed patch with a learned placeholder and run the encoder over the full-length sequence. MAE does not. Removed patches are simply absent from the encoder's input, so the encoder processes a quarter-length sequence of real image patches. Only the decoder assembles a full-length sequence, interleaving the encoded visible patches with copies of a single shared, learned mask token and adding positional embeddings so each one knows where it belongs. The decoder is deliberately small — the default has under a tenth of the encoder's computation per token — and it is thrown away after pre-training, so its design is unconstrained by anything the downstream task needs.

Both consequences of the asymmetry are large. The first is arithmetic. A Transformer block on n tokens of width d costs about $24nd^2$ in its projections and feed-forward and $4n^2d$ in the attention matmuls, so shortening the sequence by a factor of four makes the projections four times cheaper and the attention *sixteen* times cheaper. Evaluating that model for the paper's actual configuration — a 24-block, 1024-wide ViT-L encoder, an 8-block, 512-wide decoder, 196 patches, 75% masked — gives a 3.3× reduction in total pre-training FLOPs against the same model with mask tokens in the encoder, which is exactly the figure the paper reports. The

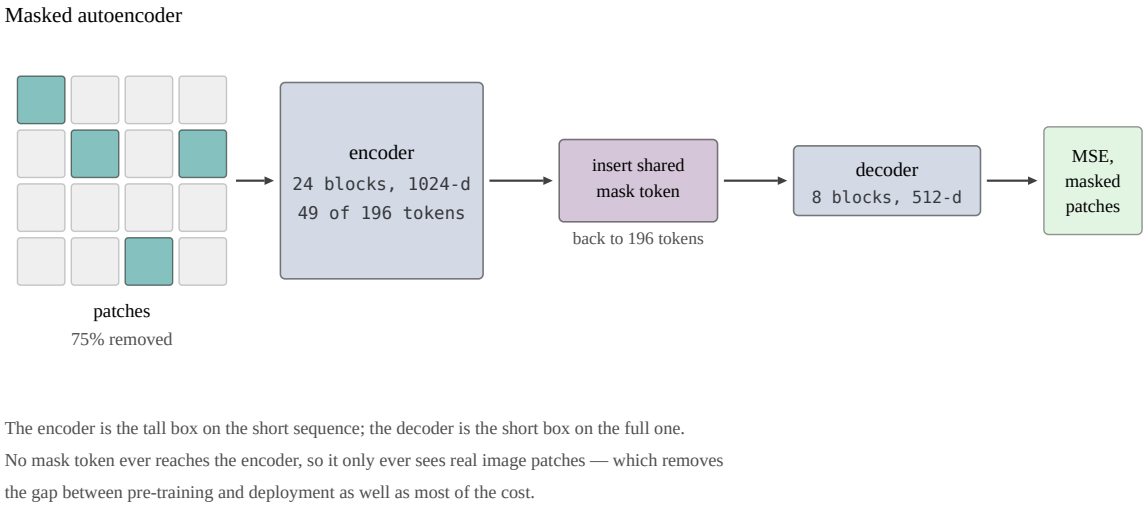


Figure 13.6.: The asymmetry. The encoder's input sequence has length vN, where v is the visible fraction, and the encoder is drawn tall because it is the heavy network; the decoder sees the full length N and is drawn short because it costs under a tenth as much per token. Both the compute argument and the accuracy argument follow from where the sequence is short and where the network is heavy.

realised wall-clock speedup was 2.8×, and rose to 4.1× for a larger encoder with a one-block decoder.

The second consequence is accuracy, and it is the less obvious of the two. Putting mask tokens into the encoder costs 14 points of linear-probing accuracy. The reason is a distribution mismatch: an encoder trained on inputs that are three-quarters placeholder is then deployed on uncorrupted images it has never seen, and the gap degrades it. Excluding mask tokens constrains the encoder to only ever look at real patches, so pre-training and deployment agree. Cheaper *and* better is an unusual combination, and it is what made the design stick.

The results justify the attention the method gets. A ViT-H pre-trained this way and fine-tuned on ImageNet-1K reaches 87.8% top-1, the best result at the time for any method using only ImageNet-1K data, and it beat the contrastive methods of Section 13.9 and Section 13.12 that had held the lead. For scale, a ViT-L trained from scratch on the same data reaches 82.5%; every masking ratio in the paper's sweep beats that.

The ablations are where the divergence between evaluation protocols becomes impossible to ignore, and it is worth dwelling on. Under linear probing, accuracy climbs steadily with the masking ratio from 54.6% at the low end to 73.5% at 75% — a spread of nearly twenty points. Under fine-tuning, everything from 40% to 80% works about equally well and the total spread is under two points. Decoder depth behaves the same way: a deep decoder helps linear probing substantially and barely moves fine-tuning.

The explanation is that the two protocols measure different things, and the paper's reading of it is convincing. The last layers of an autoencoder specialise in reconstruction, which is a pixel task rather than a recognition task. A deeper decoder absorbs more of that specialisation and leaves the encoder's output more abstract — which is precisely what a linear probe needs, since a linear probe cannot undo a representation tilted toward pixels. Fine-tuning can undo it, so fine-tuning does not care. The practical warning is that linear probing is not a neutral measure of representation quality; it is a measure of *linearly accessible* quality, and a method optimised against it can be optimised in the wrong direction.

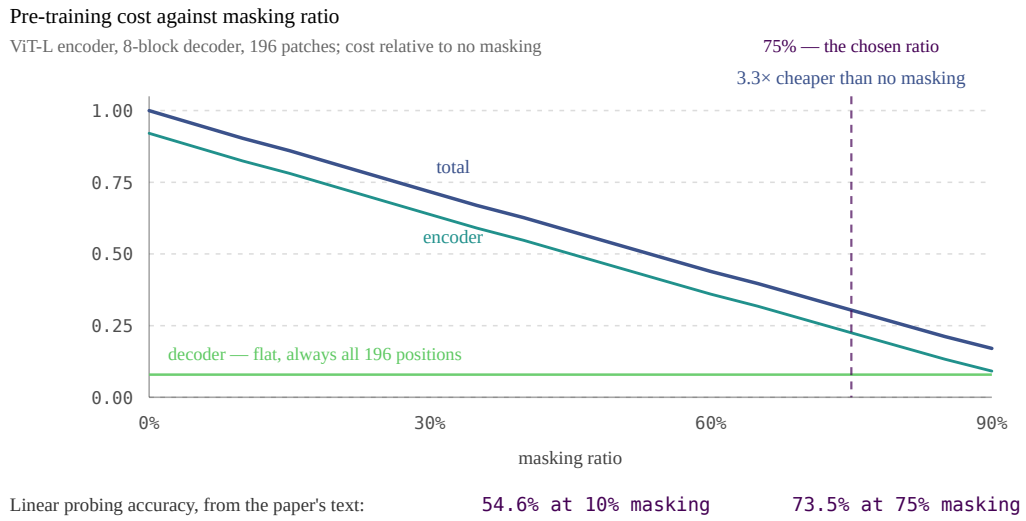


Figure 13.7.: Pre-training cost against masking ratio, evaluated from the per-block Transformer cost model in the figure's generator for MAE's ViT-L configuration. The encoder curve does bend — attention is quadratic in sequence length while everything else is linear — but only slightly, because at 196 tokens attention is just 3% of the encoder's cost; the saving is overwhelmingly the linear term. The decoder cost is flat, since it always processes all 196 positions, which is why it has to be small. The two linear-probing accuracies the paper states in text are noted below the axis.

13.7. What all the pretext tasks were approximating

Step back from the individual methods and the shared weakness is easy to state. Every task in Section 13.3 and Section 13.4 had to be invented. Somebody noticed that rotation is recoverable, or that colour is separable, or that patches have positions, and each invention required its own analysis of what shortcut might defeat it. There is no procedure for generating the next one, and the features you get are shaped by the accidents of whichever task you happened to think of.

There is also a common thread running underneath all of them that none of them states directly. A rotated image is the same object as the unrotated one. A jigsaw permutation of an image is the same object. A grayscale image is the same object as its colour version, and a masked image is the same object as the whole. Every one of these tasks is an elaborate way of asserting that a particular transformation preserves identity — and then a specific pretext head is bolted on to force the network to notice.

So assert it directly. Take an image, produce two views of it by transformations that preserve what it depicts, and require the encoder to map them near each other. Take a different image, and require the encoder to map it far away. There is no head to design, no shortcut particular to one transformation, and no invention required beyond the choice of which transformations count as identity-preserving — which is a much smaller and more transferable design decision than inventing a task. That is **contrastive representation learning**, and it dominated the field from roughly 2019 onward.

13.8. The contrastive objective

Write x for a reference image, x^+ for a view of the same image, and $x_1^-, \dots, x_{N-1}^-$ for views of different images. Let f be the encoder and s a scoring function on pairs of representations — in practice cosine similarity, scaled. What is wanted is

$$s(f(x), f(x^+)) \;\gg\; s(f(x), f(x^-)) \tag{13.4}$$

for every negative. Turning that wish into a differentiable loss is done by the standard device: exponentiate the scores, normalise, and take the negative log probability of the positive,

$$L = -\log \frac{\exp(s(f(x), f(x^+)))}{\exp(s(f(x), f(x^+))) + \sum_{j=1}^{N-1} \exp(s(f(x), f(x_j^-)))} \tag{13.5}$$

which is exactly the cross-entropy of an N-way softmax classifier whose correct answer is always index zero. The network's job, restated, is to pick the positive out of a lineup of N candidates. Nothing here is new machinery — Equation 13.5 is the softmax loss of Section 3.6 with the classes redefined at every step — and that is the appeal, since all the optimisation behaviour is already understood.

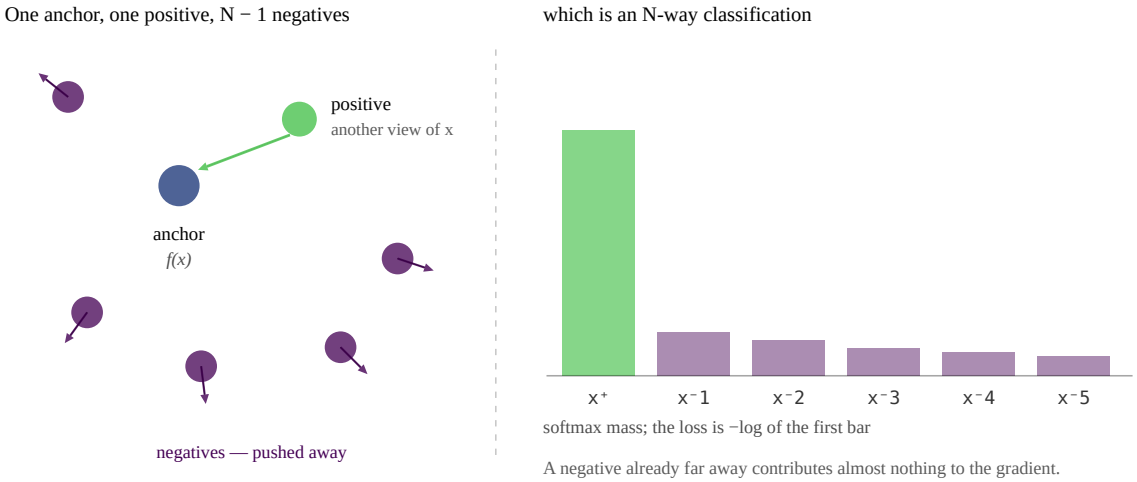


Figure 13.8.: The lineup. One anchor, one positive that is another view of the same image, and $N-1$ negatives drawn from other images. The gradient of Equation 13.5 pulls the positive toward the anchor and pushes every negative away, with each negative's push weighted by how much probability mass it currently steals — so a negative that is already far away contributes almost nothing, and the hard ones dominate.

Equation 13.5 is the **InfoNCE** loss, introduced with the contrastive predictive coding paper of van den Oord et al., 2018, and its name records a claim about what it does. Mutual information $I(x; x^+)$ measures how much knowing one variable tells you about the other. The paper shows that

$$I(x; x^+) \;\geq\; \log N - L_N \tag{13.6}$$

where L_N is the InfoNCE loss at N samples. Since the loss appears negated, minimising it raises a lower bound on the mutual information between the two views — which is a satisfying story,

because "keep whatever the two views share and discard whatever they do not" is a reasonable description of what a good representation should do.

Equation 13.6 is also where the argument for enormous batches comes from, and it is worth stating more carefully than it usually is. L_N is a cross-entropy and so is never negative, which means the right-hand side of Equation 13.6 never exceeds $\log N$. The bound cannot certify more than $\log N$ nats of mutual information no matter how good the encoder becomes: with $N = 256$ that ceiling is about 5.5 nats, and it is a property of the objective rather than of the model. Increasing N is the only way to raise it. That is a sharper claim than "more negatives tighten the bound", and it explains why the methods in the next two sections are so preoccupied with where to get more negatives from — the whole engineering effort in Section 13.9 and Section 13.10 is aimed at this one term.

One caveat belongs here, because the mutual-information framing is more often repeated than examined. Later analyses (Tschannen et al., 2020) found that tighter mutual-information estimators do not produce better representations, and that the success of these methods depends heavily on the encoder architecture and on which transformations generate the views — neither of which appears anywhere in Equation 13.6. The bound is a genuine theorem and a good motivation; it is not an explanation of why contrastive learning works. Treat the $\log N$ ceiling as the reliable part.

13.9. SimCLR

SimCLR (Chen et al., 2020) is Equation 13.5 with the three remaining blanks filled in as simply as possible, and its contribution is largely the demonstration that filling them in carefully is enough.

Positives come from augmentation. Each image in the batch is passed through two independently sampled augmentations — random resized crop, colour distortion, Gaussian blur — producing two views. The two views of one image are a positive pair; every other view in the batch is a negative. A batch of N images therefore yields $2N$ encoded views, each of which takes a turn as the anchor, giving $2N - 2$ negatives per anchor.

Which augmentations are used turns out to matter enormously, and the paper's ablation of this is its most useful practical finding. Random cropping alone is weak; colour distortion alone is weak; the *composition* of the two is what works. The reason is a shortcut of the sort Section 13.3 was full of. Two crops of the same photograph share a colour histogram, and a network can match them on that alone without looking at content at all. Distorting the colours independently in each view destroys the shortcut and forces the match to be made on structure.

The score is cosine similarity between ℓ_2-normalised embeddings, divided by a temperature τ, which gives the NT-Xent loss — normalised temperature-scaled cross entropy. The normalisation removes vector magnitude from the comparison and the temperature controls how sharply the softmax in Equation 13.5 distinguishes hard negatives from easy ones; both matter, and the paper ablates both.

The third blank is where the loss is applied, and here SimCLR does something that looks redundant and is not. The encoder produces a representation h; a small MLP projection head g maps h to a second vector z; and the contrastive loss is computed on z. Then the projection head is discarded and h is what gets used downstream. Training on one layer and deploying a different one is strange until you ask what the loss does to whatever it acts on. Equation 13.5 demands invariance to the augmentations, so it will happily destroy any information the augmentations changed — colour, orientation, scale. That is exactly right for the objective and often wrong for

the downstream task, which might well need to know what colour the object is. Interposing the head gives the loss a layer to flatten while leaving h upstream of the damage, and the measured gap between using h and using z is substantial.

The results were the ones that made the field pay attention. Linear probing on SimCLR features from a ResNet-50 (4×) reaches 76.5% top-1 on ImageNet, which matches a supervised ResNet-50 trained with all the labels; the same method on an ordinary ResNet-50 gets 69.3%, and the gap is the price of generality — a representation that has to serve unknown tasks needs more capacity than one aimed at a thousand known classes. Fine-tuned on 1% of ImageNet's labels it reaches 85.8% top-5, beating AlexNet trained on all of them.

The cost is where the $\log N$ ceiling of Equation 13.6 shows up as an engineering bill. SimCLR's negatives all come from the current batch, so raising N means raising the batch size, and the paper's best configurations use batches of 4096 and 8192 — the latter giving 16,382 negatives per positive. A batch that size does not fit on one accelerator, needs distributed training and the LARS optimiser to remain stable, and puts the method out of reach of anyone without a TPU pod. The negatives and the batch are welded together, and that is the flaw the next method attacks.

13.10. MoCo

Momentum contrast (He et al., 2020) observes that the coupling is an implementation accident rather than a requirement. A negative only needs to be an encoded view of some other image; nothing says it has to have been encoded during this step. So keep a queue.

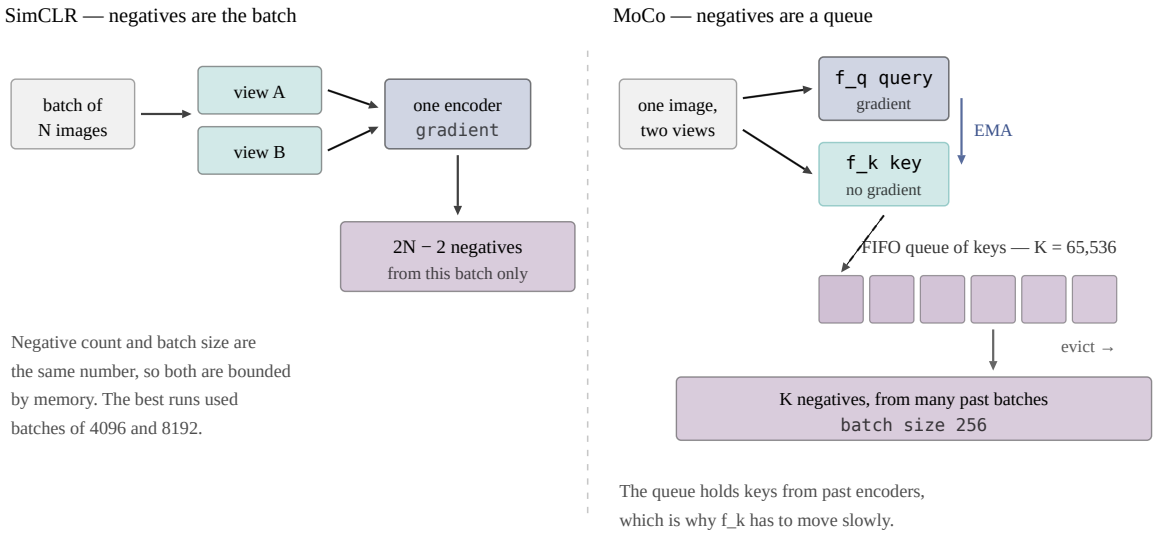


Figure 13.9.: Where the negatives come from. SimCLR draws them from the current batch, so the negative count and the batch size are the same number and both are bounded by memory. MoCo maintains a FIFO queue of encoded keys that is orders of magnitude larger than the batch, enqueuing the current batch and evicting the oldest. Gradients flow only along the query path — the dashed path carries none — which is what makes the queue affordable and also what creates the problem the momentum update solves.

Concretely: one view of each image becomes a *query*, encoded by f_q; the other becomes a *key*, encoded by f_k. Keys are pushed into a FIFO queue as each batch is processed and the oldest

are evicted, so the queue holds keys from many recent batches — MoCo uses 65,536 of them with a batch size of 256. The InfoNCE loss compares the query against its own key and against everything in the queue.

Backpropagation through the queue is impossible: those keys were produced several steps ago, their activations are long gone, and storing them would defeat the memory saving entirely. So the key path carries no gradient at all, and only f_q is updated by the optimiser. Which leaves the question of where f_k comes from.

Making f_k a copy of f_q — recopied after every optimiser step — does not work, and the reason is the crux of the method. The queue holds keys produced by whatever f_k was at the time each key was enqueued. If f_k changes quickly, keys near the front and keys near the back of the queue were produced by encoders that disagree, and comparing a single query against all of them is comparing against an incoherent dictionary. Similarities computed against inconsistent keys carry no reliable signal.

The fix is to make f_k change slowly, by an exponential moving average of the query encoder:

$$\theta_k \leftarrow m\,\theta_k + (1-m)\,\theta_q \tag{13.7}$$

with m close to one. The ablation makes the argument better than any explanation does. At $m = 0$ — the naive copy — training oscillates and fails to converge. At $m = 0.9$ the model reaches 55.2%; at $m = 0.99$, 57.8%; at $m = 0.999$, 59.0%. A hyperparameter whose useful range is between three and four nines is not being tuned, it is being used to enforce a constraint, and the constraint is that the dictionary must be self-consistent.

MoCo v2 is the unsurprising and effective conclusion: take SimCLR's MLP projection head and stronger augmentation, keep MoCo's queue and momentum encoder, and the combination beats SimCLR while training with a batch of 256 instead of 8192. That is the practical summary of this pair of papers. The projection head and the augmentation policy were the ideas; the queue was the engineering that made them affordable on ordinary hardware.

13.11. Contrasting sequences instead of instances

SimCLR and MoCo are *instance* contrastive learning: the unit of identity is one image, and positives are views of it. Contrastive predictive coding — the paper that introduced Equation 13.5 in the first place — contrasts along a different axis. The unit is a position in a sequence, and the positive is what actually comes next.

The construction has three parts. An encoder maps each element of the sequence to a latent, $z_t = g_{\text{enc}}(x_t)$. An autoregressive model, a GRU in the original paper, summarises everything up to the present into a context vector $c_t = g_{\text{ar}}(z_{\le t})$. And then, for each prediction offset k, a bilinear score

$$s_k(x_{t+k}, c_t) = z_{t+k}^{\top} W_k\, c_t \tag{13.8}$$

is fed to Equation 13.5, with the true future latent z_{t+k} as the positive and latents drawn from elsewhere in the data as negatives. A separate W_k per offset lets the model predict differently at different horizons, which it must, since one step ahead and twelve steps ahead are different problems.

The design decision worth extracting is that the prediction happens in latent space, not in the data. A model that predicted the actual future waveform or the actual future pixels would have to spend most of its capacity on the high-frequency detail that dominates those signals and carries almost none of the structure — the same complaint Section 13.4 raised about reconstruction losses. Scoring a latent against a context sidesteps this: the model never has to represent $p(x_{t+k})$, only to tell the real continuation from an impostor, and it can discard whatever detail does not help with that discrimination.

The formulation is indifferent to what the sequence contains. On audio it is applied directly to a waveform, and a linear classifier on the resulting representations recovers phonemes and speaker identity. On images the sequence has to be manufactured: cut the image into a grid of patches, read the rows top to bottom as if they were time, and use the upper rows as context for predicting the lower ones. That works, and it is worth being clear that it works less well than the instance-level methods — an image has no intrinsic ordering, and imposing one throws away the two-dimensional structure that convolutions and attention exploit. CPC's importance to vision is mostly that its loss function outlived its architecture.

13.12. DINO, and doing without negatives

Everything since Section 13.8 has been organised around negatives, and negatives are a nuisance: they are what forces the batch to be enormous or the queue to exist, and they are semantically wrong in an obvious way, since two different images of dogs are treated as things that must be pushed apart. DINO (Caron et al., 2021) does without them.

The frame is knowledge distillation with nothing to distil from. Two networks with identical architecture, a **student** g_{θ_s} and a **teacher** g_{θ_t}, each map an image to a K-dimensional distribution through a softmax. Two augmented views of the same image are produced; the student sees one, the teacher the other; and the student is trained to match the teacher's output distribution under cross entropy. Gradients flow only through the student — the teacher carries a stop-gradient — and the teacher's weights are an exponential moving average of the student's, the same Equation 13.7 that MoCo used. The teacher is the student's own recent past, which is why the method's name expands to self-distillation with no labels.

Nothing in that description prevents the trivial solution. A student and teacher that both emit the same constant distribution for every image achieve zero loss and have learned nothing. Every method in this chapter has some device for ruling that out — contrastive learning uses negatives, which make a constant output maximally wrong — and DINO's device is two operations on the teacher's output that are individually harmful and jointly sufficient.

Centring maintains a running mean c of teacher outputs across the batch, updated by its own moving average, and subtracts it before the softmax. This stops any single output dimension from dominating, but on its own it pushes the teacher's distribution toward uniform — which is the other collapse. **Sharpening** uses a low temperature in the teacher's softmax, concentrating the distribution, which pushes away from uniform but on its own encourages exactly the one-dimension domination that centring exists to prevent. The two corrections have opposite biases and neither is stable alone; run together, they balance, and no negatives are needed. It is a genuinely different mechanism from Equation 13.5, which is why DINO is often described as contrastive-adjacent rather than contrastive.

One more component matters for what emerges. DINO uses multi-crop: the teacher sees large global crops and the student sees several small local ones, and the student must predict the

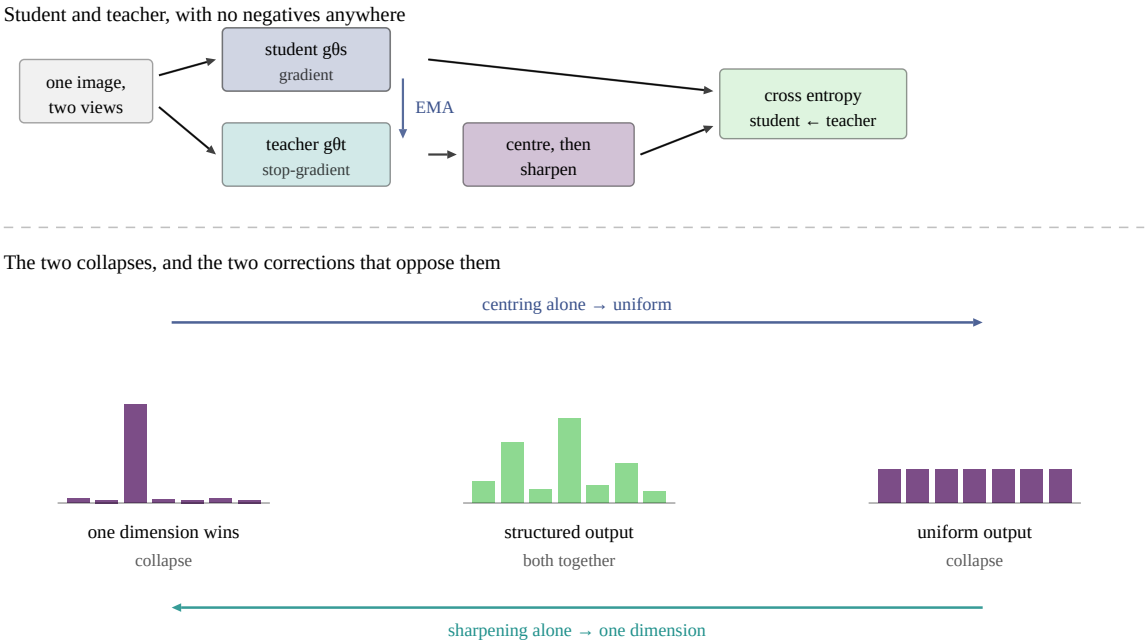


Figure 13.10.: The two failure modes and the two corrections. Collapse can be to a single dominant dimension or to the uniform distribution, and the two fixes push in opposite directions: centring subtracts a running mean of teacher outputs, which stops any dimension from dominating but drives everything toward uniform; sharpening uses a low softmax temperature on the teacher, which drives outputs away from uniform but toward a single dimension. Neither alone is stable. Applied together they hold the output between the two collapses.

global view's distribution from a local patch. That is a local-to-global objective, and it forces the representation of a small region to carry information about the whole object it belongs to.

The results include the best pure-feature numbers of the chapter. A k-nearest-neighbour classifier on frozen DINO features — no training of any kind on top, just look up the nearest neighbours and vote — reaches 78.3% top-1 on ImageNet, which is a much stronger claim than a linear probe, since a k-NN classifier has no parameters to fit and can only work if the representation's geometry is already right. Linear evaluation with a ViT-Base reaches 80.1%.

And then the property the paper is named for. Visualising the attention of the ViT's class token in a DINO-trained model shows it attending to object boundaries — the segmentation of the scene is present in the attention maps, sharply enough to be used, and it does not appear this clearly in a ViT trained with ImageNet labels or in a convnet. Nothing in the training objective mentions objects, segmentation, or spatial extent. This is the second time in this chapter that a capability nobody trained for turned up anyway, after Section 13.5, and the two cases share a shape: an objective that forces a model to relate one part of an image to another leaves behind a machine for finding those relations, and the machine is usable for things the objective knew nothing about.

DINOv2 scaled the recipe — much more data, curated rather than scraped, and a larger model — into features good enough to be used frozen across depth estimation, segmentation and retrieval without task-specific fine-tuning, which is close to the original promise of Section 13.1: train once on unlabelled images, use the representation everywhere.

13.13. Where this leaves things

Two families came out of this lecture and both are alive. Reconstruction, dismissed in 2016 and rehabilitated by Section 13.6's architectural asymmetry, learns by predicting the parts of the input it was not shown. Joint-embedding methods — contrastive in Section 13.9 and Section 13.10, distillation-based in Section 13.12 — learn by requiring two views of one thing to agree, and never model pixels at all. They have different failure modes, different costs, and different downstream profiles: masked autoencoders fine-tune superbly and probe indifferently, joint-embedding methods produce representations that are linearly separable and even nearest-neighbour separable straight out of the box.

What has not survived is the part of the lecture that came first. Nobody now designs a rotation-prediction task, and the reason is worth keeping: hand-designed pretext tasks buy features shaped by the accidents of the task, and the two surviving families both replaced the invented task with a statement about what the data is — that an image is redundant enough to be reconstructed from a quarter of itself, or that two views of one scene depict one thing. Those are claims about images rather than puzzles about them, which is why they generalise.

The thread running through both is that supervision was never really the scarce resource; *labels* were. Everything here is supervised learning in the mechanical sense — a target, a loss, a gradient — and the whole contribution is arranging for the target to be something the data already knows. That is also what the next lecture is about from a different direction: a generative model, asked to produce samples from the distribution of natural images, is solving the hardest version of the same problem, and its representations come from the same place.

Part IV.

Generative Models

14. Generative Models I

Lecture 13

Based on Lecture 13 of CS231n, Stanford University, Spring 2025.

14.1. Two axes, not one

Almost everything in this course so far has been supervised learning. A dataset of pairs (x, y), a function to be learned that maps x to y, and a loss that measures the discrepancy on examples where the answer is known. Classification, detection, segmentation and captioning differ in what y is — a category, a set of boxes, a per-pixel label, a sentence — but not in their structure. The thing you want to predict is exactly the thing your dataset records, and the work is to fit a mapping that agrees with the data and keeps agreeing on inputs it has not seen.

Unsupervised learning is described by what it lacks: there is no y. You have images and nothing else, and the goal is stated vaguely on purpose — find structure in the data. The vagueness is real. k-means finds clusters, principal component analysis finds a low-dimensional subspace that explains most of the variation, and density estimation fits a probability distribution. These are all called unsupervised, and they produce objects of entirely different kinds.

It helps to notice that they are not competing answers to one question but points along a spectrum, and that the spectrum has a direction. At one end sits **density estimation**: model the distribution that generated the data, in full, with nothing thrown away. At the other sits **representation learning**: extract from each image a short vector that retains what matters and discards the rest, where "what matters" is defined only by whatever downstream task eventually uses the vector. Clustering and PCA live near the representation end. A model that assigns a probability to every possible image lives at the density end. The previous chapter's contrastive methods are near-pure representation learning: SimCLR never assigns a probability to an image, and could not be asked to.

The second axis is orthogonal to the first, and it is the one this chapter is about. Cutting across supervised and unsupervised is the distinction between **generative** and **discriminative** models. This axis is inherently probabilistic: to be on it at all, a model has to commit to a probability distribution over some of its variables. Clustering and PCA sit off the axis entirely — both admit probabilistic readings, but neither requires one.

The distinction turns out to be sharper than it looks, and the reason has nothing to do with architectures.

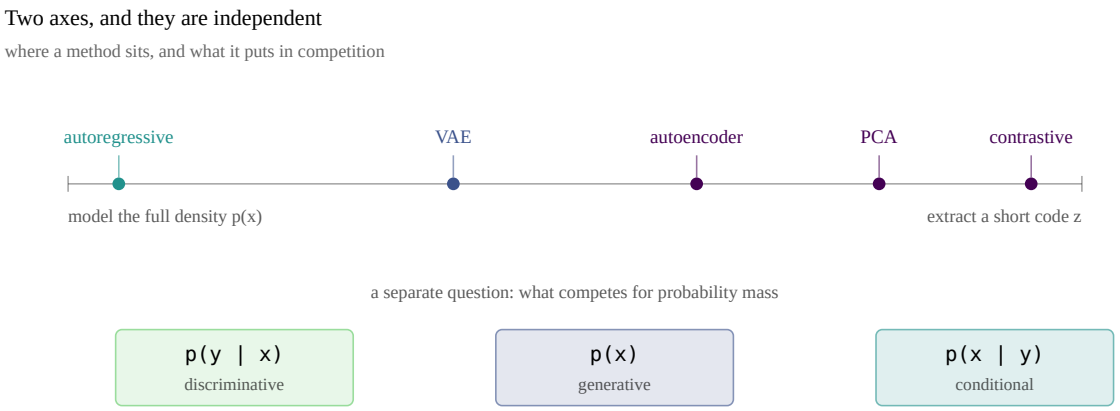


Figure 14.1.: The two axes the lecture separates. Supervision — whether a human produced the target — is orthogonal to what the method actually produces, which runs from a full density over the data to a short vector per example. The two families in this chapter are placed by what they compute: an autoregressive model is at the density end and offers no useful latent, while an autoencoder is at the representation end and offers no density at all. The variational autoencoder is the interesting case, because it is trying to reach both ends at once and pays for the attempt.

14.2. Probability mass is a fixed budget

Write x for the data — a high-dimensional thing, an image — and y for a label, caption, or other comparatively small piece of side information. A discriminative model learns $p(y \mid x)$. A generative model learns $p(x)$. A conditional generative model learns $p(x \mid y)$. These are three arrangements of two symbols, and their differences look cosmetic.

They are not, and the reason is the property that makes a function a probability distribution in the first place. A density $p(x)$ assigns a non-negative number to every point in its domain subject to

$$\int p(x)\, dx = 1. \tag{14.1}$$

The integral runs over the entire space of possible values. This single constraint is where the force of probabilistic modelling comes from, because it means there is a fixed unit of probability mass to distribute and no more. Choosing a distribution is choosing how to apportion that unit across the whole domain, and every point in the domain is therefore in competition with every other. Raising the probability of one value necessarily lowers the probability of others. There is no way to be generous to everything.

So the question to ask of any probabilistic model is: **what is competing?** The symbols on the page look alike; the competitions they set up do not.

In a discriminative model $p(y \mid x)$, the competition is among labels, and it is run separately for each x. Fix an image and the model emits a distribution over the label set — for a two-class problem, a number for cat and a number for dog that sum to one. Change the image and you get a different distribution over the same label set. Crucially, **images do not compete with each other for anything.** Each image induces its own budget, spends it among the labels, and

is done. Nothing in the objective ever asks whether one image is more plausible than another, and nothing in the model's output could express such a judgement.

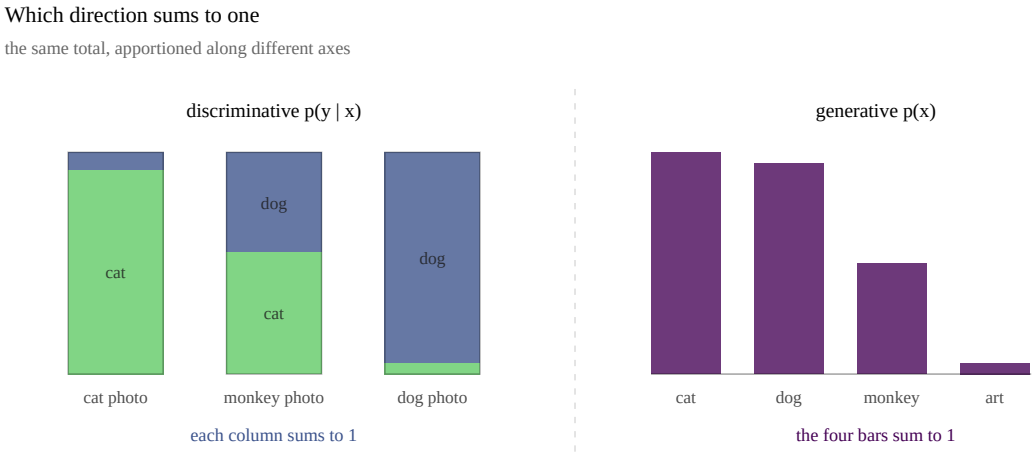


Figure 14.2.: The two competitions. On the left, a discriminative model gives each image its own budget to spend across the label set: the columns sum to one and nothing connects them, which is why the monkey must still be assigned to cat or dog. On the right, a generative model spends a single budget across the whole space of images, so the columns are no longer independent and the monkey can simply be assigned almost nothing. The difference is in which direction the bars are constrained to sum to one, and it is not a property of the architecture.

That has an immediate and slightly uncomfortable consequence: a discriminative model cannot reject its input. Fix the label vocabulary at cat and dog, feed the trained model a photograph of a monkey, and it is structurally obliged to answer with a distribution over cat and dog. It has nowhere to put the mass. It cannot say that the question does not apply, because Equation 14.1 is enforced along the label axis and the label axis contains no such option. This is not a training failure to be fixed with more data. It is the shape of the model.

A generative model $p(x)$ moves the competition. Now every image that could exist is competing with every other for one unit of mass, and the model gains exactly the capacity the discriminative one lacked: it can assign an input a probability near zero, which is a statement that the input is not the kind of thing this model is about. A density fitted to photographs of zoo animals should give a piece of abstract art essentially nothing, and that is a meaningful output rather than a refusal to answer.

The capacity is bought at a steep price, and it is worth being explicit about how steep. Apportioning mass across all possible images forces the model to answer questions it would rather avoid. Should an image of a three-legged dog receive more mass than an image of a three-armed monkey? Probably yes — a dog can lose a leg, and there is no comparable route to a third arm — but arriving at that answer requires something like a model of how the world produces the images it produces. The problem stated in one line, $\int p(x)\,dx = 1$, quietly contains most of visual understanding.

Conditional generative models are the arrangement that matters in practice, and they compound the difficulty in an interesting direction. For every value of y, the model induces a separate competition among all images. Conditioned on cat, the cat photograph should be high, the dog and monkey photographs middling — they are at least animals — and the abstract art near zero; conditioned on dog, a different apportionment of the same total. When y is a category label this

is a modest generalisation. When y is a paragraph of text, or an image plus a piece of text, the model is being asked to place a distribution over all images for every point in an enormous and richly structured conditioning space, which is a problem with no clean specification at all. All that changed from the discriminative case was the position of two symbols.

14.3. Bayes' rule connects all three

The three arrangements are not independent objects. They are related by

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)}, \tag{14.2}$$

which says that a discriminative model, an unconditional generative model, and a prior over labels together determine a conditional generative model. The identity rearranges, so any two of the pieces give you the third. In principle you never need to train a conditional generative model: train a classifier, train a density over images, count label frequencies, and multiply.

In practice nobody does this, and conditional generative models are trained directly from scratch. The reason is that Equation 14.2 is exact about the relationship and silent about the difficulty — it requires an unconditional $p(x)$ that is accurate enough to divide by, which is the hardest of the three objects to obtain. The relationship is still worth carrying, partly because it is the cleanest statement of how the families connect, and partly because it does get used: classifier-free guidance in diffusion models trains conditional and unconditional models jointly and combines their predictions at sampling time, which is Equation 14.2 doing real work rather than serving as a remark.

14.4. What a density is actually for

Discriminative models need no justification here; the whole course to this point has been their applications. Train one and you can assign labels, and the features it learns along the way transfer to tasks it was never shown, which is the transfer learning of Section 7.12.

Unconditional generative models are harder to defend, and the honest assessment is that on their own they have not turned out to be very useful. They give you outlier detection, which is a real if narrow application of the rejection capacity from the previous section. They give you a form of representation learning that needs no labels — the hope being that fitting $p(x)$ forces the model to learn something about images along the way — and the record there is poor: the contrastive methods of the previous chapter beat unconditional density estimation at producing transferable features, consistently and by a wide margin. And they give you samples. But a sample from $p(x)$ is an image you had no say in. The model hands you something plausible and you take what you are given, which makes unconditional sampling a mathematically interesting capability with very little practical purchase.

Conditional generative models are where nearly all of the value is, and it comes almost entirely from sampling. If y is a piece of text then sampling from $p(x \mid y)$ is text-to-image generation, and the conditioning is precisely the control that unconditional sampling lacks. You can also in principle classify with rejection — evaluate $p(x \mid y)$ across all y and decline to answer if every one is low — though this is rare in practice. The thing that is not rare, and that motivates the two chapters on generative models, is controllable synthesis.

There is a caveat worth attaching to the dismissal of unconditional models, because the field has partly overturned it since. Large language models are trained as unconditional density estimators over token sequences; the conditioning that makes them useful is supplied at sampling time by fixing a prefix, not by a separate conditioning pathway during training. The clean split between conditional and unconditional is a statement about how the distribution is factored, not about whether the model can be steered, and the sharpest counterexample to "unconditional models are useless" is the most commercially significant class of generative model in existence.

14.5. A taxonomy by what you can compute

The useful way to organise generative models is not by architecture. It is by which operations the model actually supports, because the operations turn out not to come as a package.

There are two you might want. You might want to **evaluate** the density: hand the model an x and get a number. You might want to **sample**: ask the model for a fresh x drawn from what it has learned. It is tempting to assume that anything which can do one can do the other — a distribution is a distribution — but the two are computationally quite different requests, and every generative model in use makes a trade between them.

Models that can return $p(x)$ for an input are **explicit density** models. Models that cannot, but can produce samples, are **implicit density** models: the density has been learned in the sense that the samples are distributed according to it, but there is no way to read the value out. That sounds like a pure loss, and it is often not, because for the applications that matter most you may genuinely not care what number the density assigns to any particular image. If all you want is a stream of good, varied samples, the ability to evaluate is a luxury. Meanwhile the explicit models are the ones for which sampling tends to be awkward, which is the reverse of what one would guess.

Each branch splits again, and this is the taxonomy Goodfellow's tutorial laid out, adapted. Among explicit models, some return the **tractable** density — the actual $p(x)$ the model defines, exactly, with no approximation. Autoregressive models are the example and they are the first half of this chapter. Others return an **approximate** density: a number that is not $p(x)$ but a computable bound on it. Variational autoencoders are that case, and they are the second half. Among implicit models, some sample **directly** — one forward pass of a network produces a sample, which is what a generative adversarial network does. Others sample **indirectly**, through an iterative procedure with no single feed-forward route to a sample. Diffusion models are the case, and they occupy the next chapter.

The distinction between the approximate-density leaf and the indirect-implicit leaf is worth stating carefully, because they can both be described as "you get something less than the real density". In the approximate case there is a number, and it is a bound on the true $p(x)$; you can compute it, compare it across inputs, and use it as a training objective. In the indirect implicit case there is no density value anywhere in the model to be found, and the only thing on offer is a sampling procedure. And the iterative procedure cannot be wrapped up and treated as a direct sampler, because the iteration converges to a true sample only in the limit. Diffusion sampling with finitely many steps gives an approximate sample, and the same was true of the Markov-chain methods that preceded it.

One notational habit to absorb before going further, because the lecture flags it and every paper in this area relies on it. Everything from here is written $p(x)$, unconditionally. The conditioning on y is dropped for readability, not because it is absent: every density, objective and bound

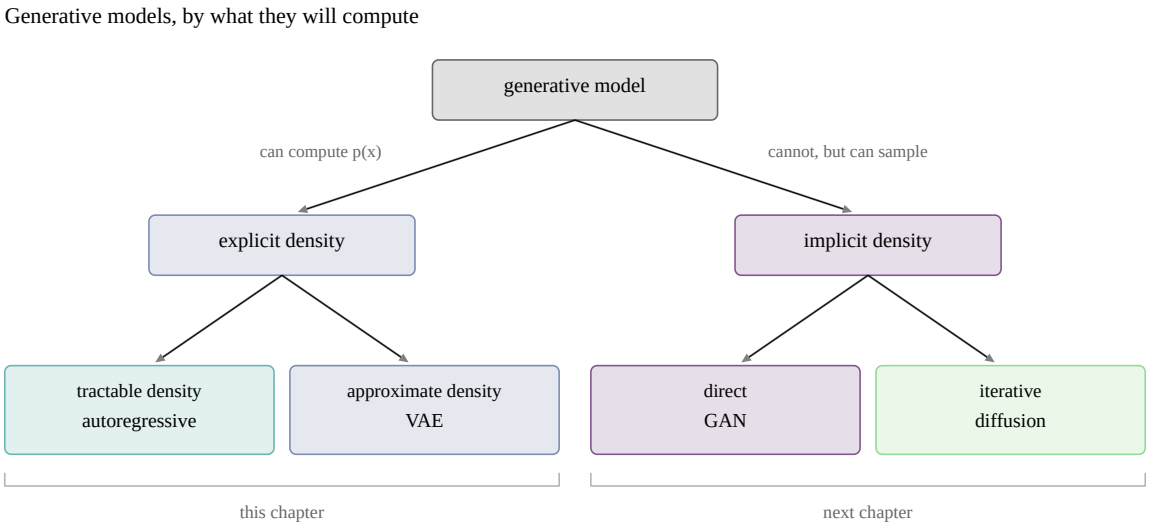


Figure 14.3.: The four leaves, arranged by what the model will do for you. The first split is whether a density value is obtainable at all; the second, on each side, is whether it is the true one and whether sampling is a single pass. Note that the two dimensions are not aligned: the leaf that gives you an exact density is the one that samples most slowly, and the leaf that samples in one pass gives you no density at all.

below has a $\mid y$ that can be reinstated wherever it is wanted, and in the models people actually deploy it is there. Read $p(x)$ as $p(x \mid y)$ with the label suppressed.

14.6. Maximum likelihood, and the direction of the divergence

Suppose the model is a neural network that takes an image x and weights W and returns a density, $p(x) = f(x, W)$. Given a dataset $x^{(1)}, \dots, x^{(N)}$, how should W be chosen? The answer that organises essentially all of explicit generative modelling is to pick the weights under which the data you actually observed is most likely:

$$W^* = \arg\max_W \prod_{i=1}^{N} p(x^{(i)}) = \arg\max_W \sum_{i=1}^{N} \log f(x^{(i)}, W). \tag{14.3}$$

The product over i assumes the samples are independent and identically distributed, so the joint density of the dataset factors. The step to the sum is the log trick, and it earns its place twice over: log is monotonic, so maximising the log of a quantity maximises the quantity, and it turns the product into a sum, which gradient descent handles and which does not underflow. In practice you will rarely see a density value in an implementation at all — models emit log-probabilities, losses are computed in log space, and this is for numerical survival rather than elegance. Superscripts index samples throughout; subscripts, from the next section, index parts of a single sample.

The word “likelihood” rather than “probability” marks what is being varied. Probability fixes the distribution and varies x: how much mass does this fixed density put on this point? Likelihood fixes the observed x and varies the distribution: among all the densities the network can express, which makes these particular samples least surprising? The equation is the same; the argument to the optimisation is not.

Underneath the whole exercise is an assumption that deserves saying out loud. There is a true distribution p_{data} from which the world drew the images you have. You cannot see it, and never will — you have a finite sample from it and nothing more. Maximum likelihood is one procedure for guessing at it from that sample, and it is worth knowing exactly what kind of guess it makes.

Take the average of Equation 14.3 over the dataset and let N grow, so the empirical average converges to an expectation under p_{data}:

$$\frac{1}{N}\sum_{i=1}^{N}\log p_\theta(x^{(i)}) \;\longrightarrow\; \mathbb{E}_{x\sim p_{\text{data}}}\big[\log p_\theta(x)\big] = -H(p_{\text{data}}) - D_{\text{KL}}(p_{\text{data}}\,\|\,p_\theta). \tag{14.4}$$

The entropy $H(p_{\text{data}})$ is a property of the world and does not depend on θ, so maximising likelihood is *exactly* minimising the Kullback–Leibler divergence from the data distribution to the model. That is a satisfying justification, and it also carries a warning, because KL divergence is asymmetric and Equation 14.4 fixes which direction you get. The expectation is taken under p_{data}, which means the model is punished savagely wherever the data has mass and the model does not — $\log p_\theta(x) \to -\infty$ — and not punished at all wherever the model has mass and the data does not. The objective is **mode-covering**: it insists the model cover everything the data does, and it is indifferent to mass spent on things the data never produces.

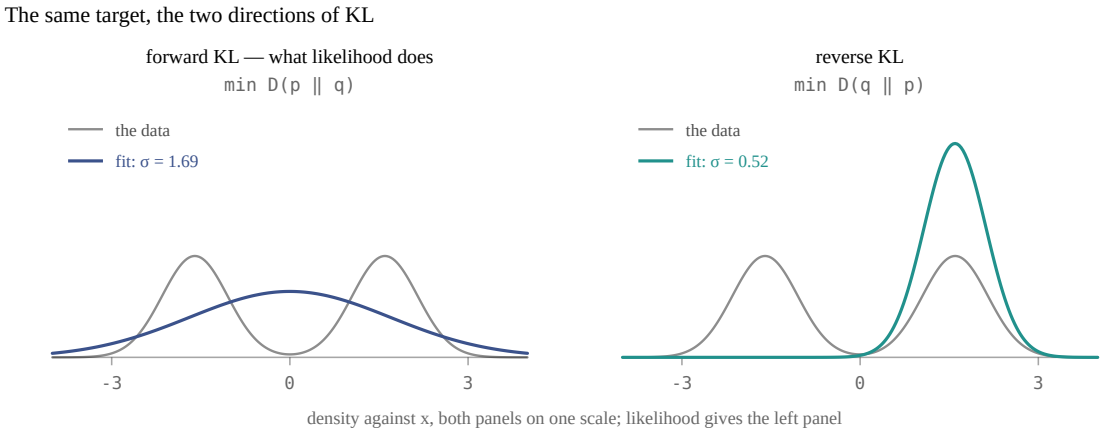


Figure 14.4.: The best single Gaussian fit to a bimodal target, under each direction of the KL divergence. Both fits are computed in the generator, the forward-KL one by moment matching (which is what minimising $D_{\text{KL}}(p\|q)$ over Gaussians reduces to) and the reverse-KL one by grid search over the objective, restricted to the right-hand mode since a symmetric target gives that direction two equivalent optima. Maximum likelihood gives the left panel: rather than commit to either mode, it spreads across both and puts substantial mass in the valley between them, where the data has almost none. When the model is a decoder over images, mass in the valley is what an average of two plausible images looks like, which is to say a blurry one.

This is not a technicality to file away. Every likelihood-trained generative model inherits the mode-covering bias, and it is the standard explanation for why models in this family produce samples that are blurry and slightly averaged rather than sharp and occasionally wrong. Adversarial models, which optimise something closer to the other direction, have the complementary failure: sharp samples, and whole modes of the data silently missing.

14.7. The chain rule needs no assumptions

Maximum likelihood tells you what to maximise but not how to write down $f(x, W)$ so that its output is a normalised density. That is the real difficulty, and it is where the model families diverge.

The autoregressive answer is to assume the data has a canonical decomposition into an ordered sequence of parts, $x = (x_1, x_2, \dots, x_T)$, and then apply the chain rule of probability:

$$p(x) = p(x_1, x_2, \dots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_1, \dots, x_{t-1}). \tag{14.5}$$

The remarkable thing about Equation 14.5 is that it assumes nothing. It is not a modelling approximation, not a conditional independence claim, not a factorisation that discards interactions. It is an identity, true of every joint distribution over every set of random variables. The only commitment is the choice of ordering, and the identity holds for any ordering you pick.

What the factorisation buys is that each factor is small. To see why that matters, count. For a vocabulary of V tokens, the joint distribution over a sequence of T of them is a table with V^T entries — a hundred thousand words and a twenty-token sentence puts the table beyond any conceivable representation. The factored form never asks for that table. It asks for one distribution over V options at a time, conditioned on a prefix, and the network's job is to map prefixes to V-way distributions. Exponential becomes linear in T, and the price is that the conditioning is now the network's problem rather than the table's.

And a network that maps a prefix to a distribution over the next element is something this course has already built twice. An RNN carries a hidden state forward that summarises the sequence so far and emits a distribution over the next element at each step, which is exactly Equation 14.5 with h_t standing in for $x_1, \dots, x_{t-1}$ — the language model of Section 8.6 was an autoregressive generative model, described in different vocabulary. A Transformer with a causal attention mask does the same thing without the recurrence: masking the attention matrix so position t attends only to positions $\leq t$ is precisely the constraint Equation 14.5 requires, as Section 9.11 set out. This is what a large language model is. There is no additional generative machinery — the next-token objective *is* maximum likelihood under a chain-rule factorisation, and the cross-entropy loss the model is trained with *is* $-\log p(x)$ summed over positions.

That correspondence makes the exactness concrete. Push a sequence through a trained Transformer once. At every position it has emitted a distribution over the vocabulary conditioned on the prefix; read off the probability it assigned to the token that actually occurred, and multiply across positions. The result is $p(x)$ for that sequence — not a bound, not an estimate, the value the model defines. One forward pass, and you have the exact density. This is what puts autoregressive models in the tractable-density leaf of Figure 14.3.

Sampling is the other direction and it is not symmetric. To draw a sample you have nothing to condition on, so you must generate x_1 from the model, feed it back, generate x_2 from the extended prefix, feed that back, and continue. Each step needs the previous step's output, so the T network evaluations are irreducibly sequential. Training parallelises perfectly — every position's target is known in advance, so one pass computes all T losses at once — and sampling parallelises not at all. That asymmetry is the defining property of the family, and for text, where T is a few thousand, it is an inconvenience.

14.8. Images do not come as sequences

Text fits Equation 14.5 almost suspiciously well. It is already one-dimensional, so the ordering is given rather than chosen, and it is already discrete, so each factor is a distribution over a finite vocabulary — which is to say a softmax, trained with cross-entropy, exactly the machinery this course has used since the second lecture.

Images have neither property. They are two-dimensional, so any ordering is imposed rather than found, and they are ordinarily thought of as continuous. The second objection is the easier one to dissolve, and the dissolution is worth appreciating for how blunt it is: images as actually stored are not continuous. A PNG or JPEG holds eight bits per channel, so a pixel is three integers in $[0, 255]$ and nothing else. The continuity was an idealisation the file format never had.

So an image can be flattened into a sequence in raster order — left to right along each row, top to bottom down the rows, and within each pixel the three channels in a fixed order — giving a one-dimensional sequence of discrete symbols with a vocabulary of 256. Every tool from the previous section now applies unchanged. This is PixelRNN and PixelCNN, which predict each subpixel as a 256-way classification and differ in how the prefix is summarised: the recurrent versions carry LSTM state across the image, and PixelCNN uses masked convolutions, where the mask zeroes the kernel weights that would otherwise read pixels later in the raster order.

Figure 14.5.: The causal structure a raster ordering imposes. Everything above the current pixel's row is available, along with everything to its left in the same row; everything to its right and below is not, regardless of how close it is. The masked convolution on the right is how this is enforced in a convolutional model: the kernel's weights over the future half of its footprint are held at zero, so the constraint is structural rather than something the network has to learn. The pixel immediately below the current one is a nearer neighbour than the one at the far end of the previous row, and the ordering makes it unavailable anyway.

The approach works, in the sense that it produces genuine likelihood numbers and they are good. Density on images is conventionally reported in bits per dimension — the negative log-likelihood per subpixel in base two, which is directly the number of bits an optimal compressor built from the model would spend. It is a scale with meaningful landmarks at both ends. A model that has learned nothing and treats every subpixel as uniform over 256 values costs exactly 8 bits per

dimension. A full multivariate Gaussian fitted to CIFAR-10 gets to 4.70. PixelCNN reaches 3.14, and the strongest of the recurrent variants 3.00, against a previous state of the art of 3.47.

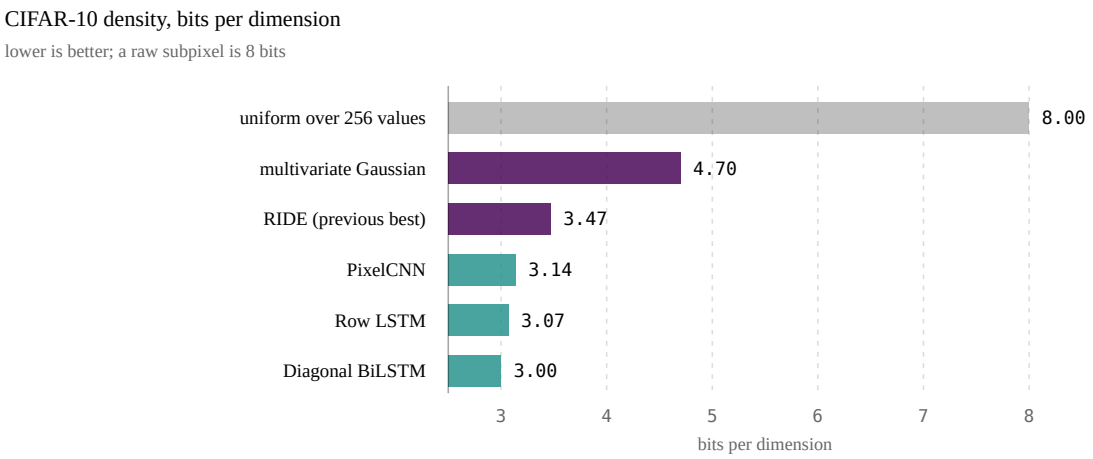


Figure 14.6.: CIFAR-10 test-set density in bits per dimension, from Table 5 of the PixelRNN paper, with the uniform baseline at exactly 8 bits marking what no modelling at all costs. Lower is better and the axis does not start at zero, because the interesting comparison is among the models rather than against the baseline. The ordering of the three PixelRNN variants tracks the size of the context each one can see, which is the paper's own explanation for the gap.

The internal ordering of those results is the more informative part. The three variants rank exactly as their receptive fields do: the bidirectional diagonal model, which can see the entire available prefix, beats the row-wise model, which sees a partially occluded prefix, which beats PixelCNN, which sees the least. Depth pushes in the same direction and saturates slowly: on the validation split the row-wise model goes from 3.30 bits per dimension with one layer to 3.06 with twelve. Both observations say the same thing — the bottleneck is how much of the prefix the model can actually condition on — and that is a reassuring thing to find, because it means the numbers are measuring the model's grasp of long-range structure rather than an artefact of the parameterisation.

Then there is the problem that killed the approach. Sampling requires one network evaluation per element, sequentially, and an image has a great many elements. A 32×32 colour image is a sequence of 3,072 subpixels, so a single sample costs 3,072 forward passes. That is tolerable, and it is why the published results are at 32×32 and 64×64 — where the paper reports 3.86 and 3.63 bits per dimension — rather than anywhere near a useful resolution. Go to 1024×1024, which is a good but hardly extravagant resolution, and the sequence is 3,145,728 subpixels long. Every one of those forward passes waits for the one before it.

Sequences of a few million elements are not beyond current models. But paying a few million sequential network evaluations for one image is not a practical generative model, and it is the reason autoregressive modelling of raw pixels was largely abandoned. The family came back — most of the strong image and video generators of the last few years are autoregressive somewhere inside — but not over pixels. The move that rescued it was to stop treating the raster as the sequence and instead learn a compression from images to a much shorter sequence of discrete tokens, then be autoregressive over those. The next chapter takes that up.

There is a second, quieter objection worth registering, because it is conceptual rather than computational. The raster ordering is arbitrary. Nothing about images makes left-to-right, top-to-bottom the right decomposition, and Equation 14.5 is exact for any ordering, which

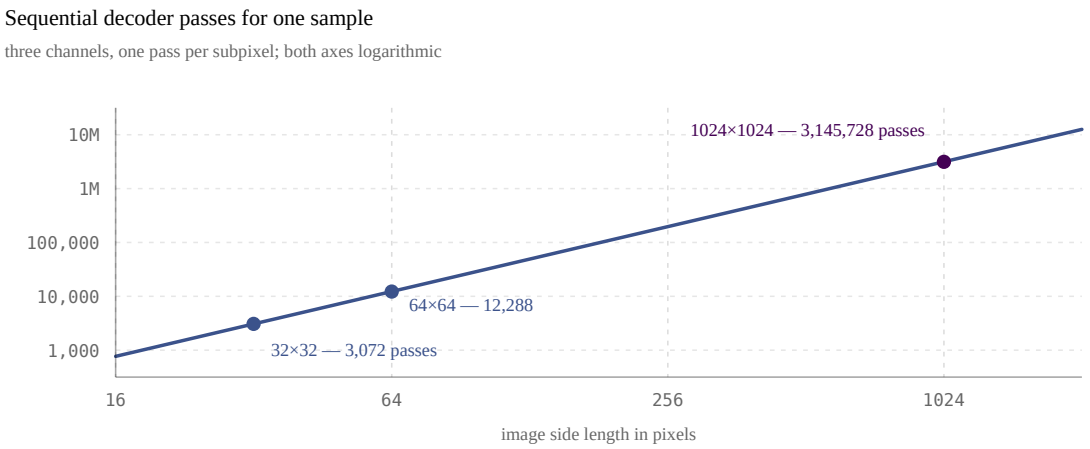


Figure 14.7.: Sequential decoder evaluations required for one sample, against image side length, for a three-channel image at one network pass per subpixel. Both axes are logarithmic. The two resolutions the PixelRNN paper reports results at are marked, along with the resolution a text-to-image model is expected to produce today; the cost is quadratic in side length, so the gap between them is four orders of magnitude and no amount of hardware closes it, because the passes cannot be run in parallel.

means the model is free to be wrong about which pixels are near each other and the mathematics will not object. The pixel directly below the current one is a close neighbour in the image and a distant one in the sequence, and a model conditioning on a raster prefix has to rediscover through its weights a spatial structure the ordering destroyed. That the convolutional variants do worse than the recurrent ones despite having the better spatial prior is a hint at how much is being asked here.

14.9. Reconstruction without a distribution

The second family in this chapter gives up something the first had and gets something the first lacked. Autoregressive models compute the exact density and hand you no representation of an image beyond the image itself; there is no vector in a PixelCNN that summarises what it is looking at. Variational autoencoders trade exactness for a latent space — an explicit, low-dimensional code per image, arriving as a by-product of the training — and the trade is worth making often enough that the family has outlasted its likelihood numbers.

Getting there needs the non-probabilistic version first, since the course has not built one. Take an encoder network — any architecture, a convolutional network is typical — mapping an image x to a vector z. Take a decoder mapping z back to an image $\hat{x}$. Train the pair so that $\hat{x}$ matches x under an L^2 loss.

Stated that baldly it is a strange thing to do. The objective is to approximate the identity function, which is already known, using an expensive network trained on a large dataset. The point is entirely in the constraint: z is made much smaller than x. A 1024×1024 colour image is about three million numbers and z might be 128 of them, so the network cannot pass the image through and must instead find a description short enough to fit and informative enough to reconstruct from. With an unconstrained z the problem is trivial and the model learns nothing; the bottleneck is what converts a pointless objective into a useful pressure. Principal component analysis is the linear special case of exactly this, which is a good way to see what the bottleneck

is doing — finding the subspace that loses the least — and how much a nonlinear encoder adds.

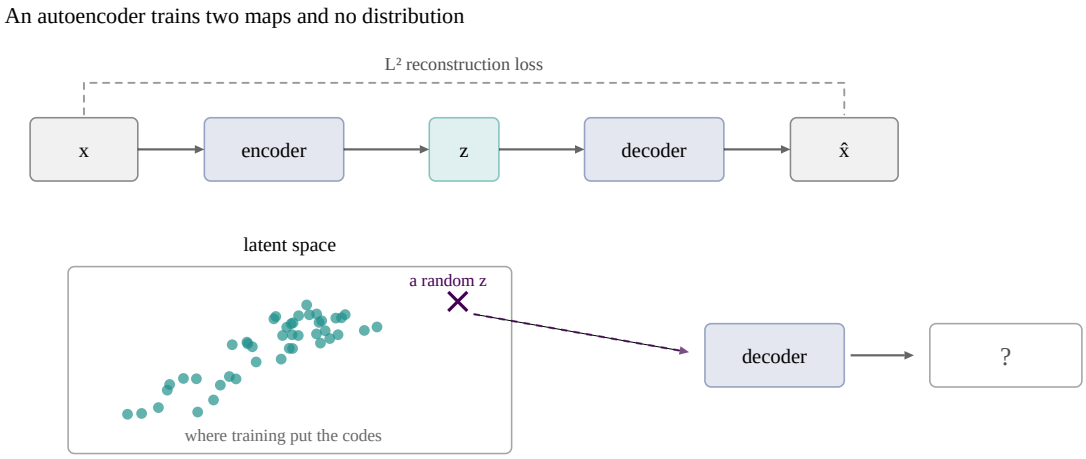


Figure 14.8.: What an autoencoder gives you and what it does not. Reconstruction constrains the encoder and decoder along the paths drawn solid, and the codes that training produces occupy some region of latent space whose shape nothing in the objective controls. Sampling requires knowing that region, and the loss never says a word about it: a vector drawn at random from the latent space is overwhelmingly likely to land where no training image ever mapped, and the decoder has no obligation to do anything sensible there.

Trained, an autoencoder can be used in the manner of the previous chapter: discard the decoder, keep the encoder, attach a head, fine-tune on whatever labels you have. It is a self-supervised method, and the reconstruction objective is a pretext task with no shortcut available.

To generate, you would want the opposite operation — discard the *encoder*, sample a z from somewhere, and decode it. This would be an implicit generative model of the kind Section 14.5 described: samples with no density value anywhere. And it does not work, for a reason that is easy to state and easy to underestimate. There is no distribution over z. Nothing in the reconstruction loss says anything about where in latent space the codes live, how they are spread, or what lies between them. Training maps each image somewhere and is indifferent to the geometry of the result, so a vector drawn from any distribution you care to name will almost certainly not resemble any code the encoder ever produced, and the decoder — never trained on such input — will return something incoherent.

Notice that this is not progress disguised as a problem. The original difficulty was that we did not know how to sample images. The autoencoder converts it into the difficulty of sampling latent codes, which is a smaller space but not a better-understood one. The can has been kicked down the road.

The variational autoencoder's idea is to stop hoping the latent space will turn out to have a convenient shape and instead **make** it have one. Add to the objective a term that pushes the codes towards a distribution chosen in advance — a unit Gaussian, in practice essentially always — and then sampling is trivial by construction: draw $z \sim \mathcal{N}(0, I)$, decode, done. The rest of this chapter is the work of making that idea into something trainable, and the work is more interesting than the idea.

14.10. One latent variable, two intractable integrals

Set up the generative story properly, as a claim about how the data came to exist. For each observed image $x^{(i)}$ there was a latent vector $z^{(i)}$ that came first, and the image was generated from it. The latent vector contains everything needed to produce the image — the identity of the objects, their number, their pose, the lighting — and it is *never observed.* There is no dataset of z's, and there never will be.

Two distributions define the model. The prior $p(z)$ is chosen, not learned, and is taken to be $\mathcal{N}(0, I)$. The decoder $p_\theta(x \mid z)$ is a neural network with weights θ. Together they specify a distribution over images, and if the $z^{(i)}$ were observed, fitting them would be ordinary maximum likelihood on pairs — precisely the conditional generative modelling of Section 14.2.

They are not observed, so the thing to maximise is the marginal, with z integrated out:

$$p_\theta(x) = \int p_\theta(x \mid z)\, p(z)\, dz. \tag{14.6}$$

Every symbol in Equation 14.6 is available. The decoder is a network we can evaluate; the prior is a Gaussian we can write down. And the expression is nonetheless useless, because the integral is over the whole latent space and the integrand runs through a neural network. There is no analytic solution and no reason to expect one. Nor does numerical integration rescue it: z has enough dimensions that quadrature is hopeless, and while the integral is an expectation under the prior that could in principle be estimated by sampling, the estimate has ruinous variance — the prior places almost none of its mass in the region where $p_\theta(x \mid z)$ is large for a given x, so nearly every sample contributes nothing. Putting a high-variance Monte Carlo estimate inside the inner loop of training is not a plan.

Try Bayes' rule instead, which is the other move available. Rearranging Equation 14.2 for this model,

$$p_\theta(x) = \frac{p_\theta(x \mid z)\, p(z)}{p_\theta(z \mid x)}, \tag{14.7}$$

which is an improvement in one respect — no integral — and a dead end in another. The numerator is fine: decoder and prior. The denominator is the **posterior** $p_\theta(z \mid x)$, the distribution over latent codes that could have produced a given image, and computing it requires exactly the integral from Equation 14.6 as its normalising constant. The two intractabilities are one intractability seen from two sides.

14.11. A second network for the posterior

The move that makes the variational autoencoder work is to give up on computing $p_\theta(z \mid x)$ and approximate it with another neural network. Introduce $q_\phi(z \mid x)$ — separate weights ϕ, trained jointly with θ — whose job is to output, for an image, a distribution over the latent codes that plausibly produced it. It approximates the posterior, and nothing forces the approximation to be good; the derivation in the next section will tell us exactly what the failure costs.

This gives the architecture its two halves and its name. The **decoder** $p_\theta(x \mid z)$ takes a code and returns a distribution over images. The **encoder** $q_\phi(z \mid x)$, also called the recognition model, takes an image and returns a distribution over codes. Compared with Section 14.9, both halves have been made probabilistic: each now emits a distribution rather than a point.

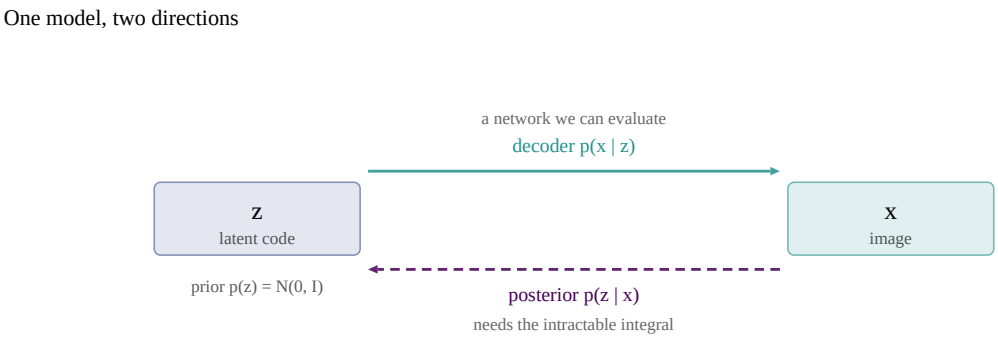


Figure 14.9.: The two directions, and which of them the model actually contains. The generative direction, drawn solid, is what the model defines: a prior over codes and a decoder that turns a code into a distribution over images. The inference direction, drawn dashed, is what Equation 14.7 needs and what no part of the model provides — recovering the code that produced an image requires inverting the decoder, and a neural network is not invertible. The whole of the next section is a response to the dashed arrow.

Which raises a mechanical question. A network outputs numbers, and a distribution is a function — how does the first produce the second? By fixing the *family* in advance and having the network output its parameters. Both halves here are Gaussian with diagonal covariance, so the encoder emits two vectors of the same length as z: a mean $\mu_{z|x}$ and a diagonal standard deviation $\sigma_{z|x}$, together specifying $\mathcal{N}(\mu_{z|x}, \mathrm{diag}(\sigma^2_{z|x}))$.

The diagonal restriction is not a matter of taste. A full covariance over a d-dimensional latent space needs $d(d+1)/2$ outputs and must be positive semi-definite, which is an awkward thing to guarantee from a network's raw output. Diagonal needs d outputs, is positive by construction if the network emits a log standard deviation, and — the reason that actually matters — makes the divergence against the prior available in closed form. The cost is that the encoder cannot express any correlation between latent dimensions. That cost buys something back in Section 14.14, where the independence turns out to be the mechanism behind the family's most attractive property.

The decoder is handled slightly differently, and the asymmetry is worth understanding because it explains why implementations look nothing like the equations. The decoder is also nominally a diagonal Gaussian, this time over pixels, with the network emitting the mean $\mu_{x|z}$ and the variance held fixed at some constant σ^2 rather than predicted. Fixing it is the right call. A predicted per-pixel variance with no correlation structure would say that each pixel independently wobbles by its own amount, and sampling from that distribution means taking the mean image and adding independent per-pixel noise — which is not a sensible model of how images vary, and produces speckle rather than plausible alternatives. So in practice the decoder's distribution is a fiction maintained for the derivation: at sampling time the mean is simply output, and nothing is ever drawn from it.

The fiction pays for itself immediately. With variance fixed, the log-likelihood of a Gaussian is

$$\log p_\theta(x \mid z) = -\frac{\|x - \mu_{x|z}\|^2}{2\sigma^2} + \text{const}, \tag{14.8}$$

so maximising it is minimising squared error between the decoder's output and the input, with σ^2 absorbed into a weight on the term. The probabilistic reconstruction objective *is* the L^2

loss of an ordinary autoencoder. The whole apparatus of distributions produces, at the point of implementation, the loss you would have written anyway.

14.12. The evidence lower bound

Now the derivation. It is one page of manipulation and it earns close reading, because the term that gets discarded at the end is what the entire method's approximation consists of.

Start from the objective — maximum likelihood, as always — and rewrite the marginal using Equation 14.7:

$$\log p_\theta(x) = \log \frac{p_\theta(x \mid z)\, p(z)}{p_\theta(z \mid x)}. \tag{14.9}$$

This holds for every z, since the right-hand side is secretly independent of z. Now multiply the fraction above and below by $q_\phi(z \mid x)$, which changes nothing, and split the logarithm into three terms in a particular arrangement:

$$\log p_\theta(x) = \underbrace{\log p_\theta(x \mid z)}_{\text{reconstruction}} - \underbrace{\log \frac{q_\phi(z \mid x)}{p(z)}}_{\text{prior}} + \underbrace{\log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}}_{\text{gap}}. \tag{14.10}$$

Still an exact equality, and still true for every z. Next comes the step that makes the derivation work, and it is entirely legitimate while looking like sleight of hand. The left-hand side does not depend on z. Anything that does not depend on z is unchanged by taking an expectation over z, under any distribution whatsoever — so take the expectation under $q_\phi(z \mid x)$, which is the distribution the encoder happens to produce. Expectation is linear, so it passes through to each of the three terms, and the second and third then match the definition of the Kullback–Leibler divergence exactly:

$$\log p_\theta(x) = \mathbb{E}_{z \sim q_\phi(z \mid x)}\big[\log p_\theta(x \mid z)\big] - D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, p(z)) + D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, p_\theta(z \mid x)). \tag{14.11}$$

Read the terms. The **first** is reconstruction, spelled out as a procedure: encode x to get a distribution over codes, draw a z from it, decode, and ask how much probability the decoder assigns to the original x. By Equation 14.8 this is squared reconstruction error. The **second** compares the distribution the encoder produces for this image against the prior we chose, and it is the term that shapes the latent space — the mechanism Section 14.9 said was missing, now written down. The **third** measures how far the encoder is from the true posterior, and it is the term we cannot compute, because $p_\theta(z \mid x)$ is exactly the object whose intractability started all of this.

So discard it. A KL divergence is non-negative whether or not you can evaluate it, so dropping the third term can only decrease the right-hand side:

$$\log p_\theta(x) \;\geq\; \mathbb{E}_{z \sim q_\phi(z \mid x)}\big[\log p_\theta(x \mid z)\big] - D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, p(z)) \;=\; \mathcal{L}(\theta, \phi; x). \tag{14.12}$$

$\mathcal{L}$ is the **evidence lower bound**, or ELBO, and it is what a variational autoencoder maximises. Both terms are computable: the first from a forward pass through encoder and decoder, the second in closed form because both distributions are diagonal Gaussians. Maximising a lower

bound does not maximise what it bounds, and this is the approximation the family is named for — this is why Figure 14.3 puts variational autoencoders under approximate rather than tractable density.

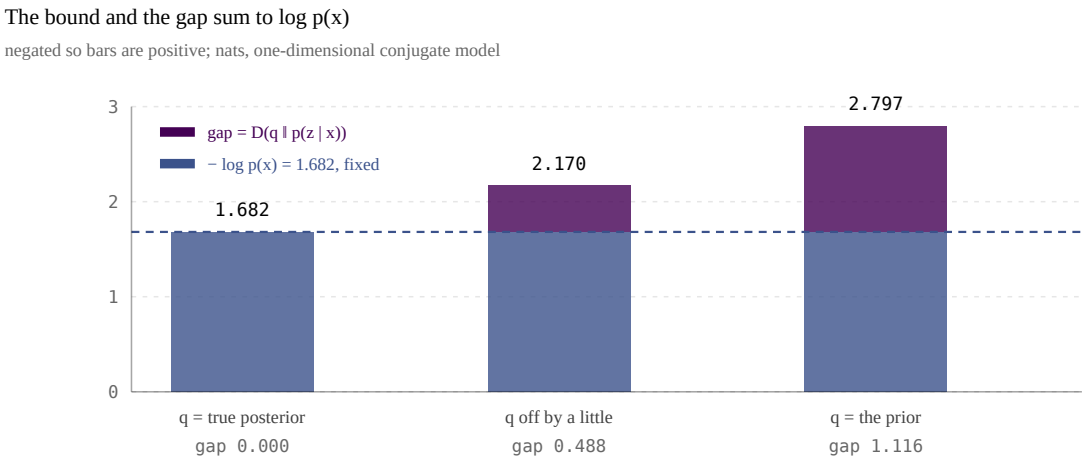


Figure 14.10.: The decomposition of Equation 14.11, evaluated in closed form. The generator uses a one-dimensional model in which the marginal, the true posterior, and all three terms have analytic expressions, so the identity can be checked rather than asserted: the bound plus the gap reproduces $\log p(x)$ exactly. The right-hand column shows what happens as the encoder's output is moved away from the true posterior — the bound falls and the gap opens by precisely the amount the bound lost, since their sum is a constant the encoder cannot affect.

The discarded term is worth one more sentence, because it is the most informative thing in the derivation. The gap between the bound and the true log-likelihood *is* $D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x))$ — not bounded by it, equal to it. So the bound is tight exactly when the encoder equals the true posterior, and it is loose in proportion to how wrong the encoder is. That reframes what the encoder is for. It looks like a piece of machinery for making training possible; it is also the thing that determines how much of the actual likelihood the objective is able to see, and every restriction placed on it — the diagonal covariance above, most obviously — puts a floor under the gap that no amount of training will lift.

14.13. Getting a gradient through a sample

One obstacle remains between Equation 14.12 and a training loop. The reconstruction term is an expectation over $z \sim q_\phi(z \mid x)$, estimated in practice by drawing a single sample. That sample depends on ϕ, since ϕ produced the distribution it came from — but sampling is not a differentiable operation, and backpropagation cannot pass a gradient through it. The encoder would receive no gradient from the reconstruction term at all, which is most of the objective.

The fix is to move the randomness out of the path. Rather than drawing z from $\mathcal{N}(\mu_{z|x}, \mathrm{diag}(\sigma^2_{z|x}))$, draw ϵ from a fixed standard normal and construct

$$z = \mu_{z|x} + \sigma_{z|x} \odot \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I). \tag{14.13}$$

The distribution of z is identical. What changes is the computation graph: ϵ is now an input, drawn from a distribution with no parameters, and z is a differentiable function of μ, σ and ϵ.

The gradient flows to μ and σ and onward into the encoder, while the stochasticity sits on a leaf where it cannot block anything.

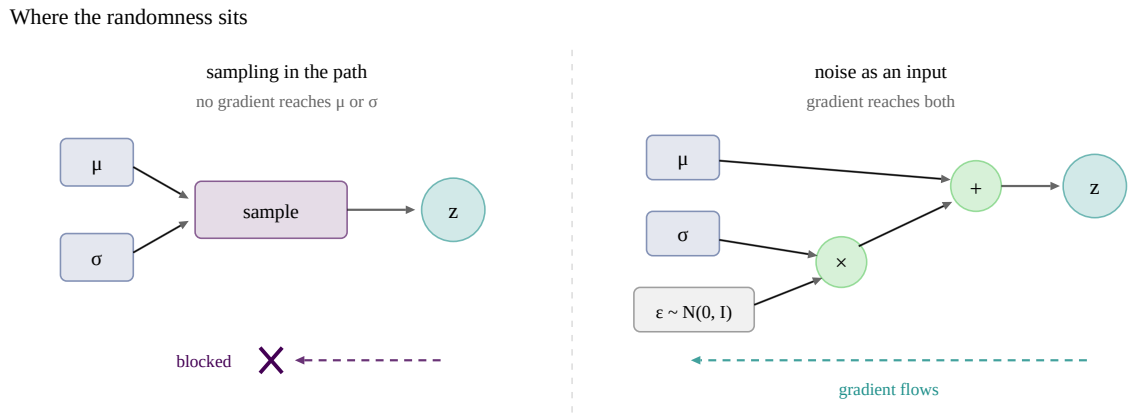


Figure 14.11.: Why Equation 14.13 is necessary. On the left the sampling operation sits between the encoder's outputs and the code, and the backward path stops there. On the right the same distribution is realised as a deterministic function of the encoder's outputs and an independent noise input, and the backward path reaches the encoder through the multiplication and the addition. The two forward passes are distributionally identical; only one of them can be trained.

This is the reparameterisation trick, and it is the technical contribution that made the whole framework practical. It was introduced by Kingma and Welling and, independently and simultaneously, by Rezende, Mohamed and Wierstra under the name stochastic backpropagation. The ELBO itself had been known in variational inference for years; what these papers supplied was the means to optimise it with the same gradient machinery as everything else, at which point the encoder and decoder could be arbitrary neural networks.

The training step is now unremarkable. Push x through the encoder to get $\mu_{z|x}$ and $\sigma_{z|x}$; evaluate the prior KL in closed form from those two vectors; draw ϵ and form z by Equation 14.13; push z through the decoder; compute squared error against x; add the two terms and take a gradient step in θ and ϕ together. Every part of that is a standard operation, which is the point of the preceding four sections.

14.14. Two terms that want opposite things

The most interesting property of Equation 14.12 is that its two terms disagree, and the disagreement is not a defect to be tuned away — it is the mechanism.

Consider what each term would do if it were alone. The reconstruction term wants $\sigma_{z|x} \to 0$ and a distinct $\mu_{z|x}$ for every image. With zero variance the code is deterministic, the sampling step does nothing, and the model is free to assign each training image its own private point in latent space and reconstruct it perfectly. Reconstruction alone turns a variational autoencoder back into the ordinary autoencoder of Section 14.9, with all of that section's problems.

The prior term wants the opposite: $\mu_{z|x} \to 0$ and $\sigma_{z|x} \to 1$ for every image, because that is when $q_\phi(z \mid x)$ matches $\mathcal{N}(0, I)$ and the divergence vanishes. Prior alone produces an encoder that ignores its input entirely and a latent space that is perfectly shaped and carries no information.

Neither extreme is what you want, and the useful model is wherever they balance. It is worth doing the balance exactly, because in one dimension it can be done in closed form and the answer is more informative than the qualitative story. Take $p(z) = \mathcal{N}(0, 1)$ and a decoder $p(x \mid z) = \mathcal{N}(z, \sigma_x^2)$ — a decoder that passes the code through and adds noise of scale σ_x, which stands in for how much of the image the code fails to determine. Maximising Equation 14.12 over μ_q and σ_q then gives

$$\mu_q = \frac{x}{1 + \sigma_x^2}, \qquad \sigma_q^2 = \frac{\sigma_x^2}{1 + \sigma_x^2}, \tag{14.14}$$

which are exactly the mean and variance of the true posterior $p(z \mid x)$. The two terms fighting to a draw land on the right answer, the gap in Equation 14.11 closes to zero, and the bound becomes tight — the derivation and the tension are the same fact viewed twice.

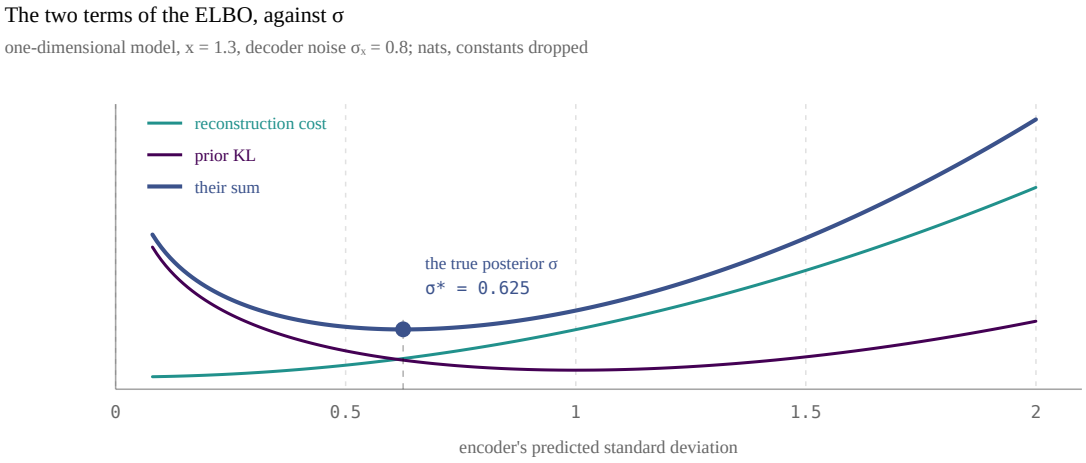


Figure 14.12.: The two terms of the ELBO as functions of the encoder's predicted standard deviation, for the one-dimensional model above. Reconstruction cost falls monotonically as the code becomes more deterministic; the prior term has its minimum at $\sigma_q = 1$ and rises steeply as $\sigma_q \to 0$, because the logarithm in the Gaussian KL diverges there. Their sum has an interior minimum at the value Equation 14.14 predicts, marked, and the generator solves for it numerically as a check on the closed form.

Equation 14.14 also explains the family's characteristic failure without any additional argument. Let σ_x grow — a decoder whose output the code barely constrains, either because the decoder is weak or because it is powerful enough to reconstruct the data without help. Then $\mu_q \to 0$ and $\sigma_q^2 \to 1$: the optimal encoder output is the prior, identical for every input. This is **posterior collapse**, and it is not a bug in the optimiser. It is the correct solution to the objective when the reconstruction term has nothing to gain from the code. The standard mitigations — reweighting the prior term, annealing it from zero, restricting the decoder's capacity — all amount to shifting where Equation 14.14 lands.

With training done, sampling is finally the trivial operation promised in Section 14.9: draw $z \sim \mathcal{N}(0, I)$, run the decoder, output its mean. This works now, where it failed for the plain autoencoder, because the prior term spent all of training pushing the encoder's outputs towards the prior, so the region of latent space the decoder was trained on is approximately the region you are now sampling from.

There is one further consequence of the diagonal covariance, and it is the property that keeps the architecture in use. Because the latent dimensions are modelled as independent, they can be varied independently and often turn out to mean something separately: hold every coordinate of z fixed but one, sweep that one, decode at each step, and the output frequently changes along a single interpretable axis — a digit's slant, a face's angle. This is **disentanglement**, and it comes free in the sense that nobody asked for it, which is also the sense in which it is unreliable. It appears on simple data and degrades on complex data; β-VAE obtains more of it by upweighting the prior term, at a cost in reconstruction quality that is exactly the trade Figure 14.12 describes. The honest position is that variational autoencoders often produce partly disentangled latent spaces and that no one can currently guarantee it.

14.15. What each family bought

The two model families in this chapter differ in every dimension except the objective they are trying to maximise, and the differences are consequences of two design choices — the chain rule versus a latent variable — rather than of architecture.

	Autoregressive	Variational autoencoder
Density $p(x)$	exact, one forward pass	lower bound only
Sampling	T sequential passes	one pass
Latent code	none	explicit, low-dimensional
Sample quality	sharp, locally coherent	blurry

Both are trained by maximum likelihood, so both inherit the mode-covering bias of Figure 14.4, and the last row is where that bias shows. Autoregressive models get away with it because each factor is a distribution over 256 discrete values and a sharp softmax costs nothing; the mode-covering happens at the level of individual subpixels, where it is cheap. A variational autoencoder places a Gaussian over the whole image and reconstructs through a squeezed code, so covering the modes means averaging over them, and the average of several plausible images is a blurry one.

That leaves the field with an unsatisfying pair of options: exact likelihoods and sharp samples at a sampling cost that scales with the number of pixels, or fast sampling and a latent space at the price of blur. The two families that fill the other half of Figure 14.3 are attempts at the missing corner. Generative adversarial networks abandon likelihood entirely, which buys sharpness and costs the mode coverage — samples that are individually convincing and collectively miss parts of the data. Diffusion models keep a likelihood-based objective and recover sharpness by giving up the single-pass sample, taking many steps instead, and they are what the next chapter is about.

15. Generative Models II

Lecture 14

Based on Lecture 14 of CS231n, Stanford University, Spring 2025.

15.1. A model defined only by its sampler

The previous chapter closed on a dilemma. An autoregressive model gives an exact likelihood and sharp samples but needs one network pass per subpixel, so producing a megapixel image is three million sequential forward passes. A variational autoencoder samples in a single pass and hands you a latent code, but it is trained through a Gaussian reconstruction term that rewards hedging, and hedging over images means averaging them, which is what blur is. Both failures trace back to the same commitment: both models are trying to put a number on $p(x)$, and the number is what costs them.

Generative adversarial networks abandon the number. There is no expression for $p_\theta(x)$ anywhere in a GAN, no bound on it, and no way to ask the trained model how probable a particular image is. What the model provides is a procedure for producing samples, and nothing else. This is the position at the far right of the taxonomy in the previous chapter: an *implicit* density model, one that defines a distribution the way a physical process does, by generating from it rather than by describing it.

The mechanics are the same as a variational autoencoder's decoder, stripped of everything else. Fix a prior $p(z)$ that you chose yourself and can sample from — a unit Gaussian, in practice always. Sample $z \sim p(z)$, push it through a network, and call the output an image:

$$x = G(z), \qquad z \sim p(z). \tag{15.1}$$

The network G is deterministic; all the randomness enters through z. Because G is a fixed function, pushing the prior through it induces a distribution over images, written p_G, whose density nobody can write down — it would require inverting G and tracking how the map compresses and stretches volume, and G is neither invertible nor volume-preserving. That inability is not an inconvenience to be worked around later. It is the whole design. The generator is free to concentrate its output onto a thin, curved, disconnected region of image space, and it pays no penalty for doing so, because no term in its objective ever evaluates a density.

The goal is stated distributionally: make p_G match p_{data}. If they match, then sampling z and running one forward pass gives a genuine sample from the data distribution, at the cost of exactly one network evaluation. What is missing is any way to measure whether they match, since the whole point was that neither density is available. A maximum-likelihood objective is out of reach by construction. Something else has to supply the training signal.

15.2. The discriminator is the loss function

The answer is to stop writing the objective down and train a second network to be it. A **discriminator** D takes an image and returns the probability that the image came from the dataset rather than from the generator, so $D(x) = 1$ means *real* and $D(x) = 0$ means *fake*. It is an ordinary binary classifier, trained on an ordinary cross-entropy loss, over a dataset half of which is the real training set and half of which is whatever the generator is currently producing.

The two networks are trained against each other on a single scalar quantity:

$$\min_G \max_D \; V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}}\big[\log D(x)\big] + \mathbb{E}_{z \sim p(z)}\big[\log\big(1 - D(G(z))\big)\big]. \tag{15.2}$$

Read Equation 15.2 twice, once from each side, because it means different things depending on which network you hold fixed. Hold G fixed and it is the discriminator's problem. The first term is large when $D(x)$ is near one on real images; the second is large when $D(G(z))$ is near zero on generated ones. Maximising their sum is exactly binary cross-entropy with the labels *real* and *fake*, written without the usual minus sign. Now hold D fixed and read it as the generator's problem. The first term contains no G and is a constant to it. The second term is small when $D(G(z))$ is near one, so minimising drives the generator towards images the current discriminator calls real.

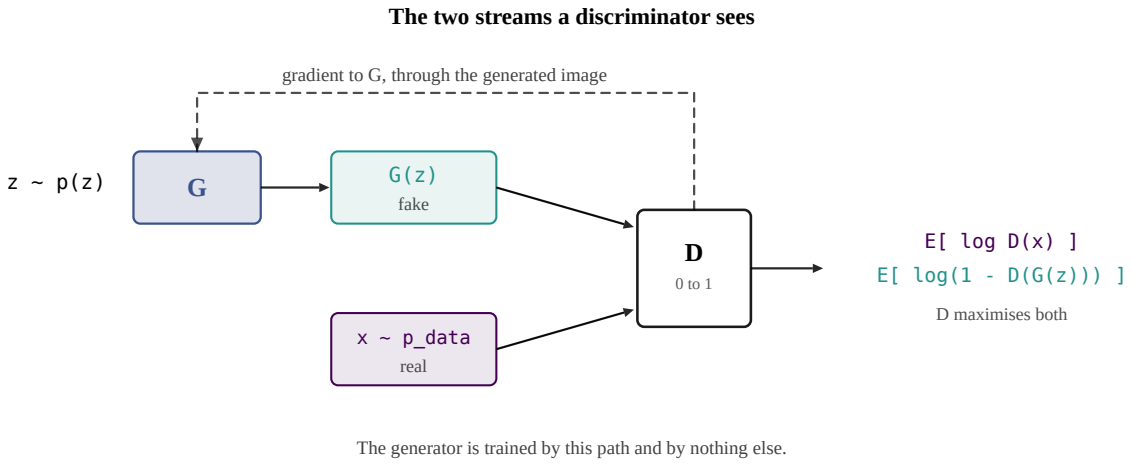


Figure 15.1.: The game and the two gradient paths. The discriminator sees both streams and is trained on the ordinary two-class objective; the generator sees only the fake stream and only through the discriminator. The dashed path is the whole of the generator's training signal: there is no reconstruction term, no likelihood, nothing anchoring the generator to any particular training image. Backpropagation runs from the discriminator's output through the generated image and into the generator's weights, which is why D must be differentiable and why the game is trained end to end.

Training alternates. Take an ascent step on D to sharpen the current loss, then a descent step on G against that sharpened loss, and repeat. The intuition is an arms race: as the discriminator gets better at spotting the artefacts of generated images, the only way for the generator to keep scoring is to stop producing them.

The most important structural fact about Equation 15.2 is easy to skip past. V is not a loss. Its value at any moment is a joint property of two networks and says nothing about the quantity anyone cares about, which is how close p_G has come to p_{data}. A weak generator against a weak discriminator and a strong generator against a strong discriminator produce the same numbers. Every other model in this course comes with a curve that goes down when things are going well; a GAN does not, and the practical consequence is that the standard method of debugging a training run — look at the loss — is unavailable. You look at samples, by eye, and that is most of what you have.

15.3. The gradient the obvious objective does not deliver

Consider the beginning of training, when both networks are random. The generator emits noise, and telling noise apart from photographs is trivial, so within a few hundred iterations the discriminator is confident and $D(G(z))$ sits close to zero on everything the generator makes. This is precisely the regime in which the generator most needs a gradient, and it is precisely the regime in which Equation 15.2 stops providing one.

To see why, the derivative has to be taken with respect to the right variable. The discriminator's final layer is a sigmoid, $D = \sigma(s)$, and the generator influences D only through the pre-sigmoid score s, so the quantity that determines how much signal reaches the generator's weights is the derivative of its loss with respect to s. For the objective as written, the generator minimises $\log(1 - \sigma(s))$, and

$$\frac{\partial}{\partial s} \log\big(1 - \sigma(s)\big) = -\sigma(s) = -D. \tag{15.3}$$

The gradient is proportional to D itself. When the discriminator is confident and $D \approx 0$, the gradient is approximately zero: the generator receives almost nothing exactly when it is worst and has the most to learn. The loss curve is flat in the region where training starts, and the game stalls before it begins.

The fix is a change of objective for the generator alone. Instead of minimising $\log(1 - D(G(z)))$, minimise $-\log D(G(z))$. Both are minimised by driving $D(G(z))$ to one, so they express the same wish, but they express it with different curvature:

$$\frac{\partial}{\partial s}\big[-\log \sigma(s)\big] = -\big(1 - \sigma(s)\big) = -(1 - D). \tag{15.4}$$

Now the gradient is proportional to $1 - D$, and it is *largest* when the discriminator is most confident. The two expressions are mirror images, and at $D = 0.01$ the ratio between them is 99: the modified objective delivers ninety-nine times the gradient in the regime where training actually begins. This is the **non-saturating** generator loss, and it is not an optional refinement. A GAN trained from scratch on Equation 15.2 as literally written usually does not train at all.

Note what has been given up to get this. The two networks are no longer optimising a single shared quantity — the discriminator maximises Equation 15.2 while the generator minimises something else — so the tidy minimax formulation is already a description of a game nobody plays. Whatever guarantees attach to Equation 15.2 attach to a training procedure that is not the one in use.

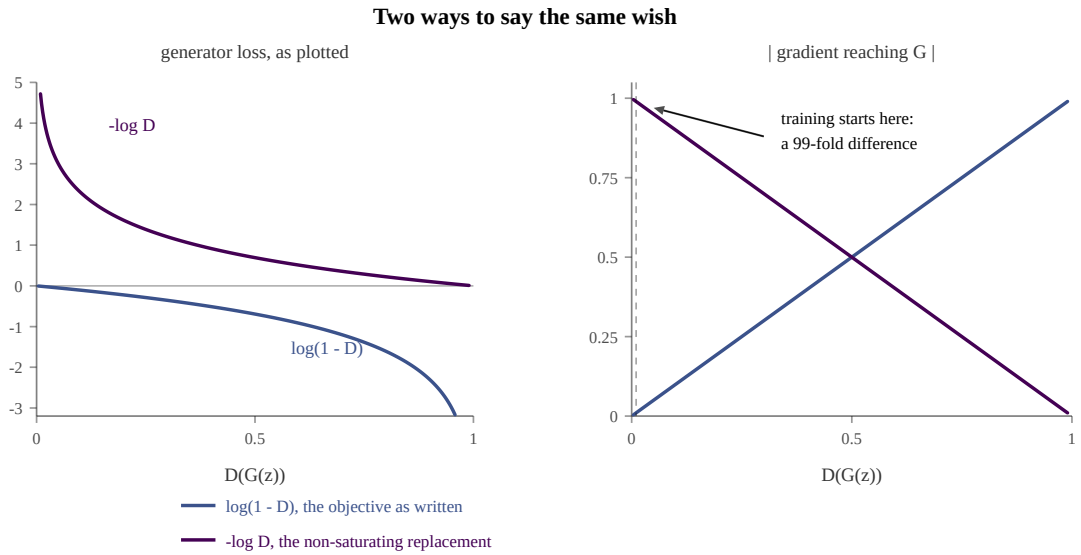


Figure 15.2.: The two generator objectives and what they deliver. The left panel plots each loss against the discriminator's current output on generated images; both fall as $D(G(z))$ rises, so both encode the same preference. The right panel plots the magnitude of each loss's gradient with respect to the discriminator's pre-sigmoid score, which is the signal that actually reaches the generator. Training starts at the left edge, where the confident discriminator puts $D(G(z))$ near zero, and there the two curves differ by two orders of magnitude. The saturating loss is flat where the generator is worst; the non-saturating one is steepest there.

15.4. What the game would converge to

There is a reason to believe Equation 15.2 is the right game and not an arbitrary one, and it comes from solving the inner maximisation exactly. For a fixed generator, the discriminator's problem decouples across images: at each x it is choosing a single number to maximise $p_{\text{data}}(x)\log D(x) + p_G(x)\log(1-D(x))$, and differentiating that scalar expression gives the optimum immediately,

$$D_G^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}. \tag{15.5}$$

The optimal discriminator does not learn features of real images. It reports a ratio of densities — one half wherever the two distributions agree, near one where only real data lives, near zero where only generated data lives. It is also, as the lecture notes, a formula you can write and never evaluate: computing it needs p_{data}, and if you had p_{data} you would not be training a generative model.

Substituting Equation 15.5 back into Equation 15.2 turns the game's value into a divergence. Adding and subtracting $\log 2$ inside each expectation converts the two terms into Kullback–Leibler divergences against the mixture $m = \frac{1}{2}(p_{\text{data}} + p_G)$, and what is left is

$$V(G, D_G^*) = 2\, D_{\text{JS}}(p_{\text{data}} \,\|\, p_G) - \log 4, \tag{15.6}$$

where $D_{\mathrm{JS}}(p\|q) = \frac{1}{2}D_{\mathrm{KL}}(p\|m) + \frac{1}{2}D_{\mathrm{KL}}(q\|m)$ is the Jensen–Shannon divergence. It is non-negative and zero only when its arguments are equal, so the outer minimisation is solved uniquely at $p_G = p_{\mathrm{data}}$, and the value there is $-\log 4 \approx -1.386$.

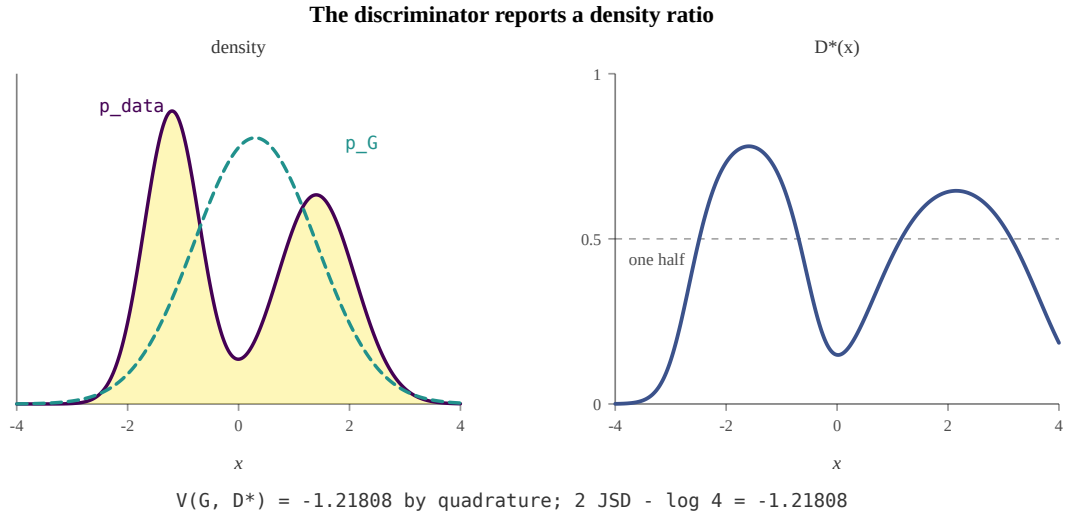


Figure 15.3.: The optimal discriminator and the identity it satisfies, on a one-dimensional problem where every quantity is closed form. The left panel shows a bimodal data density and a Gaussian generator distribution; the right shows D^* from Equation 15.5, which crosses one half exactly where the densities cross and saturates wherever one distribution has the field to itself. The generator's script evaluates both sides of Equation 15.6 by numerical integration and asserts they agree, then sweeps the generator's mean to confirm the value bottoms out at $-\log 4$ when the two distributions are made identical.

This result is worth understanding for what it does not say. It assumes both networks have unlimited capacity, so that the discriminator can actually reach Equation 15.5 and the generator can represent p_{data} exactly; real networks are finite and can do neither. It assumes the inner maximisation is solved before every generator step, whereas practice takes one discriminator step per generator step and the discriminator trails behind a moving target. It says nothing about convergence — alternating gradient steps on a saddle point are not guaranteed to find it, and frequently orbit it instead. And, as the previous section established, the generator is minimising a different objective than the one the theorem is about. What survives is real but modest: the game has a sensible unique optimum, which is more than can be said for an arbitrary adversarial loss.

15.5. What GANs turned out to give, and what they cost

Both networks are convolutional in every implementation that mattered; GANs fell out of use before vision transformers arrived. The first architecture to produce non-trivial samples was DC-GAN, a five-layer fully convolutional generator with no pooling and no fully connected layers, and the family peaked with StyleGAN, which injects the latent code not at the input but at every layer through adaptive instance normalisation. Each layer normalises its activations and then rescales them with a per-channel scale w_i and shift b_i predicted from the latent:

$$\mathrm{AdaIN}(x, w, b)_i = w_i \frac{x_i - \mu(x)}{\sigma(x)} + b_i. \tag{15.7}$$

Feeding the latent in at every resolution rather than only at the bottom is what gives StyleGAN its control over coarse and fine attributes separately, and its samples were, for several years, the best images any generative model produced.

The property that made GANs interesting beyond sample quality is that their latent space turned out to be smooth. Draw two codes z_0 and z_1, interpolate between them, and decode along the path: the images morph continuously, passing through intermediate states that are themselves plausible images rather than superpositions of the endpoints. Nothing in Equation 15.2 asks for this. The lecture makes the point sharply with a counterexample — a generator that ignored z entirely and memorised ten training images would fool a discriminator perfectly well, placing all its mass on ten spikes and leaving the latent space meaningless. Smoothness is an empirical consequence of training a continuous network under pressure to cover the data, not a guarantee.

The costs are three, and they compound. The first is that there is no encoder. The generator maps $z \to x$ and nothing maps back; a GAN cannot tell you the code for a given image except by numerical inversion after the fact. The discriminator enforces only that the *distribution* of outputs matches the *distribution* of data, with no correspondence between any particular z and any particular training image, which is a strictly weaker relationship than a variational autoencoder's.

The second is **mode collapse**, the failure the memorisation counterexample gestures at. Because no term in the objective is a density, nothing penalises the generator for ignoring whole regions of the data. A generator that produces only one convincing kind of image is doing well by its own loss, and collapse onto a narrow set of outputs is the characteristic GAN pathology — the reverse-KL, mode-seeking behaviour of the previous chapter's Figure 14.4, arrived at by a different route.

The third is that the first two are invisible. With no meaningful loss curve, a run that has collapsed, diverged, or filled with NaNs looks from the outside like a run that is working. Between roughly 2016 and 2021 GANs were the default choice for image generation and attracted thousands of papers, a large fraction of them attempts to stabilise training or to make the curves mean something; the lecturer's assessment of that literature is that nothing stuck. What ended the era was not a fix but a replacement.

15.6. Trading one hard step for many easy ones

Set the adversarial machinery aside and ask why the single-pass generator was hard to train in the first place. It is being asked to do something genuinely difficult: convert an unstructured Gaussian vector into a photograph in one shot, with the entire global structure of the image — layout, lighting, object identity, texture — decided in a single forward pass. A variational autoencoder attempts the same jump with a tractable loss and produces blur. A GAN attempts it with a learned loss and produces sharp images at the price of a training procedure nobody could control.

Diffusion models keep the tractable loss and remove the jump. The generator's job is broken into many steps, each one small enough that a plain regression objective can supervise it, and the difficulty is moved out of the loss function and into the number of times the network is run.

The construction begins with a constraint that looks arbitrary and is load-bearing: the noise has the same shape as the data. Where a GAN's latent might be a 512-vector standing behind a $256 \times 256 \times 3$ image, a diffusion model's z is itself $256 \times 256 \times 3$. Noise and data now live in the same space, so it is meaningful to speak of a point that is part-way between them.

That is exactly what the model is built on. Introduce a noise level $t \in [0, 1]$ and define a family of corrupted images x_t interpolating between clean data at $t = 0$ and pure noise at $t = 1$. Then train a single network $f_\theta(x_t, t)$ to remove a little of the noise, and at sampling time start from $x_1 \sim p_{\text{noise}}$ and apply it repeatedly. Every individual application is a small, well-posed regression problem with an unambiguous target; the hard global work of turning noise into an image is done by the composition of a hundred of them, and no single one of them ever has to be clever.

The literature on this idea is notoriously unapproachable — three mathematically distinct formalisms, no shared notation, and papers that spend five pages on derivations before saying what is computed. This chapter follows the modern, geometric formulation, **rectified flow**, which is what recent large image and video models actually implement, and returns in Section 15.14 to say how the other formalisms describe the same object.

15.7. Straight lines between two distributions

Rectified flow makes the vague word *corrupt* concrete in the simplest way available: a straight line. Sample a data point $x \sim p_{\text{data}}$, which in practice means drawing an image from the training set, and independently sample a noise point $z \sim p_{\text{noise}}$ of the same shape. Sample a time $t \sim \text{Uniform}(0, 1)$. Then define the noisy sample and the velocity that connects the two endpoints:

$$x_t = (1 - t)\, x + t\, z, \qquad v = z - x. \tag{15.8}$$

At $t = 0$ this is the clean image, at $t = 1$ it is pure noise, and in between it is a linear blend. The vector v is the constant velocity of a particle travelling that line in unit time: x_t satisfies $\mathrm{d}x_t/\mathrm{d}t = v$ exactly, at every t along the segment. Each training pair therefore comes with a complete, exactly known trajectory, which is the entire trick. There is no unknown target to be approximated — the target is a subtraction.

The network is trained to predict it. Feed it the noisy sample and the noise level, and regress on the velocity:

$$L(\theta) = \big\| f_\theta(x_t, t) - v \big\|_2^2. \tag{15.9}$$

That is the whole objective. There is no adversary, no bound, no variational term, no reparameterisation trick — a mean squared error against a target computed by subtracting two things you already sampled. The training loop is four lines: draw x from the data and z from a Gaussian, draw t uniformly, form x_t by Equation 15.8, and take a gradient step on Equation 15.9. And unlike Equation 15.2, it is a loss in the ordinary sense: it goes down when the model gets better, it can be plotted, and a run that has gone wrong looks wrong.

The noise level is passed to the network as an input rather than being inferred from the image, and one network serves every value of t. This is a genuine saving: a separate denoiser per noise level would be a hundred networks, and the noise levels are similar enough that sharing weights across them is a form of multi-task training that helps rather than hurts.

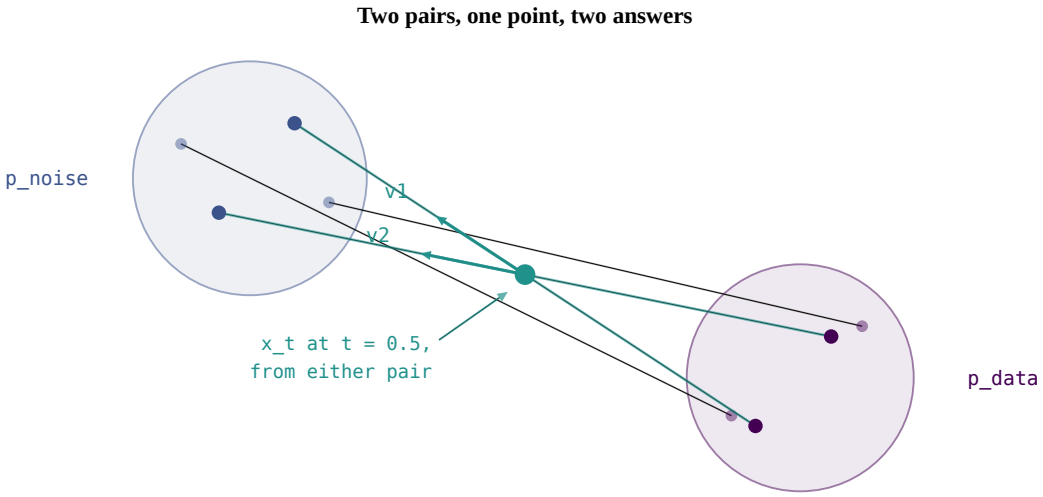


Figure 15.4.: The construction Equation 15.8 defines, on a two-dimensional cartoon where the geometry is visible. Each training example is an independently drawn pair — one point from the noise distribution, one from the data distribution — joined by a straight segment; the time t selects a point along it, and the velocity v is the same vector everywhere on that segment. Several such pairs are drawn to make the essential difficulty visible: the segments cross. Two different pairs can put their interpolants at nearly the same place with velocities pointing in different directions, so the regression target at a given (x_t, t) is not a single well-defined vector. What the network can learn there is the subject of the next section.

15.8. Sampling is an ODE solve

At training time both endpoints are known. At sampling time only the noise end is: draw $x_1 \sim p_{\text{noise}}$ and there is no x to subtract, so there is no line to follow. All the model provides at any point is a predicted velocity, and a velocity field is an instruction to integrate.

That is exactly what the sampler does. Fix a number of steps T, start at pure noise, and march t from 1 down to 0 in equal decrements, at each step evaluating the network and moving along its prediction:

$$x \leftarrow x - \frac{1}{T} f_\theta(x, t), \qquad t = 1,\ 1 - \tfrac{1}{T},\ \ldots,\ \tfrac{1}{T}. \tag{15.10}$$

This is the forward Euler method applied to the ordinary differential equation $\mathrm{d}x/\mathrm{d}t = f_\theta(x, t)$, integrated backwards in time; the minus sign is the negative time step, and $1/T$ is its magnitude. Sampling a diffusion model is numerical integration of a learned vector field, and every question about samplers — how many steps, what step sizes, which solver — is a question about numerical integration that the field of numerical analysis had already answered before anyone applied it here.

The step size is why T cannot be one. If the network's prediction were constant along the trajectory, a single step of size 1 would land exactly on a data point and diffusion would collapse into a GAN. It is not constant: the predicted velocity is a function of position, and a large step accumulates the error of pretending otherwise. A commonly used value is $T = 50$, and $T = 30$ is often tolerable — meaning that a diffusion model costs thirty to fifty network evaluations per sample against a GAN's one. That is the price of the stable objective, and Section 15.13 is about trying to get it back.

There is a check worth doing here, because the claim that this procedure produces samples from p_{data} is not obvious and the lecture asserts it without argument. Take $p_{\text{noise}} = \mathcal{N}(0, 1)$ and $p_{\text{data}} = \mathcal{N}(\mu, \sigma^2)$ in one dimension. Everything is then jointly Gaussian, and the velocity field a perfectly trained network would learn — the conditional expectation $u(x_t, t) = \mathbb{E}[v \mid x_t]$, derived in the next section — is available in closed form:

$$u(x_t, t) = -\mu + \frac{t - (1-t)\sigma^2}{(1-t)^2\sigma^2 + t^2}\,(x_t - (1-t)\mu). \tag{15.11}$$

Because this is linear in x_t, the ODE it defines maps Gaussians to Gaussians, and the mean and variance of the solution can be propagated by integrating two scalar equations. The marginal law of x_t under Equation 15.8 is separately known: it is $\mathcal{N}\big((1-t)\mu,\ (1-t)^2\sigma^2 + t^2\big)$. These are two independent routes to the same quantity, and the chapter's figure script integrates the first and compares it against the second at every t, agreeing to nine decimal places and landing on $\mathcal{N}(\mu, \sigma^2)$ at $t = 0$. The flow really does transport one distribution onto the other.

15.9. What the network can actually predict

Figure 15.4 left a problem open, and it is the one that determines everything the model can and cannot do. The interpolant x_t does not remember where it came from. A given noisy image is reachable from many different (x, z) pairs, and those pairs carry different velocities, so the training data contains the same input paired with contradictory targets. The network cannot

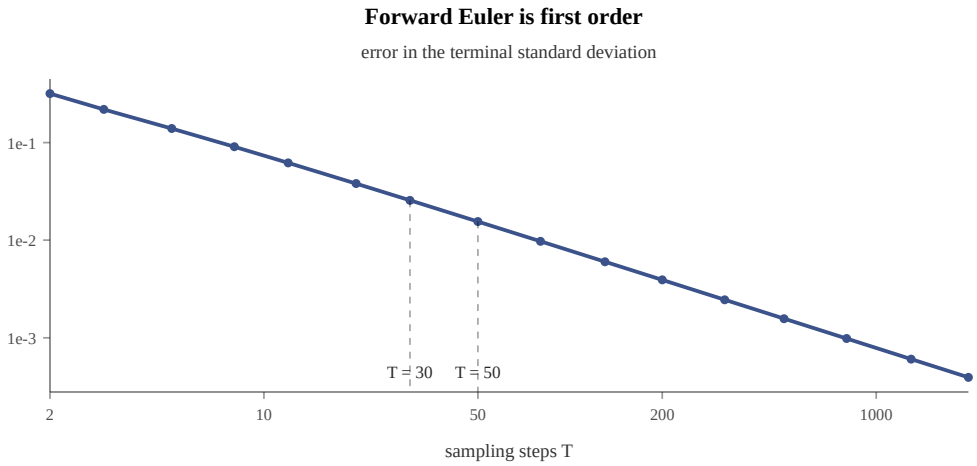


Figure 15.5.: Discretisation error against the number of sampling steps, for the Gaussian problem of Equation 15.11 where the exact answer is known. Both axes are logarithmic, and the measured error is the discrepancy between the terminal standard deviation produced by Equation 15.10 and the true σ. The slope is -1: forward Euler is a first-order method, so halving the step size halves the error, and there is no step count at which the error vanishes. The two step counts the lecture names are marked. This is why a single-step sampler is not merely inaccurate but structurally wrong — at $T = 1$ the method assumes the velocity field is constant over the entire journey from noise to data, which is precisely the assumption that fails.

satisfy them all, and squared error decides how it fails: the minimiser of Equation 15.9 is the conditional mean of the target given the input,

$$f^*(x_t, t) = \mathbb{E}\big[\, z - x \;\big|\; x_t \,\big], \tag{15.12}$$

an average over every clean image and every noise sample that could have produced this particular x_t. The network is not learning to denoise a specific image. It is learning, at each point of image space and each noise level, the average direction of all the trajectories passing through.

The consequences are cleanest at the two ends. At $t = 0$ there is no ambiguity at all — x_t *is* the clean image — so the expectation collapses to $\mathbb{E}[z] - x$, the vector from the data point to the mean of the noise distribution, and the problem is trivial. At $t = 1$ the ambiguity is total, since x_t is pure noise and carries no information about which image it will become, so the expectation is $z - \mathbb{E}[x]$: the model can do nothing but point away from the mean image of the entire dataset. Both extremes are easy for the same reason, that neither requires the network to know anything about the data beyond one global statistic.

Everything interesting happens in between, and it is worth being concrete about what the intermediate prediction looks like, because it explains the characteristic behaviour of these models. Rearranging Equation 15.8 gives the clean sample implied by a velocity, $\hat{x} = x_t - t\, f_\theta(x_t, t)$, so the network's output at any noise level can be read as a guess at the final image. Under Equation 15.12 that guess is $\mathbb{E}[x \mid x_t]$ — an average of every image consistent with the current noisy state. At high noise that average is over almost the whole dataset and is correspondingly featureless; as t falls the set of consistent images narrows and the average sharpens. This is why the intermediate states of a sampling run look the way they do, blurry and global early and

detailed late, and it is the same mode-covering blur that ruined the variational autoencoder in the previous chapter. The difference is that here it appears only inside a single step, and the next step is taken from a point where the ambiguity is smaller.

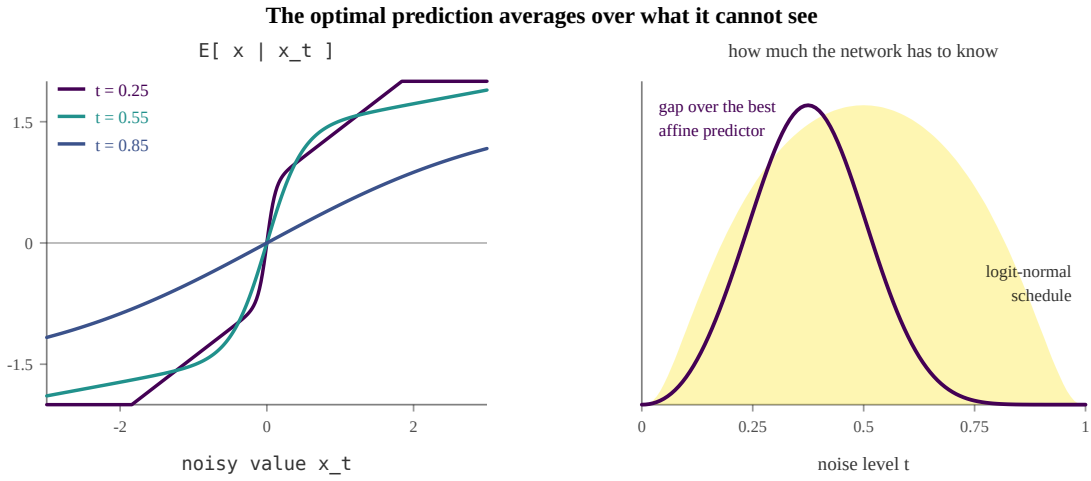


Figure 15.6.: What ambiguity costs, on a one-dimensional two-mode dataset where every quantity is closed form. The left panel plots the implied clean prediction $\mathbb{E}[x \mid x_t]$ against the noisy value, at three noise levels: at low noise the curve is a sharp step that commits to whichever mode is nearer, and at high noise it is nearly flat at the dataset mean, meaning the network is producing an average of both modes rather than either one. The right panel measures difficulty as the amount by which the optimal predictor beats the best *affine* predictor of the same target — that is, how much the network must know about the data beyond its first two moments. This is not the irreducible loss, which stays positive at both ends because the target still contains unobserved noise there; it is the quantity that genuinely vanishes at $t = 0$ and $t = 1$, since the conditional mean is affine at both. The generator asserts both endpoint values are zero. The peak sits nearer $t = 0.3$ than the exact centre, so the logit-normal schedule drawn over it is a rough match to the difficulty rather than a derived one. Both curves are rescaled to their own maximum, because only their shapes are being compared.

Sampling $t \sim \text{Uniform}(0, 1)$ therefore spends equal training effort on noise levels that differ enormously in difficulty. The standard correction is to sample t from a distribution concentrated in the middle, most commonly by drawing a Gaussian variable and squashing it through a sigmoid — **logit-normal** sampling, used in Stable Diffusion 3 among others. Nothing about the model changes; only the frequency with which each noise level appears in a batch.

The schedule is also where resolution enters, and this is the point at which the clean formulation stops being resolution-agnostic. Adding Gaussian noise of a fixed magnitude destroys much less information in a high-resolution image than in a low-resolution one, because neighbouring pixels in a high-resolution image are strongly correlated and the redundancy survives corruption that would erase an equivalent low-resolution image. A noise level that makes an image genuinely uninformative at 64×64 leaves a 1024×1024 image still legible, so the schedule has to be **shifted** towards higher noise as resolution grows. A diffusion model trained at one resolution does not transfer to another by changing the input size alone, and the beautiful formulation of Section 15.7 turns out to need a resolution-dependent knob before it works on large images.

15.10. Conditioning, and buying obedience with a second forward pass

Unconditional generation is a curiosity. What people want from these models is control: an image of a specific thing, described in words. Rectified flow accommodates conditioning with almost no machinery — the dataset becomes pairs (x, y), the network takes y as an extra input, and both Equation 15.9 and Equation 15.10 are otherwise unchanged. Geometrically the model is now learning a velocity field that points towards $p_{\text{data}}(x \mid y)$, a sub-region of the data distribution, rather than towards all of it.

Trained this way, the model tends to underuse the condition. It learns to produce plausible images and treats the prompt as a suggestion, which is a rational response to an objective in which ignoring y costs only a little loss on most examples. There is no term anywhere in Equation 15.9 that specifically rewards agreement with the condition, so obedience has to be bought at sampling time.

Classifier-free guidance buys it with a change to training that takes one line. On each iteration, flip a coin; if it comes up heads — with probability around one half — replace y with a designated null token before feeding it to the network. The same set of weights now learns two velocity fields at once: $v^{\emptyset} = f_\theta(x_t, y_\emptyset, t)$, which is the unconditional field pointing towards $p_{\text{data}}(x)$, and $v^{y} = f_\theta(x_t, y, t)$, which points towards $p_{\text{data}}(x \mid y)$.

Having both, the difference between them is the part of the velocity that is *due to* the condition, and it can be amplified. Guided sampling evaluates the network twice and steps along an extrapolation:

$$v^{\text{cfg}} = (1 + w)\, v^{y} - w\, v^{\emptyset} = v^{\emptyset} + (1 + w)(v^{y} - v^{\emptyset}). \tag{15.13}$$

At $w = 0$ this is the ordinary conditional field. At $w > 0$ the step points further towards the conditional distribution than the model's own conditional prediction does — past it, in fact, which is why this is extrapolation rather than interpolation and why it is not the same as simply trusting the conditional model. Typical values in deployed systems are between 3 and 8.

The name is a historical accident worth explaining only because the literature is full of it. The earlier technique, classifier guidance, trained a separate classifier $p(y \mid x_t)$ on noisy images and pushed each sampling step along $\nabla_x \log p(y \mid x)$, in effect running an adversarial attack on that classifier at every step. Classifier-free guidance obtains the same effect without the second model, and inherited a name that describes what it does not contain.

The cost is arithmetic. Every sampling step now requires two network evaluations instead of one, so guidance doubles the cost of generation on top of the thirty-to-fifty-fold cost the iterative sampler already carries. Against a GAN's single forward pass, a guided fifty-step diffusion sampler is one hundred network evaluations per image.

15.11. Diffusing somewhere cheaper

A hundred forward passes is survivable if each pass is cheap and ruinous if it is not, and on raw pixels it is not. The claim that diffusion models are the dominant approach to image generation is, strictly, false: what dominates is latent diffusion, which runs the entire construction of the last four sections somewhere other than pixel space.

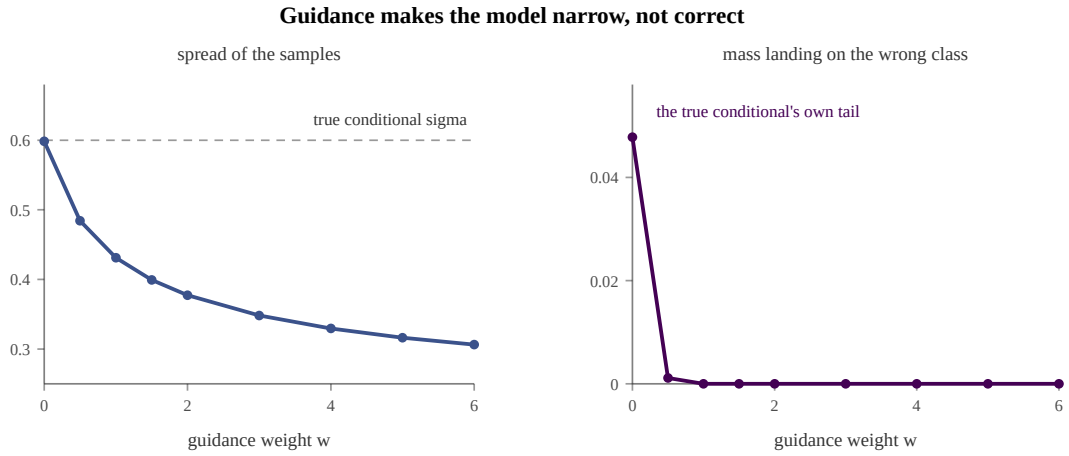


Figure 15.7.: The trade guidance makes, computed by integrating Equation 15.13 on a two-class one-dimensional problem where the exact conditional and unconditional fields are both available in closed form. As the guidance weight rises, the fraction of samples that land closer to the wrong class's mean falls towards zero — the model obeys the condition — while the standard deviation of the samples within the correct class falls well below the true conditional standard deviation, marked. Guidance does not make the model correct: it makes it narrow. Past a certain weight the samples are cleaner, more prototypical examples of the requested class and a worse sample of the distribution the class actually has, which is the reported behaviour of over-guided text-to-image models — saturated, stereotyped, and short on variety.

The pipeline has two stages that are trained separately. First, an encoder and decoder are trained to compress images into a latent representation, downsampling by a factor D spatially while widening the channel dimension to C; a common setting is $D = 8$ and $C = 16$, so a $256 \times 256 \times 3$ image becomes a $32 \times 32 \times 16$ latent. Then the encoder is **frozen**, and a diffusion model is trained on the latents exactly as before: encode an image, interpolate the latent towards noise by Equation 15.8, regress on the velocity. No gradient flows back into the encoder. At sampling time the process runs in reverse — noise in latent space, fifty guided Euler steps, one call to the decoder at the end to turn the clean latent into pixels.

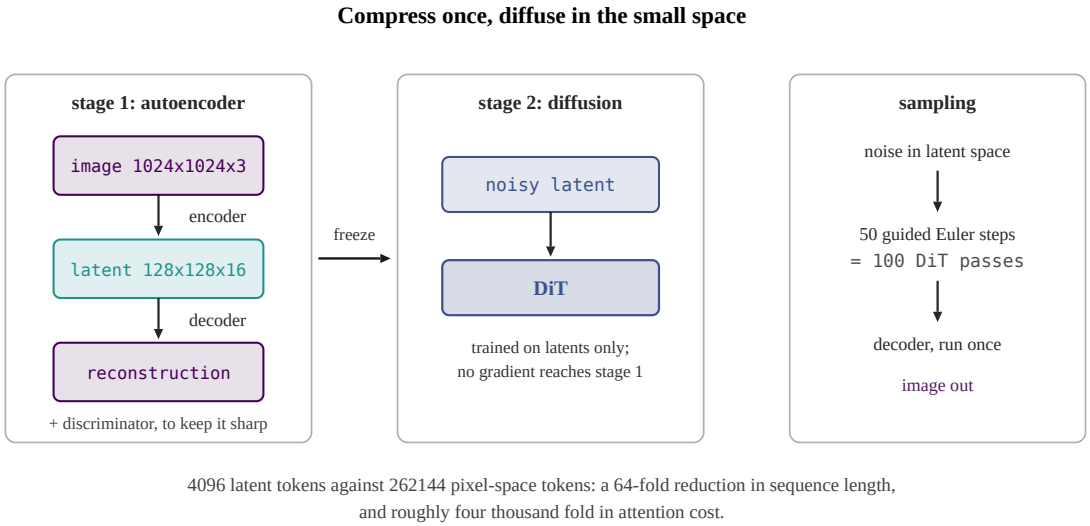


Figure 15.8.: The two-stage pipeline and where each network's gradients stop. The autoencoder is trained first and then frozen; the diffusion model never sees a pixel, and the decoder is invoked exactly once per sample rather than once per step. The token counts along the bottom are for the configuration a current text-to-image model uses, and the 64× reduction they show is the whole reason the pipeline has this shape.

The saving is not subtle. An 8×8 spatial downsampling removes a factor of 64 from the number of positions the diffusion network must process, and the transformer that processes them pays attention costs quadratic in that number, so the same architecture costs roughly four thousand times less per forward pass on latents than on pixels. Multiply that by the hundred passes a guided sampler takes and the difference is between a model that can be trained and one that cannot.

What makes the second stage legitimate is that the compression is chosen to be perceptual rather than semantic. The encoder is asked to throw away exactly the high-frequency detail that a reconstruction loss cannot see anyway, leaving the structure that matters, so the diffusion model spends its capacity on content rather than on texture that the decoder can reproduce unaided.

Which raises the question of what the autoencoder is. It is a variational autoencoder — the architecture of the previous chapter, trained with the ELBO of Section 14.12 but with the prior term's weight turned far down, since nobody intends to sample from this latent space directly and the only thing wanted from the KL term is that the latents not drift to arbitrary scale. But a variational autoencoder's decoder is blurry, and here the decoder's output *is* the final image: whatever it cannot reconstruct, the whole pipeline cannot generate, however good the diffusion model above it.

The fix is the previous section of this chapter. Add a discriminator on the decoder's output and train the autoencoder adversarially alongside its reconstruction loss, which restores the high-frequency detail that the Gaussian likelihood averages away. Note where the GAN has ended up: not as the generative model, where it was uncontrollable, but as a perceptual loss on an autoencoder, where the thing it is asked to do is local and the training is stabilised by the reconstruction term sitting next to it.

So the state of the art is not one of the families in these two chapters. It is a variational autoencoder, trained with an adversarial loss, whose frozen latent space carries a diffusion model, conditioned on the output of a pretrained language model. Every family in this course's treatment of generative modelling appears in the modern pipeline, doing the one job it does well.

15.12. The backbone, and where the conditioning attaches

The network inside all of this is a plain transformer. Diffusion transformers cut the latent into patches, treat the patches as a sequence, and run standard blocks over them; there is no diffusion-specific architecture, and the U-Nets that earlier diffusion models used have largely been replaced. The one genuine design question is how three quite different inputs — the noisy latent, the scalar timestep, and the conditioning signal — get into a stack that takes a sequence.

They get in by two different routes, chosen to match their shapes. The timestep is a single scalar shared by every position, so it is injected by **modulation**: a small network maps t to a per-channel scale and shift that rescale the normalised activations inside each block, which is the same adaptive-normalisation mechanism Equation 15.7 used in StyleGAN, applied here to a different quantity. The conditioning signal is itself a sequence — a text prompt is a variable number of embedding vectors — so it is injected by attention, either through cross-attention layers that let image tokens attend to text tokens, or by concatenating both into one sequence and running joint attention over the whole thing. The DiT paper compares these choices directly and finds adaptive normalisation the strongest for the timestep.

For text-to-image, the text embeddings come from a pretrained, frozen encoder — T5, CLIP, or both — so the language understanding is inherited rather than learned. The full path for a current open-weights model, FLUX.1, is: prompt through the frozen text encoders; a 1024×1024 target image corresponding to a $128 \times 128 \times 16$ latent; a 2×2 patchify on top of the autoencoder's downsampling, giving a 64×64 grid, which is 4,096 image tokens; a twelve-billion-parameter transformer run once per sampling step; and the decoder run once at the end. The deck states this token count as 1,024, which does not follow from the 64×64 grid on the same line — the sequence a model in this configuration processes is 4,096 tokens, and that is the number the video comparison below is built on.

15.13. Video, and the bill for many steps

Text-to-video changes one thing in the diagram and it is not the architecture. The latent acquires a time axis, so it is $t \times h \times w \times c$ instead of $h \times w \times c$; the autoencoder becomes spatio-temporal, downsampling in time as well as space; and everything else — the DiT, the frozen text encoder, guided Euler sampling — is unchanged. Meta's Movie Gen uses $8 \times 8 \times 8$ downsampling and a thirty-billion-parameter transformer to produce a 257-frame 1024×576 video from a $33 \times 128 \times 72 \times 16$ latent.

The difficulty is entirely in the sequence length. A $1 \times 2 \times 2$ patchify over that latent yields about 76,000 tokens against a text-to-image model's 4,096 — around nineteen times as many, and since attention is quadratic, roughly three hundred and forty times the attention cost per forward pass. This is why video models are expensive in a way that does not follow from their parameter counts, and why the entire field's progress over the eighteen months to mid-2025 reads as a sequence-length engineering problem more than a modelling one.

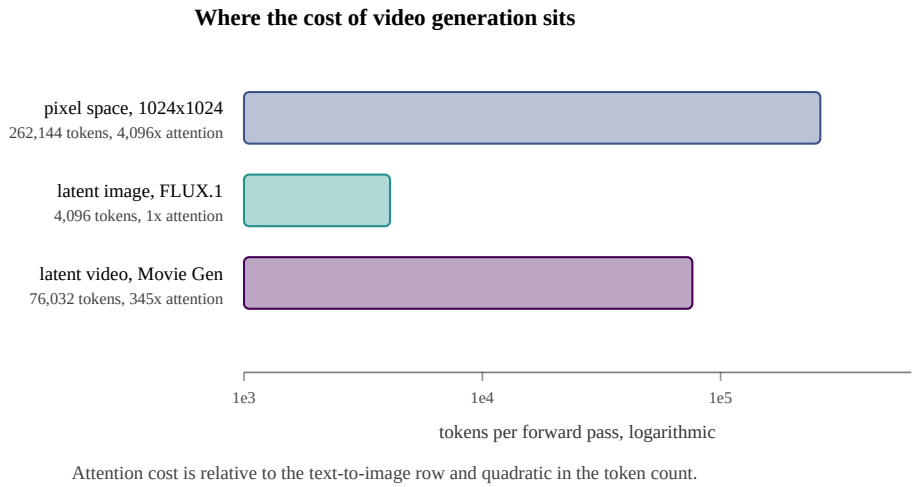


Figure 15.9.: Where the cost sits, on a logarithmic axis because the quantities span four orders of magnitude. Bars show the sequence length a transformer processes under three configurations — pixel space at the target resolution, the latent used by a current text-to-image model, and the latent used by a current text-to-video model — with the quadratic attention cost of each shown relative to the text-to-image case. Latent diffusion buys back four orders of magnitude of attention cost; adding a time axis gives most of that back. The numbers are computed from the downsampling and patchify factors the respective models report, and the axis is logarithmic, so equal bar lengths represent equal ratios rather than equal differences.

That leaves the other cost, the one Section 15.8 introduced and every section since has multiplied: a sample requires tens of sequential forward passes through a very large model, and guidance doubles them. **Distillation** is the family of methods aimed at this. The shared idea is to train a student that reproduces what the many-step sampler produces in far fewer steps — sometimes one — by supervising it against the teacher's trajectory or its endpoint rather than against the data. All of them trade sample quality for step count, and the research question is how little quality has to be given up; single-step distillation works and visibly costs something. This is an area moving fast enough that any specific method named here would date quickly, which is a reasonable summary of where inference-time efficiency in generative modelling currently stands.

15.14. One object, several languages

Section 15.7 presented one specific recipe, and the reason the literature is hard to read is that almost every paper presents a different one. The differences are smaller than the notation suggests. Write the training step in its general form: sample x and z, sample $t \sim p_t$, and then

$$x_t = a(t)\, x + b(t)\, z, \qquad y_{\text{gt}} = c(t)\, x + d(t)\, z, \tag{15.14}$$

with the loss always $\|f_\theta(x_t, t) - y_{\text{gt}}\|_2^2$. Four scalar functions of t, and every variant is a choice of those four. Rectified flow takes $a = 1 - t$, $b = t$, $c = -1$, $d = 1$, the last two being constants, which is why its target is the simple difference $z - x$.

Two other choices are common enough to name. **Variance-preserving** schemes set $a(t) = \sigma(t)$ and $b(t) = \sqrt{1 - \sigma(t)^2}$, collapsing the two interpolation coefficients into a single noise schedule chosen so that $a^2 + b^2 = 1$; when x and z are independent with unit variance, x_t then has unit variance at every noise level, which keeps the network's input distribution constant across t. **Variance-exploding** schemes take the opposite tack, $a(t) = 1$ and $b(t) = \sigma(t)$, leaving the data untouched and adding noise of growing magnitude on top, which requires $\sigma(1)$ to be large enough to drown the signal entirely.

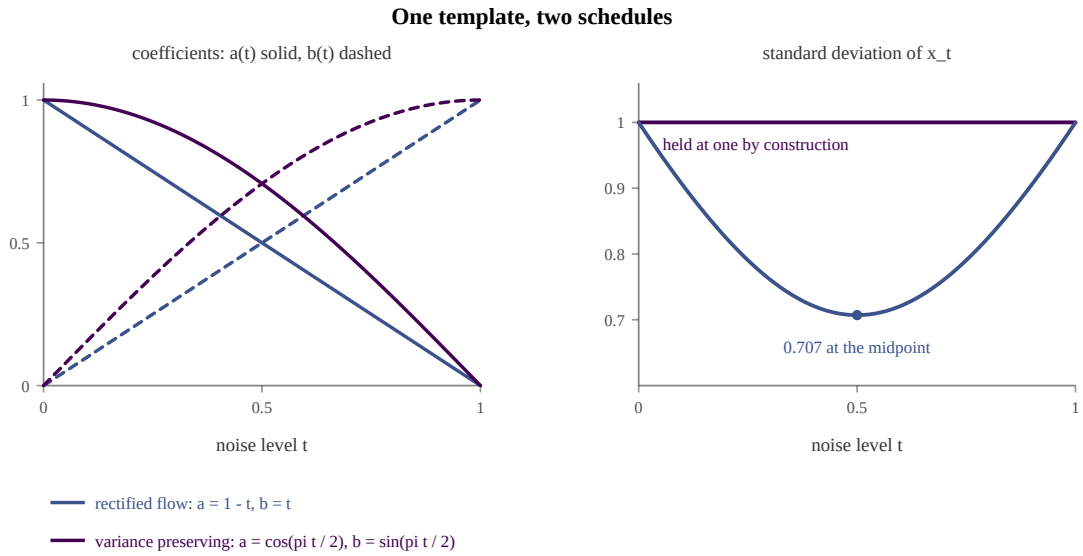


Figure 15.10.: The two most common interpolation schemes, and the property one of them is named for. The left panel plots the coefficients $a(t)$ and $b(t)$ against noise level: rectified flow's are straight lines, while the variance-preserving pair traces a quarter circle. The right panel plots the resulting standard deviation of x_t for unit-variance data and noise. Variance preservation is exactly the flat line — the network sees inputs of the same scale at every noise level — while the linear interpolation sags to $1/\sqrt{2}$ in the middle, so a rectified-flow network must cope with an input whose magnitude varies by thirty per cent over the range of t it is trained on.

The target functions c and d have names too. Setting $c = 1, d = 0$ is **x-prediction**, asking the network for the clean image; $c = 0, d = 1$ is **ϵ-prediction**, asking for the noise that was added, which is what the original denoising diffusion papers used; and $c = b(t), d = -a(t)$ is **v-prediction**. These are not different models. Given x_t and t, any one of the three determines the other two by Equation 15.14, so a network trained on one target can be read as predicting any of them. What differs is the implied weighting of the loss across noise levels, since predicting the noise at low t is a differently scaled problem from predicting the image there, and that weighting matters in practice even though the parameterisations are formally equivalent.

Choosing four functions of t by intuition is hopeless, which is why the field reaches for mathematics, and there are three distinct frameworks that supply it. The first treats **diffusion as a latent variable model**: the noisy versions $x_1, \dots, x_T$ of a clean image are unobserved variables with a known forward process, the network approximates the reverse process, and training maximises

a variational lower bound derived exactly as in Section 14.12. This is the DDPM formulation, and under it a diffusion model is a very deep variational autoencoder whose encoder has no parameters.

The second treats **diffusion as score estimation**. For any density p, its score is the gradient of its log,

$$s(x) = \nabla_x \log p(x), \tag{15.15}$$

a vector field pointing towards regions of higher probability. The claim is that a diffusion network learns not one score but a family of them, one for each noise level, for the sequence of increasingly smoothed distributions p_t. That claim is checkable rather than atmospheric, because the score and the velocity are related by an exact affine identity. Conditioned on x, the interpolant is Gaussian with mean $(1-t)x$ and standard deviation t, and averaging its score over the posterior gives $\nabla \log p_t(x_t) = -\mathbb{E}[z \mid x_t]/t$; substituting the optimal velocity from Equation 15.12 yields

$$\nabla_{x_t} \log p_t(x_t) = -\frac{x_t + (1-t)\, f^*(x_t, t)}{t}. \tag{15.16}$$

The chapter's figure script checks this by computing the left-hand side directly, differentiating the closed-form marginal density of the Gaussian problem in Equation 15.11, and comparing it against the right-hand side built from Equation 15.11 itself. They agree to twelve decimal places across the range of t. The velocity network and the score network are the same network in different clothes.

The third treats **diffusion as solving a stochastic differential equation**, $\mathrm{d}x = f(x,t)\,\mathrm{d}t + g(t)\,\mathrm{d}w$, which describes the noising process as a continuous-time random walk and generation as its time reversal. Its practical payoff is that sampling becomes a problem in numerical integration with a literature attached, and Equation 15.10 is revealed as the crudest method in that literature — forward Euler on the deterministic reduction of the equation. Better integrators exist, which is where much of the recent progress on few-step sampling comes from. Sander Dieleman's survey of these perspectives counts eight rather than three and is the best entry point to the area.

15.15. Autoregression comes back

The previous chapter dismissed autoregressive models for images on a single ground: sampling costs one network pass per subpixel, which is millions of sequential passes for a real image. That argument is about pixels, not about autoregression, and Section 15.11 showed how to stop working in pixels.

The construction is the same two-stage pipeline with a different second stage. Train an encoder and decoder that map images to a grid of **discrete** latents — integers drawn from a learned codebook, as in VQ-VAE — and then train an autoregressive transformer over the resulting sequence of tokens, exactly as a language model is trained over text. Generation samples tokens one at a time from the transformer and hands the completed grid to the decoder. A 32×32 latent grid is 1,024 sequential steps rather than three million, which is expensive but no longer absurd, and the model recovers the exact likelihood that the diffusion route gives up.

The two modern recipes are therefore the same shape — compress, model the compressed representation, decode — and differ only in what models the middle. That is a more useful summary of the field than a list of five families, and it is why both chapters were necessary before either recipe could be stated.

15.16. What each family bought

The previous chapter's ledger can now be completed. Each row is a consequence of one design decision rather than of architecture, and no column dominates.

	Autoregressive	Variational autoencoder	GAN	Diffusion
Density $p(x)$	exact	lower bound	none	bound, via Section 15.14
Latent code	none	explicit	implicit, no encoder	same shape as the data
Passes per sample	one per token	one	one	30–100
Training signal	likelihood	likelihood	a second network	regression on noise
Characteristic failure	slow sampling	blur	mode collapse	slow sampling

The pattern across the row for training signal is the one worth carrying away. Maximum likelihood is stable and mode-covering, and its price is paid in blur. Adversarial training is mode-seeking and sharp, and its price is paid in stability. Diffusion's contribution was to notice that these are not the only two options: a plain regression loss, applied to a problem decomposed into steps small enough that each one has a well-defined answer, is stable like the first and sharp like the second, and pays instead in sampling time — the one currency that hardware and distillation research can actually work on.

What the field settled on is not a winner but an assembly. A modern text-to-image system contains a variational autoencoder for compression, a discriminator to keep its decoder sharp, a diffusion transformer in the latent space, and a frozen language model for the conditioning; the autoregressive alternative swaps the third component and keeps the rest. Every family in these two chapters is in the pipeline, doing the job that its particular trade-off makes it good at.

Part V.

Beyond 2D

16. 3D Vision

Lecture 15

Based on Lecture 15 of CS231n, Stanford University, Spring 2025, given by Jiajun Wu.

16.1. The question images never ask

Everything in the preceding fourteen chapters rested on a convention so uniform that it never had to be argued for. An image is a grid of pixels. Two hundred by two hundred, three channels, row-major; a JPEG and a PNG disagree about compression and agree completely about what they are compressing. Convolution assumes that grid, pooling assumes it, patch embedding assumes it, and the entire apparatus of transfer learning works because every image anyone has ever trained on is the same kind of object as every other.

Geometry has no such convention, and the absence is not an oversight waiting to be corrected. A shape can be handed to you as a few hundred thousand points scraped off a laser scanner, as a mesh of triangles with a texture atlas, as a solid assembled from cylinders and boxes by Boolean operations in a CAD kernel, as the zero level set of a function, or as the weights of a neural network that answers queries about occupancy. These are not five encodings of one canonical object in the way that PNG and JPEG are. They support different operations, they fail in different places, and converting between them loses something every time.

The lecture opens by listing what a representation is being asked to do, and the list is longer than it first appears. It has to be **stored**, which for a pixel grid is a matrix and for an irregular point set is already a question. It has to support **creation** — someone or something has to author new shapes, whether that is an artist with a mouse, a scanner, or a generative model conditioned on a sentence. It has to support **operations**: editing, smoothing, simplification, repair, the whole toolbox that a shape passes through between capture and use. It has to **render**, turning geometry back into pixels, and this one matters more than the others for our purposes, because rendering is the bridge that lets two-dimensional data supervise three-dimensional models. And for anything that moves — humans, animals, articulated machines — it has to **animate**, which means the representation has to survive being deformed.

No single choice is good at all five. That is the whole reason the taxonomy exists rather than a winner, and it is why a chapter on 3D vision has to spend its first half on geometry before it can spend its second half on learning. The deep-learning results in Section 16.7 onwards are not a sequence of architectures. They are a sequence of *bets on representation*, and each one is legible only against the map that the classical literature drew first.

16.2. Points, and what they cannot tell you

A **point cloud** is an unordered set of coordinates in $\mathbb{R}^3$, and it is the simplest thing that can be called a shape. Where an image is an $n \times n$ matrix whose entries are indexed by position, a point cloud is a $3 \times n$ matrix whose columns are positions and whose column order carries no information at all. That last clause is doing more work than it looks like; Section 16.9 is entirely about its consequences.

Points are what sensors produce. A structured-light depth camera, a lidar unit, a photogrammetry pipeline, the scanner in a modern phone — all of them emit points, possibly noisy, usually in the sensor's own frame, and getting a single coherent cloud out of several passes is the registration problem. Each point may carry a surface normal alongside its position, in which case it is called a **surfel**: a position with an orientation. The normal is not decoration. Shading a surface requires knowing how it faces the light, so a cloud without normals can be plotted but not convincingly rendered.

The appeal is that points constrain nothing. Any geometry at all can be approximated by enough of them, at any scale, with no requirement that the shape be closed, connected, or of any particular topology, and adding detail in one region is a matter of putting more points there. The price is that a set of positions is *less than* a surface, and the shortfall is precisely topological. Given a ring of points, there is no way to tell whether they were sampled from a torus or from a circle of disconnected beads; the samples are identical and the connectivity that would distinguish them was never recorded. Nothing in the representation says which points are neighbours, so the operations that depend on neighbourhood — subdividing, simplifying, smoothly interpolating between samples — have nowhere to start. They can be recovered, at a cost, by imposing a spatial data structure and estimating neighbourhoods from proximity, which is exactly what the point-cloud networks of Section 16.9 quietly spend their capacity on.

The **polygonal mesh** is the answer to that shortfall, and it is the representation the rest of the graphics world runs on. Keep the vertices, and add faces: a list saying which vertices bound each polygon. This is a *boundary representation* — the surface of the object is stored, the interior is implied — and it converts a cloud into a piecewise-linear surface with a well-defined inside, a well-defined normal per face, and an explicit notion of adjacency. Meshes scale to essentially any size. The Digital Michelangelo Project's scan of the *David* carries 28,184,526 vertices and 56,230,343 triangles, and Google Earth's reconstruction of the planet from satellite and aerial imagery runs to trillions.

What connectivity buys is the toolbox. **Subdivision** raises resolution by inserting vertices according to a fixed rule, so a coarse control mesh can carry fine detail without an artist placing every vertex. **Simplification** goes the other way, collapsing edges to reduce triangle count while preserving silhouette and appearance, which is how a game engine draws a distant object cheaply. **Regularisation** redistributes vertices so that triangles are near-equilateral and roughly uniform in size, because almost every downstream algorithm behaves better on a well-conditioned mesh than on a mesh with slivers. None of these three has a natural analogue on a raw point cloud, and all three are immediate once faces exist.

The same connectivity is what made meshes awkward for early deep learning. A raw mesh is irregular in two separate ways: different faces may have different numbers of vertices, and different meshes have different numbers of faces. A convolutional network of the era wanted a fixed-resolution grid, and a mesh is not one. This is a large part of why deep learning arrived in 3D vision several years after it had settled everything in 2D — not because the ideas did not transfer, but because there was no agreement on what the network should consume.

16.3. Surfaces as maps

Points and meshes are both *non-parametric*: they store samples, and the number of numbers grows with the amount of detail. But a great deal of real geometry is not arbitrary. The intrinsic dimensionality of a curve is one and of a surface is two, whatever ambient space they sit in, and a manufactured object is usually built from pieces that were specified by a designer rather than measured off a scanner. A **parametric** representation exploits both facts by storing the map instead of the samples.

The pattern is to name a low-dimensional domain and a function out of it. A circle in the plane is the image of an interval,

$$\gamma(t) = (\cos t, \sin t), \qquad t \in [0, 2\pi), \tag{16.1}$$

and a sphere in space is the image of a rectangle under the two angles of latitude and longitude. In both cases the shape is *the range of a function*: the parameters are the true degrees of freedom, and the function is what carries them into $\mathbb{R}^3$. Nothing is sampled until someone asks for a point.

Trigonometric formulas cover the shapes that happen to have them, which is not many. The workhorse of design is instead a family of polynomial maps controlled by a handful of points that an artist can drag. A **Bézier curve** of degree n is a weighted average of $n+1$ control points $\mathbf{b}_0, \dots, \mathbf{b}_n$, where the weights are the Bernstein polynomials:

$$\gamma(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \, \mathbf{b}_i, \qquad t \in [0, 1]. \tag{16.2}$$

The weights are non-negative and sum to one for every t, which is why the curve never escapes the convex hull of its control points — the property that makes the control polygon a usable interface. The same expression can be evaluated without ever writing a binomial coefficient, by **de Casteljau's algorithm**: repeatedly linearly interpolate along the control polygon,

$$\mathbf{b}_i^{(k)}(t) = (1-t)\, \mathbf{b}_i^{(k-1)}(t) + t\, \mathbf{b}_{i+1}^{(k-1)}(t), \tag{16.3}$$

starting from $\mathbf{b}_i^{(0)} = \mathbf{b}_i$ and stopping when a single point $\mathbf{b}_0^{(n)}(t) = \gamma(t)$ remains. That the recursion and Equation 16.2 agree is not obvious from the two expressions, and the generator behind Figure 16.1 checks it numerically rather than taking it on faith.

Surfaces follow by taking the tensor product. Run a Bézier curve in u whose control points are themselves the values of Bézier curves in v, and the result is a **Bézier patch** governed by a 4×4 grid of control points for the bicubic case; a complicated object is then a quilt of patches joined with matching tangents at their seams. This is the machinery underneath the spline and NURBS surfaces that CAD systems use, and it is why a designed object can be edited long after it is authored: moving one control point changes the surface smoothly and locally, which is not true of moving one vertex of a dense mesh.

Subdivision sits between the parametric and non-parametric worlds and is worth seeing as the limit of a rule rather than as a formula. Chaikin's corner-cutting scheme takes a polygon and replaces every edge by the two points one quarter and three quarters of the way along it, discarding the original vertices; iterate, and the polygons converge to a quadratic B-spline. The surface analogues — Catmull–Clark, Loop — do the same thing to a mesh, and they give an

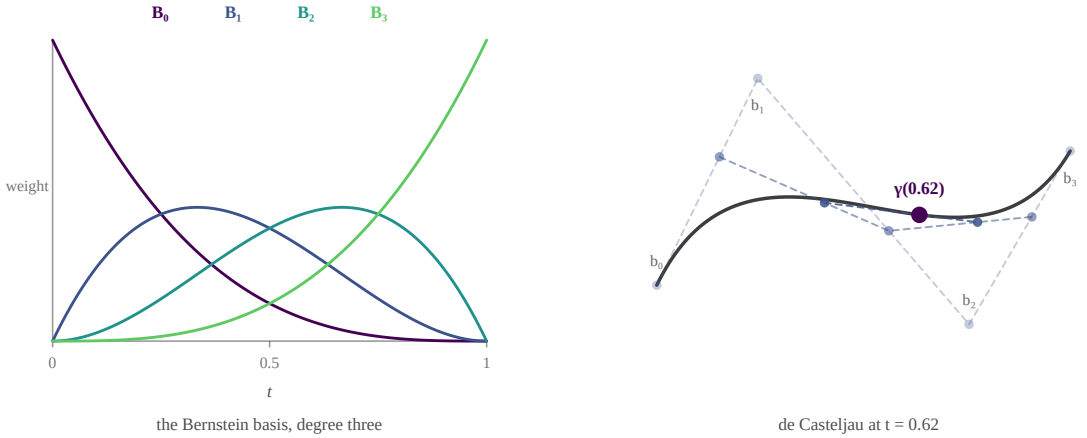


Figure 16.1.: Two routes to the same cubic Bézier curve. Left, the Bernstein weights of Equation 16.2 as functions of t; right, the de Casteljau construction of Equation 16.3 at $t = 0.62$, where each round of linear interpolation shrinks the control polygon by one vertex until a single point is left on the curve. The generator evaluates both forms on a dense grid and asserts they agree to machine precision.

artist a coarse control cage whose refinement is smooth by construction. The representation stored is discrete; the object it denotes is the limit.

16.4. The asymmetry that organises everything

Points, meshes, splines and subdivision surfaces differ in almost every respect and agree on one: they all hand you points. Sample the parameters, evaluate, and you have a location that is *on the surface* by construction. This is what makes them **explicit**. The alternative is to give up direct access to the points and instead state a condition that the points on the surface satisfy — to define the shape as the set where some function vanishes,

$$S = \{\, \mathbf{x} \in \mathbb{R}^3 : f(\mathbf{x}) = 0 \,\}, \tag{16.4}$$

which is an **implicit** representation. The unit sphere is $f(\mathbf{x}) = x^2 + y^2 + z^2 - 1$, and nothing about that expression tells you a single point on the sphere.

The contrast is sharpest when the same object is written both ways. A torus is the image of a square under

$$f(u, v) = \big((2 + \cos u)\cos v,\ \ (2 + \cos u)\sin v,\ \ \sin u\big) \tag{16.5}$$

and it is also the zero set of

$$g(x, y, z) = \Big(2 - \sqrt{x^2 + y^2}\Big)^2 + z^2 - 1. \tag{16.6}$$

Ask each for a hundred points on the surface. The explicit form answers by drawing a hundred (u, v) pairs and evaluating — a hundred function calls, every answer exact. The implicit form

has to solve $g = 0$, an equation in three unknowns with a two-dimensional solution set, and for Equation 16.6 that is still tractable by hand while for an arbitrary g it is a root-finding problem with no closed form at all.

Now ask each whether the point (3/4, 1/2, 1/4) lies inside the corresponding sphere. The implicit form evaluates its function once:

$$f\left(\tfrac{3}{4}, \tfrac{1}{2}, \tfrac{1}{4}\right) = \tfrac{9}{16} + \tfrac{4}{16} + \tfrac{1}{16} - 1 = -\tfrac{1}{8} < 0,$$

so the point is inside, since f is negative within the shape, zero on it and positive outside. The explicit form has no comparable move. A parameterisation is a map *out of* the domain; it offers no way to run backwards from an arbitrary query point, and deciding containment means something like casting a ray and counting crossings of the surface — a global computation, not an evaluation.

The two representations are therefore not ranked. They are **exactly opposed**, and it is worth stating the opposition plainly because everything in the rest of this chapter is a consequence of it.

	Explicit	Implicit
Sample a point on the surface	evaluate the map — trivial	solve $f = 0$ — hard
Test whether a query point is inside	global computation — hard	evaluate f once — trivial
Compose two shapes	remesh the intersection	arithmetic on the two functions
Storage grows with	the amount of detail stored	the complexity of f

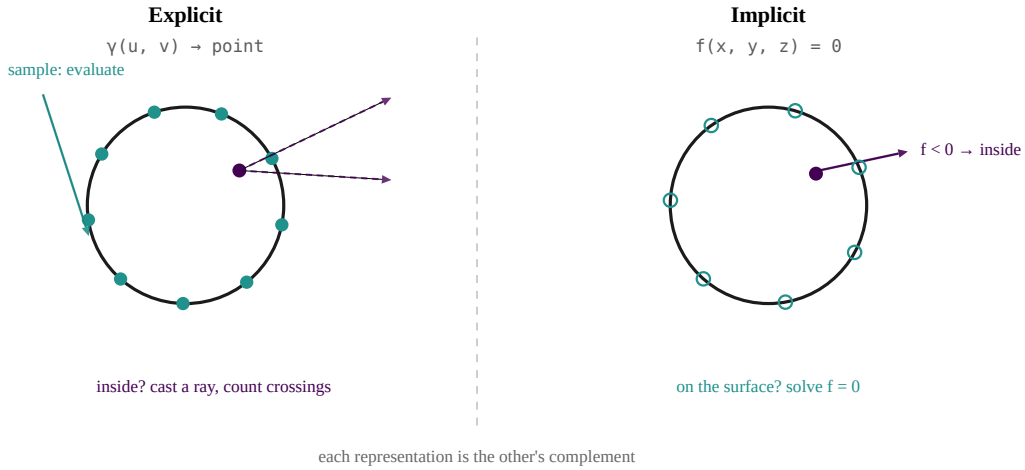


Figure 16.2.: The same asymmetry on the same two objects. Sampling the surface is one evaluation of the explicit map and a root-find for the implicit function; testing containment is one evaluation of the implicit function and a ray cast for the explicit map. Neither representation is better; each is the other's complement.

Which query a method needs is what decides its representation, and the reason this section is load-bearing is that the query changed. Classical geometry processing mostly wanted to sample surfaces, which is why explicit forms dominated for thirty years. Neural rendering wants to ask,

of an arbitrary point in space, what the density is there, what the colour is, what the material is — the containment query, repeated millions of times per image. That shift is the single best explanation for why implicit representations, a niche of CAD and level-set methods in 2010, became the default in 3D deep learning by 2020.

16.5. Building shapes out of functions

An implicit function need not be pulled out of the air. The simplest family is the **algebraic surfaces**, zero sets of polynomials, which gives spheres, ellipsoids, cylinders, quadrics and the occasional decorative quartic and then runs out of usable expressions long before it reaches anything one would want to manufacture. Nobody writes down the polynomial whose zero set is a cow.

What rescues the implicit view is that its shapes **compose arithmetically**. If f and g are the functions of two solids under the sign convention that negative means inside, then

$$f \cup g = \min(f, g), \qquad f \cap g = \max(f, g), \qquad f \setminus g = \max(f, -g), \tag{16.7}$$

and each of these is a one-line operation on values rather than a geometric construction. This is **constructive solid geometry**, and it is how mechanical CAD has worked for decades: a part is a tree whose leaves are primitives and whose internal nodes are Booleans, and the whole tree is evaluated at a query point by evaluating the leaves and folding upwards. Doing the same thing to two meshes means computing the surface of intersection, splitting triangles along it and repairing the result — a notoriously fragile operation that implicit geometry gets for the price of a comparison.

Booleans give hard creases. Something smoother is available when the function carries more information than its sign, and the natural choice is the **signed distance function**, whose value at $\mathbf{x}$ is the distance from $\mathbf{x}$ to the surface, negative inside:

$$d(\mathbf{x}) = \pm \min_{\mathbf{y} \in S} \|\mathbf{x} - \mathbf{y}\|. \tag{16.8}$$

Now the magnitudes mean something, so shapes can be averaged rather than merely combined: interpolating between two distance fields moves a boundary continuously from one shape to the other, and softened variants of Equation 16.7 blend two solids into a fillet instead of a crease. Practitioners of the technique compose dozens of such fields into entire rendered scenes, and the reason it is practical at all is that a distance field also tells a ray marcher how far it may safely step — the value at the current point is a radius within which no surface can lie.

That last property is where the lecture's presentation of Equation 16.7 is worth pushing on, because the identity is exact in one place and only a bound in another. Take two overlapping unit disks with centres at $(\pm\frac{1}{2}, 0)$ and query the origin. Each disk reports $d = -\frac{1}{2}$, so min reports $-\frac{1}{2}$; the true distance from the origin to the boundary of the *union* is $\sqrt{3}/2 \approx 0.866$, because the arcs that ran nearest the origin are interior to the other disk and no longer part of the boundary at all. Outside the union the two agree exactly, and inside it min systematically understates the depth. The composition rules are therefore safe for the thing ray marchers use them for — stepping in from outside — and are not a true distance function on the interior, which is why implicit-surface code that needs interior distances recomputes rather than composing. The generator behind Figure 16.3 evaluates $\min(d_1, d_2)$ against a brute-force distance to the union's

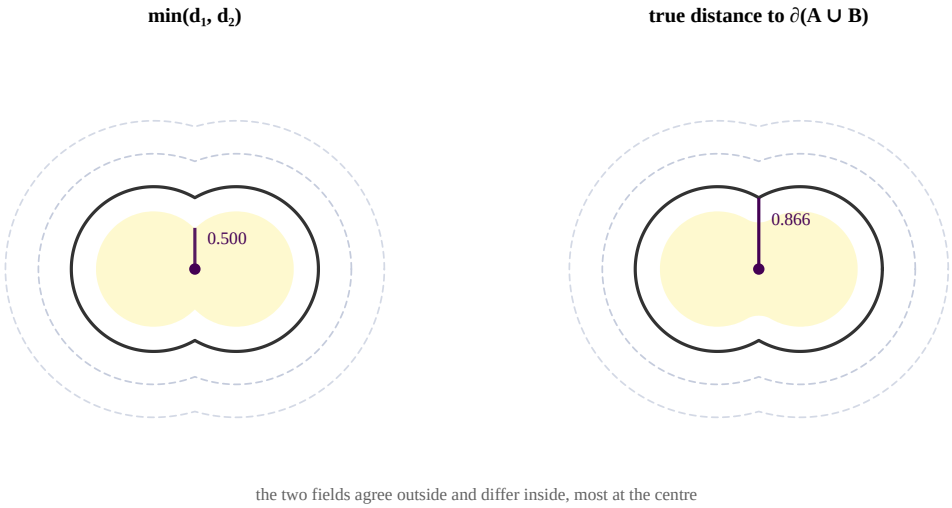


Figure 16.3.: The signed distance fields of two overlapping disks and of their union. Contours are level sets of the field, and the heavy contour is the surface itself. Outside the union, $\min(d_1, d_2)$ reproduces the true distance exactly; inside it, the arcs swallowed by the other disk are no longer boundary, so the composed field understates the true depth — largest at the centre, where it reports 0.5 against a true 0.866.

boundary over a grid and reports the largest discrepancy on each side of the surface, so the claim is measured rather than remembered.

Composition still leaves the problem that a complicated object needs a large expression tree, and evaluating hundreds of primitives per query is slow. The fix is to give up the closed form and **precompute**: lay a regular grid over the domain, evaluate the function once at every node, and store the values. The surface is then recovered wherever interpolated values cross zero, which for a grid means wherever two adjacent samples have opposite sign. This is the **level-set** representation, and it is what medical imaging has always produced — a CT or MRI volume is a scalar field, and an isosurface of constant tissue density is a shape extracted from it exactly this way.

Discard the magnitudes and keep only the sign and the level set collapses into **voxels**: a three-dimensional array of occupancies, one bit or one probability per cell. The lecture arrives at voxels by this route deliberately, because the usual route obscures what they are. Presented as "pixels with one more index", a voxel grid looks like the obvious first thing to try and its costs look incidental. Presented as a *sampled implicit function*, it is visibly an approximation that has thrown away both the closed form's exactness and the distance field's magnitudes, and the only thing it has bought in exchange is a regular array. That purchase turns out to matter enormously, because a regular array is what a convolution wants.

Every representation in this chapter's first half now has a place on a single map, and the map is the object the rest of the chapter moves around. One axis is the explicit–implicit distinction of Section 16.4. The other is whether the shape is stored as samples or as a function of a few parameters.

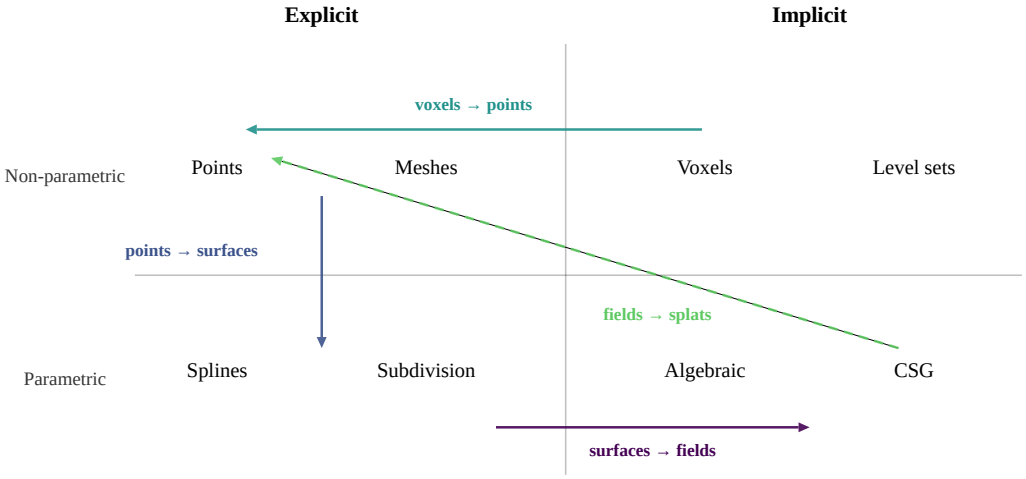


Figure 16.4.: The taxonomy the lecture builds, with the four moves that the deep-learning literature made across it. Voxels were the first target because they are the only cell a convolution can be run on unchanged; each later move is a bet that some other cell suits the query better.

16.6. What there was to learn from

Methods in 3D lagged 2D by several years, and part of that was the representation problem of Section 16.2. The rest was data. The standard benchmark before deep learning was the Princeton Shape Benchmark of 2004: 1,814 models across 182 categories, which is about ten examples per class. It was considered large. Ten examples per class is not a training set under any modern reading; it is an evaluation set for methods that learn nothing.

The change came from the same place it came from in 2D — someone assembled a corpus large enough that learning became the obvious thing to do. ShapeNet indexed more than three million models from online repositories, of which 220,000 were sorted into 3,135 categories, and released **ShapeNetCore**, a clean subset of 51,300 models in 55 categories that became the benchmark for essentially every method in the second half of this chapter. The scale-up did not stop: Objaverse collected 818,000 artist-made assets, and Objaverse-XL roughly 10.2 million, drawn mostly from GitHub and Thingiverse.

Synthetic assets have a specific weakness. They are what someone chose to model, which means clean topology, canonical poses, plausible materials and no sensor noise — the distribution of *CAD files*, not of objects. So a second lineage grew alongside it, made of scans. CO3D contains nearly 19,000 turntable-style videos of objects in 50 MS-COCO categories, 1.5 million frames with camera poses attached. ScanNet does the same for interiors: 2.5 million RGB-D views across 1,513 scans, with poses, reconstructed surfaces and instance-level semantic labels, later extended by ARKitScenes and ScanNet++. A third lineage annotates *below* the object: PartNet labels 573,585 fine-grained part instances across 26,671 shapes in 24 categories, hierarchically and with mobility, which is the data that Section 16.14 needs and that nothing else supplies.

The comparison to draw from Figure 16.5 is not that 3D data grew, but where it ended up. Objaverse-XL is a genuinely large corpus and it is still smaller than ImageNet, and smaller by a wide margin than the billion-image sets behind current 2D foundation models. The scanned

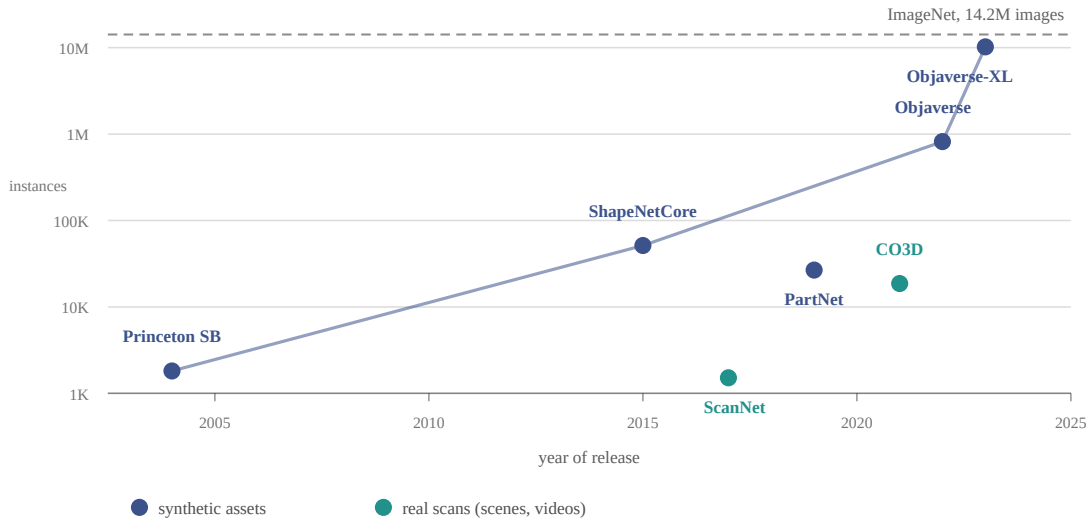


Figure 16.5.: 3D corpora by size and year, on a logarithmic axis, against ImageNet for scale. Four orders of magnitude in twenty years, and still roughly one order short of the 2D corpora that pretrained models are built on — which is the practical case for supervising 3D models through rendering rather than through 3D labels.

datasets, which carry the realism, are three to four orders below that. Any method that needs 3D supervision is therefore working in a regime that 2D vision left behind a decade ago, and this is the standing argument for the recurring move in the rest of the chapter: put a **differentiable renderer** between the 3D model and the loss, and supervise it with photographs, of which there are effectively unlimited numbers.

16.7. Sidestepping the problem: render it first

The first serious attempt to run a network on shapes did not solve the representation problem. It avoided it.

Multi-view CNN places a ring of virtual cameras around the object, renders one image per camera, and hands each image to an ordinary image network — the same architecture, initialised from the same ImageNet weights, that had just finished winning everything in 2D. The per-view features are combined by **view pooling**, an element-wise maximum across the views, and the pooled descriptor goes through a second convolutional stage and a softmax. The max is what makes the design work: it is invariant to which view is which, so the network produces one descriptor per shape rather than one per camera, and the shape can be classified or retrieved from that descriptor alone.

On ModelNet40 the results were not close. The best learned 3D descriptor of the day, the voxel-based 3D ShapeNets, reached 77.3% classification accuracy and 49.2% retrieval mAP. Twelve rendered views through an ImageNet-pretrained network reached 89.9% and 70.1%; going to eighty views moved classification only to 90.1%, so the gain came from rendering at all rather than from rendering exhaustively. The paper's sharpest observation is stranger still: a network trained on the twelve views but shown a *single* view at test time already beat the 3D methods by about eight points. Whatever the voxel networks were learning about three-dimensional

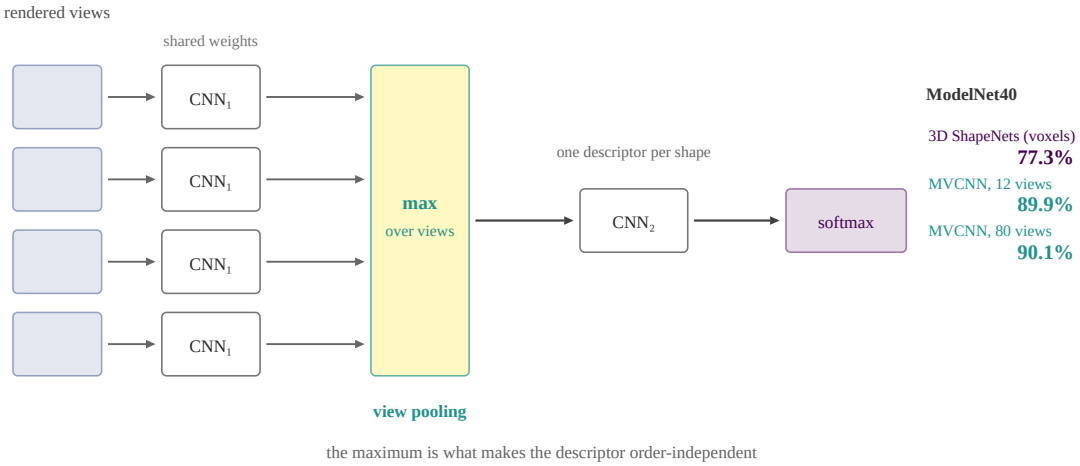


Figure 16.6.: Multi-view CNN. Each rendered view goes through the same weights, an element-wise maximum over views collapses them into a single order-independent descriptor, and a second stage classifies it. The maximum is the whole architectural idea: everything before it is an ordinary image network, and everything after it sees a shape rather than a picture.

structure was worth less than what an ImageNet-pretrained 2D network already knew about surfaces, materials and silhouettes.

The lesson is about data, not about geometry, and it is the direct consequence of Figure 16.5. A projection throws away most of the shape; what it buys in return is access to a pretraining corpus three orders of magnitude larger. In 2015 that trade was overwhelmingly favourable. The field then spent a decade building native 3D methods on the argument that it would stop being favourable as 3D corpora grew — and the lecturer's aside is that with current image and video foundation models the trade may be swinging back, which is a live question rather than a settled one.

Multi-view has two genuine weaknesses that the accuracy numbers hide. It needs a renderable input, so a noisy or partial point cloud has to be reconstructed before it can be photographed, and reconstruction is the hard problem it was supposed to avoid. And its output is a descriptor, not geometry: the pipeline classifies and retrieves, but it cannot produce a shape.

16.8. Voxels, and the exponent they pay

The obvious native alternative was the one that required changing a single line. Voxels are a dense array, so a convolution runs on them unchanged, and the entire 2D toolkit transfers by replacing every 2D operator with its 3D counterpart. 3D ShapeNets did this with a convolutional deep belief network in 2015, generating and classifying shapes at low resolution; 3D-GAN followed by mapping a 200-dimensional latent vector through a volumetric generator to a 64^3 occupancy cube, trained adversarially exactly as in Section 15.2 with every 2D operation swapped for a 3D one.

The interesting move came next, and it is the first appearance in this chapter of the rendering bridge. Visual Object Networks factor generation into three codes — shape, viewpoint, texture — and place a **differentiable projection** between the first two and the third: the generated

voxel shape is rendered from the sampled viewpoint into a depth map and a silhouette, and a second network paints that geometry into a colour image. Two adversarial losses then apply, one demanding that the 3D shapes look like real shapes and one demanding that the 2D renderings look like real photographs. Because the three codes enter at different points, they stay disentangled, so viewpoint, geometry and appearance can be varied independently and a texture can be transferred from one car onto the shape of another. This is the pattern that Section 16.12 makes the centre of the field: the renderer is the thing that lets abundant 2D data supervise scarce 3D structure.

What limits all of it is arithmetic. A voxel grid at resolution n costs $O(n^3)$ memory and compute, while the object it represents is a surface, whose area grows as $O(n^2)$. The ratio is the fraction of cells that carry any information at all, and it falls like $1/n$: almost every cell in a fine grid is either empty space or buried interior, and the network spends the overwhelming majority of its capacity convolving nothing with nothing. Doubling resolution costs eight times the memory to gain four times the surface detail.

Octrees recover the exponent. Partition the cube into eight children, and recurse only into the children that the surface actually crosses; cells far from the surface stay large. The stored structure is then proportional to the surface rather than the volume, which restores $O(n^2)$ scaling, and specialised convolution implementations — OctNet, O-CNN — run directly on the tree. For generation the trick is subtler, because the tree's structure is not known in advance: Octree Generating Networks predict the octree layer by layer, deciding at each level which cells are empty, filled, or mixed and therefore worth subdividing, and only the mixed cells are expanded at the next level. That buys output volumes as large as 512^3 in a single forward pass on hardware where a dense decoder had been stuck near 64^3.

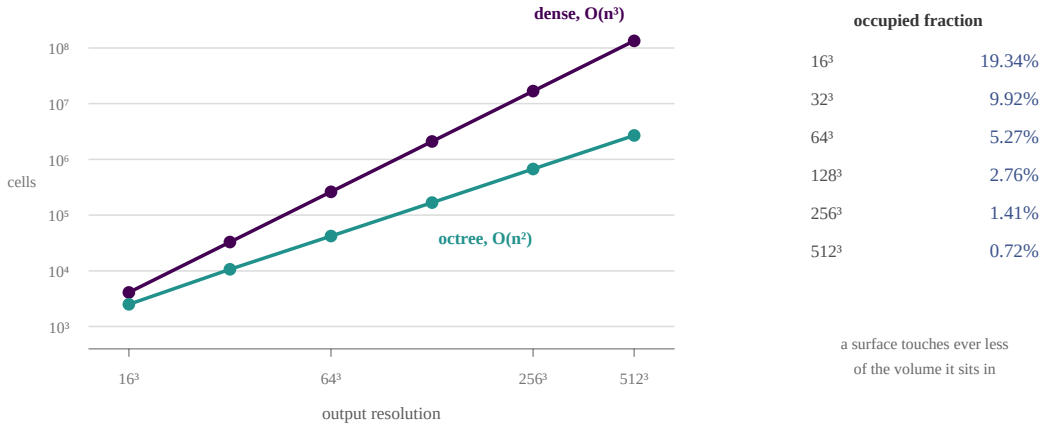


Figure 16.7.: Why the exponent matters. A dense grid costs $O(n^3)$ cells while the surface it represents occupies $O(n^2)$ of them, so the occupied fraction falls as $1/n$: counting the cells a sphere's boundary actually crosses gives 19.3% at 16^3 and 0.72% at 512^3, halving at each doubling to within a few percent. The octree curve counts the nodes a surface-following subdivision actually creates. Both curves are computed by the generator, not traced from a published chart.

An octree is still a sampled implicit function; it has only stopped sampling uniformly. The next move is a bigger one — abandoning the grid entirely.

16.9. Learning on sets: PointNet

Voxels and octrees both discretise *space* and record what is in it. Points discretise the *object* and let it sit wherever it likes. The lecture labels the difference with the terminology of fluid mechanics — Eulerian versus Lagrangian — and it is exactly right: a voxel grid is a fixed lattice of observers reporting what passes through, and a point cloud is a set of particles carrying their own coordinates.

Running a network on the Lagrangian view means confronting the property that Section 16.2 flagged. A point cloud is a *set*. Reorder the columns of the $3 \times n$ matrix and the shape is unchanged, so the network's output must be unchanged too, for all $n!$ orderings. A convolution has no such invariance; feeding a point cloud to an MLP in matrix order gives a network that must learn the invariance from data, which for $n = 1024$ is hopeless. PointNet was the first architecture to build it in.

The requirement is that the network compute a **symmetric function** of its inputs: one whose value is unchanged under any permutation. Familiar examples are the maximum, the sum and the mean, and each is symmetric because it aggregates without reference to position. Applying one directly to the raw coordinates does work and is useless — the element-wise maximum over a point cloud returns a corner of its bounding box, and nothing about the shape survives. The insight is that a symmetric aggregator becomes expressive if the points are transformed first and the aggregate is transformed after:

$$f(\{\mathbf{x}_1, \dots, \mathbf{x}_n\}) \;=\; \gamma\Big(\max_{i=1,\dots,n} h(\mathbf{x}_i) \Big), \tag{16.9}$$

where h is a multilayer perceptron applied to each point independently with shared weights, the maximum is taken element-wise over the n resulting feature vectors, and γ is a second perceptron on the pooled result. Composing a symmetric function with anything applied per-point leaves it symmetric, so Equation 16.9 is permutation-invariant by construction rather than by training. What makes it more than a trick is the accompanying theorem: any continuous set function — continuous with respect to Hausdorff distance on sets — can be approximated arbitrarily well in this form, provided the pooled feature vector is wide enough. The maximum is not a lossy shortcut forced on the architecture by symmetry. It is sufficient.

The same design has a visible weakness. A global maximum discards which features came from which points and where those points were relative to each other, so Equation 16.9 has no local receptive field at all — it is a bag of points, and the analogue of a convolution's locality has to be added back. PointNet++ does it by applying the same construction hierarchically over nested neighbourhoods; dynamic graph CNNs do it by building a k-nearest-neighbour graph over the points and passing messages along its edges, recomputing the graph in feature space at each layer so that "neighbouring" comes to mean semantically rather than spatially close. Both restore locality without giving up permutation invariance, because a graph over a set is itself order-independent.

16.9.1. Comparing two sets

A network that *outputs* a point cloud has a second version of the same problem. The prediction is a set and the target is a set, and a per-element squared error would compare the i-th predicted point to the i-th ground-truth point, which is meaningless when neither ordering carries information. The loss has to be permutation-invariant too, and the Point Set Generation Network introduced the two that the field has used ever since.

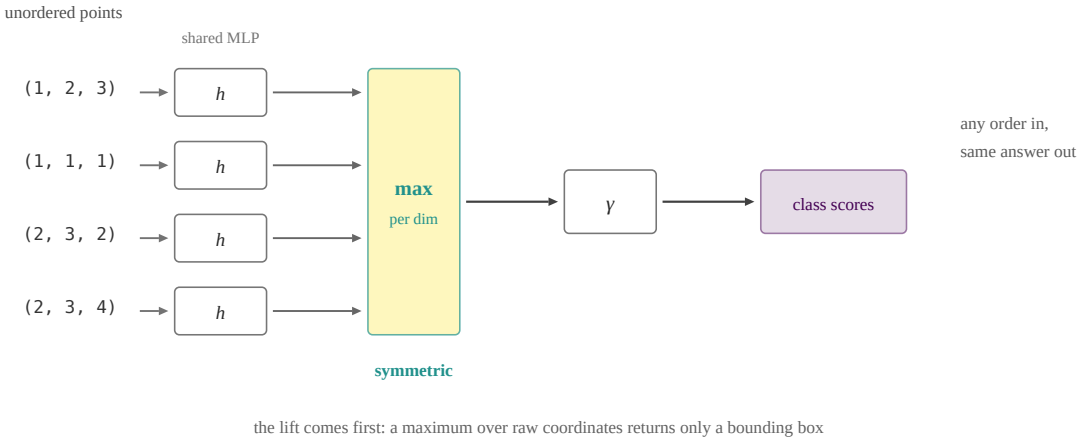


Figure 16.8.: PointNet. Every point passes through the same weights, so the network is blind to input order before pooling; the element-wise maximum then reduces n feature vectors to one global descriptor regardless of n. Applying the maximum to raw coordinates instead of features would return a corner of the bounding box, which is why the per-point lift comes first.

The **Chamfer distance** matches every point to its nearest neighbour in the other set, in both directions:

$$d_{\mathrm{CD}}(S_1, S_2) = \sum_{\mathbf{x}\in S_1} \min_{\mathbf{y}\in S_2} \|\mathbf{x}-\mathbf{y}\|_2^2 \; + \sum_{\mathbf{y}\in S_2} \min_{\mathbf{x}\in S_1} \|\mathbf{x}-\mathbf{y}\|_2^2. \tag{16.10}$$

Each nearest-neighbour search is independent of the others, so the whole thing is trivially parallel and accelerable with a k-d tree, and it is continuous and piecewise smooth in the point positions. It is also not a metric — the triangle inequality fails — and, more consequentially, the matching is not one-to-one. Many predicted points may share a single nearest target, which means a model can leave regions of the target uncovered while every one of its own points sits comfortably close to something.

The **Earth Mover's distance** closes that hole by insisting on a bijection between two sets of equal size:

$$d_{\mathrm{EMD}}(S_1, S_2) = \min_{\phi: S_1 \to S_2} \sum_{\mathbf{x}\in S_1} \|\mathbf{x}-\phi(\mathbf{x})\|_2, \tag{16.11}$$

with ϕ ranging over bijections. Every predicted point is spent on exactly one target point, so nothing can be double-counted and no region can be quietly abandoned. The cost is that Equation 16.11 is an assignment problem rather than a search, far too expensive to solve exactly inside a training loop, so implementations use a $(1+\epsilon)$-approximation with a fixed time budget per instance. It is differentiable almost everywhere, since the optimal assignment is generically unique and locally constant under small movements of the points.

The practical summary is that Chamfer is cheap and tolerant of density mismatch, EMD is expensive and enforces coverage, and a model trained on Chamfer alone tends to produce clouds that score well while sparing themselves the difficult parts of the shape.

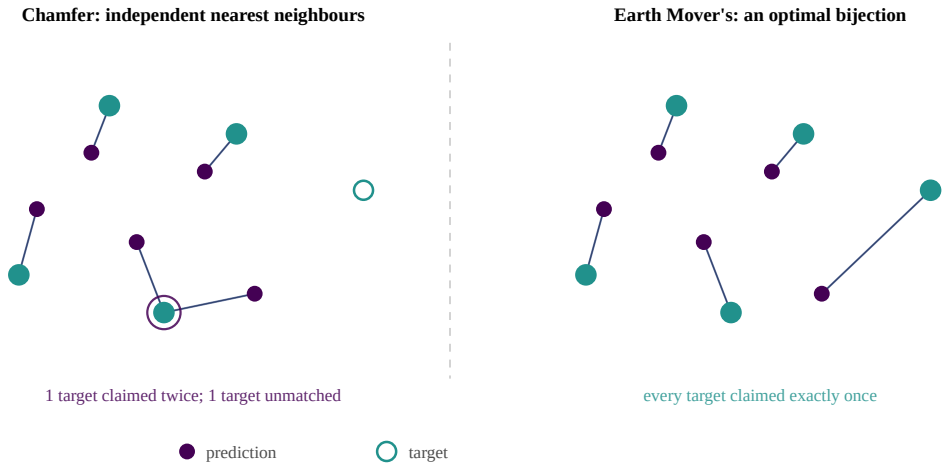


Figure 16.9.: Chamfer against Earth Mover's on the same pair of sets. Chamfer's arrows are independent nearest-neighbour lookups, so several predictions can collapse onto one target and leave a target unmatched; the Earth Mover's assignment is a bijection and every target is claimed exactly once. The matchings are computed by the generator — nearest-neighbour for one, an exact optimal assignment for the other.

16.10. Predicting a surface instead of a cloud

A generated point cloud is still non-parametric, and it inherits everything Section 16.2 said about points: no connectivity, no normals, nothing that renders. If you want a mesh you have to reconstruct one, and reconstruction from a noisy cloud is its own research area. The alternative is to move one cell left on Figure 16.4 and have the decoder emit a **parameterisation** rather than a list.

AtlasNet makes the change with almost no machinery. Instead of a decoder $z \mapsto \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$ producing a fixed number of points, take a decoder

$$\mathbf{p} = \mathrm{MLP}(z, u, v), \qquad (u, v) \in [0, 1]^2, \tag{16.12}$$

that maps a shape code together with a coordinate in the unit square to a point in space. The network is now exactly the parametric surface of Section 16.3, with a learned map in place of a Bézier polynomial. A single square cannot cover a shape of arbitrary topology, so the model uses K of them with separate decoders — the papier-mâché of its title, strips of paper laid over the object — and the union of their images is the surface.

Three things fall out that a point decoder cannot give. The output resolution is unbounded: sample as many (u, v) pairs as you want at inference, from a network trained on far fewer. The output has connectivity for free, because adjacency in the unit square is inherited by its image, so triangulating the square yields a mesh with no reconstruction step. And the parameterisation itself is useful downstream — it is a texture atlas, which is what a renderer wants and what a point cloud has never been able to supply.

The comparison the lecture draws is between the three explicit decoders side by side on the same input image: a voxel grid, blocky and resolution-limited by the cubic cost of Section 16.8; a point cloud, unstructured and unrenderable; and an AtlasNet surface, smooth and directly

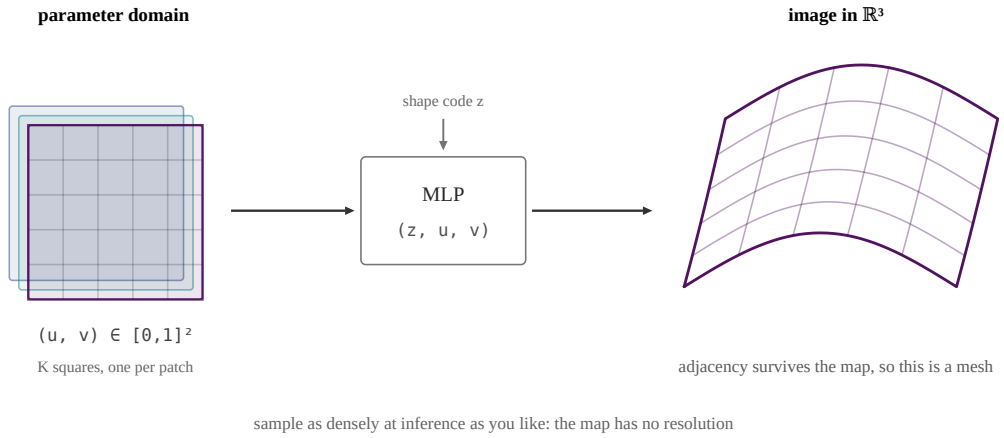


Figure 16.10.: The AtlasNet decoder. A shape code and a coordinate in the unit square go in and a point in space comes out, so the square's own triangulation is carried onto the surface and the output is a mesh rather than a cloud. Several squares are used because one cannot cover arbitrary topology, and the seams between their images are the method's characteristic artefact.

usable. The trend across them is toward representations that carry more structure per stored number, and the next step continues it by removing the stored numbers altogether.

16.11. The network is the field

Strip a voxel grid down to what it actually asserts and it is a table of answers to one question: given a point, is it inside the object? Section 16.5 arrived at voxels precisely by precomputing that query on a lattice. The realisation of 2019 was that the precomputation is the mistake. If the question is *given a point, what is here*, then a network can answer it directly — take coordinates in, give a value out — and the lattice, its cubic cost and its resolution ceiling all disappear at once.

Four papers reached that conclusion within months of each other, differing mainly in what the network outputs. Occupancy Networks and IM-Net predict a binary occupancy, so the surface is the decision boundary of a classifier. DeepSDF predicts the signed distance of Equation 16.8, so the network carries the magnitudes as well as the sign. In every case the model is a function

$$f_\theta : \mathbb{R}^3 \times \mathbb{R}^d \to \mathbb{R}, \qquad (\mathbf{x}, z) \mapsto \text{occupancy or } d(\mathbf{x}), \tag{16.13}$$

conditioned on a latent code z that selects which shape of the learned family is being queried, and the surface of that shape is the level set of Equation 16.4 for the function $f_\theta(\cdot, z)$.

What this buys is the property no discrete representation can have: the model is **resolution-free**. There is no grid, so there is no maximum detail decided at training time; extracting a mesh at 512^3 from a network trained on sampled queries costs only more queries, and memory scales with the network's parameters rather than with the cube of the resolution. NeRF's own comparison makes the size argument vivid — a competing method that stored a voxel grid per input image needed over 15 GB for a single scene, where the network weights came to 5 MB, a factor of

roughly three thousand. A network is a very compact way to hold a function that a grid holds badly.

The cost is that every query is a forward pass, and turning the field back into geometry means running marching cubes over a great many of them. It is a representation optimised for *asking*, which is the containment query of Section 16.4, and it is why implicit methods were positioned exactly right for what came next.

Composition survives the move to networks, too. Local deep implicit functions and their predecessors represent a shape as a *collection* of small implicit functions with local support rather than one global field, recovering the CSG idea of Equation 16.7 with learned rather than authored primitives, and gaining locality — editing one region no longer perturbs the whole object.

16.12. Geometry and appearance in one function

A deep implicit function trained on occupancy needs occupancy labels, which means 3D supervision, which returns us to Figure 16.5 and the shortage it describes. NeRF makes two changes to Equation 16.13 and between them they remove the requirement entirely.

The first is to widen what the field carries. Instead of one scalar, the network takes a position **and a viewing direction** and returns a volume density together with a colour:

$$F_\theta : (\mathbf{x}, \mathbf{d}) \mapsto (\sigma, \mathbf{c}), \qquad \mathbf{x} \in \mathbb{R}^3,\ \mathbf{d} \in \mathbb{S}^2. \tag{16.14}$$

The density σ depends on position alone — geometry does not change with where you stand — while the colour may depend on direction, which is what lets the model reproduce specular highlights that move as the camera does. Nothing here is a new kind of object; it is the implicit representation of Section 16.5 extended from geometry to appearance, exactly as Section 16.4 predicted the containment query would demand.

The second change is what makes it trainable. March a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ from the camera through the scene, sample N points along it, and composite them front to back:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i\, \alpha_i\, \mathbf{c}_i, \qquad \alpha_i = 1 - e^{-\sigma_i \delta_i}, \qquad T_i = \prod_{j=1}^{i-1} (1 - \alpha_j), \tag{16.15}$$

where δ_i is the distance between consecutive samples, α_i is the opacity that a segment of length δ_i at density σ_i contributes, and T_i is the **transmittance** — the fraction of light that survives everything nearer the camera. The product of the two, $w_i = T_i \alpha_i$, is how much this sample contributes to the pixel.

This is standard volume rendering from graphics, and the only property NeRF needs from it is that every operation in Equation 16.15 is differentiable with respect to σ_i and $\mathbf{c}_i$, which are the network's outputs. So the loss can be squared error between $\hat{C}$ and the *actual pixel of an actual photograph*, and gradients flow back through the compositing sum into the field. Training data is a set of images with known camera poses. No 3D ground truth is involved at any point.

The weights have a structure worth naming, because it is what makes the sum behave like a rendering rather than an arbitrary weighted average. Since $T_{i+1} = T_i(1 - \alpha_i)$, each weight is a difference of consecutive transmittances, $w_i = T_i - T_{i+1}$, and summing telescopes:

$$\sum_{i=1}^{N} w_i \;+\; T_{N+1} \;=\; 1. \tag{16.16}$$

The weights and the light that escapes out the back of the volume partition unity exactly. That is why an opaque surface produces weights concentrated on one sample, why an empty ray produces $T_{N+1} \approx 1$ and a background colour, and why the compositing can never manufacture or lose energy. The generator behind Figure 16.11 evaluates Equation 16.16 on a sampled ray and asserts it to machine precision. It also checks the sum against a closed form, and that check returns something slightly better than convergence: for a slab of *constant* density the discrete sum equals $1 - e^{-\sigma L}$ exactly, at any number of samples, because the transmittance product telescopes into a single exponential. The discretisation error in Equation 16.15 therefore comes entirely from density varying between samples, which is why NeRF's hierarchical sampling — spending samples where the weights are large — buys as much as it does.

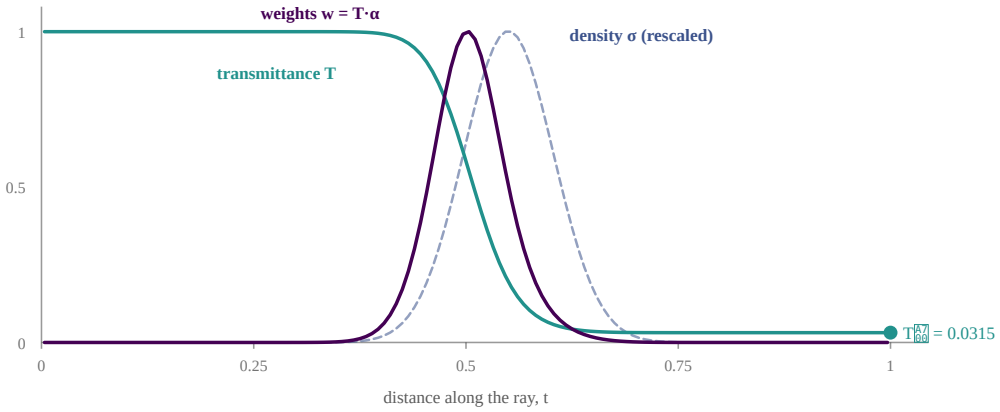


Figure 16.11.: Volume rendering along one ray through a soft-edged surface. Density rises and falls across the surface; transmittance T_i decays monotonically as light is absorbed; the compositing weights $w_i = T_i\alpha_i$ peak *before* the density does, because transmittance has already begun to fall by the time the density reaches its maximum. That single narrow band of weight is what makes the sum behave like a surface intersection without anyone having to find one. All three curves are evaluated by the generator from the same density profile, and the weights are checked against Equation 16.16.

The consequence reaches beyond reconstruction. The generative pipeline of Section 16.8, which paired a voxel generator with a differentiable projection, can have its voxels replaced by a radiance field — this is what -GAN and its relatives do — and the resulting model learns 3D-aware image synthesis from *photographs alone*, with camera viewpoint and object identity as separate controls. Categories no longer need a 3D dataset to be modelled in 3D. That is the payoff Section 16.6 was pointing at.

16.13. Back to explicit, for speed

NeRF's weakness is visible in Equation 16.15. Producing one pixel requires evaluating a network at every sample along a ray, and the samples are placed along the whole ray because nothing tells the renderer in advance where the object is. Most of them land in empty space and return a density near zero, which is the same waste that Section 16.8 diagnosed in voxel grids, moved from memory into compute. The original method takes on the order of tens of seconds per frame.

3D Gaussian splatting fixes it by changing where the scene is parameterised. NeRF defines its field **densely** — at every point in space, whether or not anything is there. Gaussian splatting defines it **sparsely**, storing a set of anisotropic 3D Gaussians, each with a position, a covariance, an opacity and a view-dependent colour, and treating the space between them as empty by definition. This is a point cloud whose points have extent: explicit and non-parametric, back in the top-left cell of Figure 16.4, after the field spent a decade travelling to the opposite corner.

Rendering then becomes projection rather than search. Each Gaussian projects to an ellipse in the image, the ellipses covering a pixel are sorted by depth and alpha-composited with the same Equation 16.15 arithmetic, and no sample is ever spent on a region containing nothing. The representation is optimised by gradient descent through that rasteriser, starting from the sparse point cloud that structure-from-motion already produces during camera calibration, with Gaussians split, cloned and pruned as the fit demands.

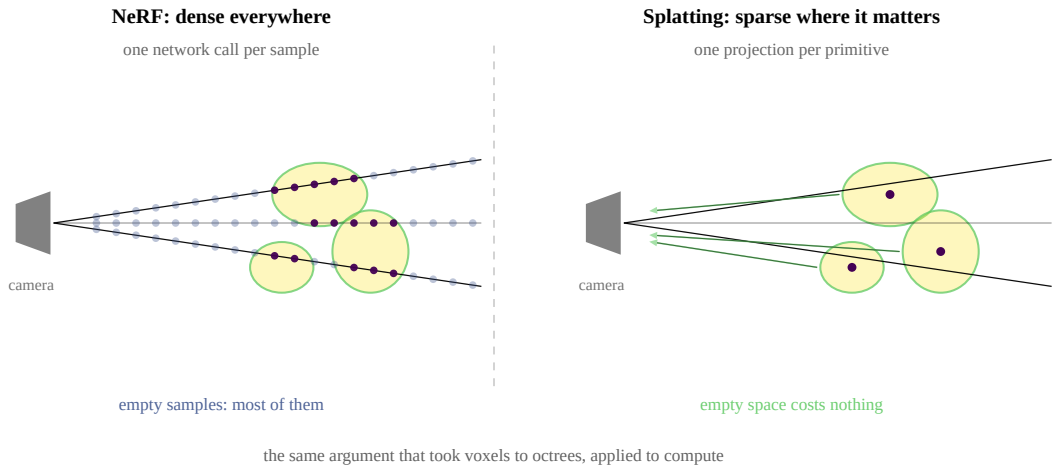


Figure 16.12.: Two ways to say where the scene is. NeRF queries a network at samples spread along the whole ray, so most evaluations land in empty space; splatting stores primitives only where density is nonzero and projects them, so the empty space costs nothing. This is the same argument that took voxels to octrees in Section 16.8, applied to compute rather than memory.

The numbers are the reason the method took over within a year. Against Mip-NeRF360, the strongest quality-oriented radiance-field method at the time, splatting reports higher PSNR — 25.2 against 24.3 — while training in 51 minutes rather than 48 hours and rendering at 93 frames per second rather than 0.071. That last ratio is a factor of about 1,300, and it is the difference between a research artefact and something that runs in a browser. A shorter-trained configuration reaches 135 frames per second at 23.6 PSNR, still ahead of the fast baselines it is compared against.

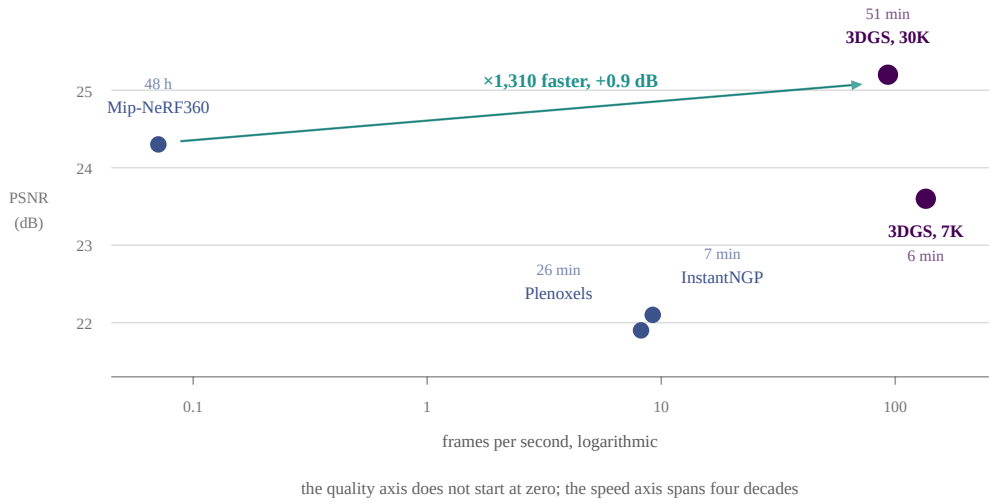


Figure 16.13.: Rendering speed against quality, from the numbers 3D Gaussian splatting reports for itself and four baselines; the label on each point is training time. Note the logarithmic speed axis, which spans four orders of magnitude, and that the quality axis does not start at zero — the PSNR differences are real but small next to the speed differences, which is the whole claim.

Read against Figure 16.4, the arc is not a march toward implicit representations at all. It is a circuit. Voxels were chosen because a convolution needed a grid; points were chosen because the grid wasted its cells; implicit fields were chosen because the containment query is what rendering asks; and splats were chosen because the field wasted its queries. Each move fixed the previous one's dominant cost, and the destination happens to sit next to the starting point with a great deal learned in between.

16.14. What geometry leaves out

Every representation so far — points, meshes, fields, splats — describes a shape as an undifferentiated surface. None of them records that a chair's four legs are copies of one another, that its left and right halves are mirror images, that the seat rests on the base rather than floating near it, or that a bed in a bedroom is against a wall. These regularities are not incidental decoration on the geometry. They are most of what makes an object recognisable as a designed thing, and a generative model that does not represent them has to rediscover them, imperfectly, in every sample it draws.

A **structure-aware** representation splits a shape into *element geometry* — the surface of each part, handled by anything in this chapter so far — and *element structure*, which is what the parts are and how they relate. The design space for the second half is a ladder, and each rung buys a guarantee and costs something.

At the bottom is **segmented geometry**: an ordinary shape with per-region labels. It is trivial to build from existing models and it guarantees nothing, because a generative model producing labelled geometry must learn to keep its segments coherent and may not. **Part sets** promote the parts to first-class objects — volumetric primitives or small implicit functions assembled into

a whole — which makes each part valid by construction, at the price of saying nothing about how they sit together; nothing stops a leg from floating.

Relationship graphs add edges for adjacency, support and symmetry, so connectivity can be enforced rather than hoped for, but generating graphs is an open problem in a way that generating sequences and trees is not. **Hierarchies** trade the other way: a tree is much easier to generate, and matches how designed objects decompose — a room contains chairs, a chair has a base, the base has legs — but not every relationship is naturally a tree, and symmetry between siblings is exactly the kind that is not. **Hierarchical graphs** combine them, keeping a tree for the decomposition and a small graph at each level for the lateral constraints, which is StructureNet's design and needs a graph network that can encode and decode variable-degree nodes. Its cost is annotation: data in this format is expensive, and PartNet exists largely because it did not otherwise.

At the top are **programs**. A shape described by the code that constructs it subsumes every other rung, since a program can emit any of them, and it expresses the object's real degrees of freedom as free parameters — the number of legs, the height of the seat — which is exactly what a parametric representation was for in Section 16.3, now generalised beyond a fixed formula. It is also the hardest format to obtain data in, because almost nothing in the world's 3D corpora ships with the program that made it.

The ladder is unresolved, and the lecture presents it as such. What it establishes is that the taxonomy of Figure 16.4, which organises the whole of this chapter, is a taxonomy of *geometry* — and that a full account of a 3D object needs a second axis the chapter has barely touched. Every method in Section 16.7 through Section 16.13 improved how well a surface can be stored, queried and rendered. None of them made a model that knows a chair has four legs because chairs do.

16.15. The map is the result

The first half of this chapter is older than deep learning and reads, at first, like background. It is not. Every architectural decision in the second half is a choice of cell in Figure 16.4, and every one of them was made for a reason stated in Section 16.4: some query got cheap, or some query got expensive.

The reason it is worth carrying the map rather than the list is that the list keeps changing and the map does not. Voxels lost to points because a grid pays $O(n^3)$ for an $O(n^2)$ object. Points lost to parametric decoders because a set has no connectivity and a renderer needs one. Parametric decoders lost to implicit fields because rendering asks *what is at this point*, which is the one question an explicit representation cannot answer cheaply. Implicit fields lost — for real-time rendering, not everywhere — to splats because answering that question everywhere wastes most of the answers. Four moves, four different dominant costs, and each successor sitting in whichever cell made the current bottleneck disappear.

Two things follow that are worth holding onto past the specific methods. The first is that rendering is the load-bearing bridge, not an output stage. It appears in Section 16.7 as a way to reach ImageNet pretraining, in Section 16.8 as a differentiable projection carrying an adversarial loss, and in Section 16.12 as the entire supervision signal — and each time the underlying reason is Figure 16.5, which shows a 3D data supply that has not caught up and probably will not. A method that can be trained on photographs has access to a corpus that no amount of scanning will match.

The second is that geometry was never the whole problem. Section 16.14 is two minutes of an eighty-minute lecture and it names the thing that none of the preceding representations do: the regularities that make an object a designed object rather than a surface. A splat cloud can reproduce a chair to a fraction of a millimetre and still not encode that it has four legs. Whatever comes after the circuit described here is likely to be judged on that axis rather than on PSNR.

Part VI.

Multimodal

17. Vision and Language

Lecture 16

Based on Lecture 16 of CS231n, Stanford University, Spring 2025, given by Ranjay Krishna.

17.1. One model, or one model per task

Every chapter so far has described the same loop, and described it so consistently that the loop stopped looking like a choice. Collect a dataset with a training split and a test split. Train a model on the training split for the one task the dataset was built to measure. Report a number on the test split. Detection, segmentation, captioning, video classification — each got its own dataset, its own architecture, its own training run, and its own leaderboard, and a model that was excellent at one of them was, in the strict sense, useless at any other. The weights encoded a task as much as they encoded the world.

What changed is not that anyone found a better architecture for image classification. What changed is the observation, arriving from language first, that the expensive part of training is learning what things *are*, and the cheap part is learning what to *do* with them. If the expensive part can be paid once, on data that is not organised around any particular task, then the cheap part can be paid many times over, in some cases with a handful of examples and in some cases with none at all. A model trained that way is called a foundation model, and the name is doing real work: it is a claim about position in a pipeline rather than about architecture.

The definition is contested at the edges, and the lecture is honest about this. The one property everyone agrees on is generality — the same weights serve many downstream tasks without being retrained for each. The properties that usually accompany it but do not define it are the familiar three: a large parameter count, a large and diverse pretraining corpus, and a self-supervised objective. Those three are means rather than ends. A model can have all of them and remain a specialist, and the interesting question about any candidate foundation model is not how large it is but how little you have to do to it before it is useful for something its authors did not anticipate.

Language got there first, and got there for a reason worth stating plainly, because it is the constraint that everything in this chapter is working around. Text is *self-labelling*. The next word in a sentence is both the input to the model and the supervision for it, and there is no annotation step between a web crawl and a training corpus. That single property is what let language models scale to corpora nobody could have labelled, and it is exactly what vision lacked. Pixels do not come with their next pixel in any semantically useful sense, and the self-supervised objectives of Section 13.9 — asking two augmentations of one image to agree — produce representations without producing an *interface*. A SimCLR encoder maps a photograph to a good vector. Nothing about that vector tells you what the photograph is of, and recovering that still needs a labelled dataset and a supervised head.

So the vision story of the last five years is best read as a search for something that plays the role that next-word prediction plays in language: a supervision signal that already exists at web scale, that nobody has to pay for, and that carries semantics rather than just invariance. The answer turned out to be text — not as the output of the vision model but as its *supervisor*.

17.2. Text as the label that already exists

Recall the shape of the SimCLR objective. Take an image, produce two augmentations of it, push their representations together, and push them away from the representations of every other image in the batch. The positive pair is manufactured — it exists because the augmentation pipeline made it — and the negatives are whatever else happened to be in the minibatch. The model learns to be invariant to crops, colour jitter and blur, which is a real and useful thing to learn, but the semantics come in only by implication: two crops of a cat share a label because they share an origin, not because anything in the loss knows the word *cat*.

CLIP changes exactly one thing about that picture, and the change is small enough to state in a sentence and consequential enough to have reorganised the field. Keep the image encoder on the left. Replace the second image encoder on the right with a *text* encoder. The positive pair is no longer two augmentations of one photograph; it is a photograph and the caption that was published alongside it. Everything else — the contrastive form, the in-batch negatives, the temperature — carries over unchanged.

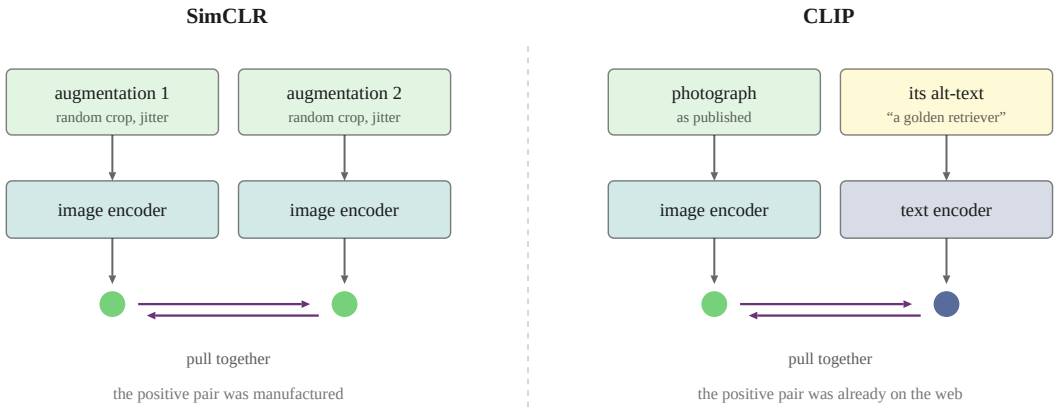


Figure 17.1.: The move from SimCLR to CLIP. Both objectives pull a positive pair together and push everything else in the batch apart; they differ only in where the positive pair comes from. SimCLR manufactures it with augmentation, so the model learns invariance. CLIP takes it from the web, so the model learns to associate an image with a description — and inherits an interface it can be queried through.

Write the batch as N pairs (x_i, y_i) of image and caption, and let $u_i = g_{\text{img}}(x_i)$ and $v_i = g_{\text{txt}}(y_i)$ be the two encoders' outputs after projection to a shared space and normalisation to unit length, so that $\langle u_i, v_j \rangle$ is a cosine similarity in $[-1, 1]$. The batch defines an $N \times N$ matrix of these similarities, and the entire training signal is the claim that its diagonal should dominate. Reading the matrix along its rows gives the image-to-text loss,

$$\mathcal{L}_{\text{I}\to\text{T}} = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp(\langle u_i, v_i \rangle / \tau)}{\sum_{j=1}^{N} \exp\big(\langle u_i, v_j \rangle / \tau\big)}, \tag{17.1}$$

which asks each image to select its own caption out of the N on offer. Here τ is a learned temperature that sets how sharply the softmax discriminates: small τ makes the loss care intensely about the single closest competitor, large τ spreads the pressure across all of them. Reading the same matrix along its columns gives the complementary constraint, that each caption should select its own image,

$$\mathcal{L}_{\mathrm{T}\to\mathrm{I}} = -\frac{1}{N}\sum_{j=1}^{N} \log \frac{\exp\big(\langle u_j, v_j\rangle/\tau\big)}{\sum_{i=1}^{N}\exp\big(\langle u_i, v_j\rangle/\tau\big)}, \tag{17.2}$$

and the objective actually optimised is the average of the two. The symmetry is not decoration. A one-sided loss can be driven down by an encoder that spreads images out and collapses text, or the reverse; requiring both directions to work forces both spaces to be informative.

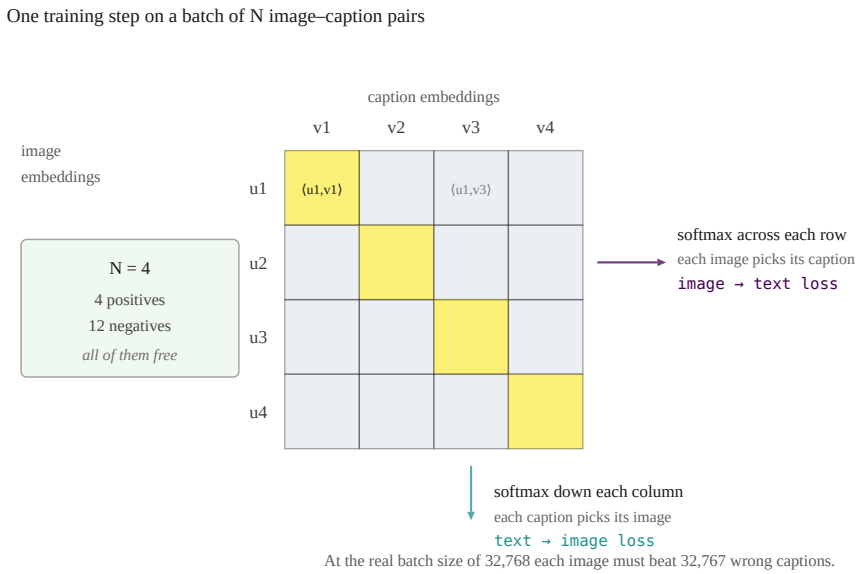


Figure 17.2.: CLIP's training step in full. A batch of N image–caption pairs produces an $N \times N$ matrix of cosine similarities; the N diagonal entries are the positives and every off-diagonal entry is a negative. Cross-entropy along the rows is Equation 17.1, along the columns Equation 17.2, and the loss is their average. Every negative is free — it is another example's positive, reused.

Two things about Figure 17.2 are worth dwelling on, because they explain why this objective and not a more obvious one. The first is that the negatives cost nothing. A batch of N pairs yields N positives and $N^2 - N$ negatives, all of them assembled by matrix multiplication from encodings that had to be computed anyway. This is why batch size is a first-class hyperparameter rather than a memory-management detail: CLIP trains at a minibatch of 32,768, and the difficulty of the discrimination problem — how many wrong captions each image has to beat — grows with it. The second is that the loss never asks either encoder to *reconstruct* anything. It only asks them to agree on an ordering. That is a far weaker requirement than generation, it is what makes the objective cheap, and in Section 17.5 it turns out to be exactly what the model exploits.

The payoff is the data. Alt-text is attached to images across the web for reasons that have nothing to do with machine learning, and it was already there. CLIP's corpus is 400 million image–text pairs collected from public sources, and no one annotated any of it. This is the property that self-supervised vision objectives were reaching for and could not get: not merely unlabelled data, but data whose labels are *semantic* and *free*. The augmentation-based objectives

could scale, but everything they scaled was invariance. Adding text to the same machinery scales meaning.

17.3. A similarity model is already a classifier

What comes out of training is not a classifier. It is a function that scores how well an image and a piece of text go together, which is an odd artefact if what you wanted was ImageNet accuracy. The move that turns one into the other is simple enough to feel like a trick and is, in a way that matters, not a trick at all.

Suppose you want to classify images into C categories. Write each category name into a short phrase — `a photo of a {label}` — and push all C phrases through the text encoder. That gives C unit vectors $v_1, \dots, v_C$, one per class, computed once and cached. To classify a new image, encode it to u and take the class whose vector it is closest to:

$$\hat{y}(x) = \arg\max_{c \in \{1,\dots,C\}} \langle u(x),\, v_c \rangle. \tag{17.3}$$

The lecture describes this as nearest-neighbour classification with one exemplar per class, where the exemplars happen to have been written rather than photographed, and that is a good way to hold it. But it is worth noticing what Equation 17.3 is in more familiar terms. Stack the v_c as the rows of a matrix $W \in \mathbb{R}^{C \times d}$; then $\langle u, v_c \rangle$ is the c-th component of Wu, and the prediction is $\arg\max_c (Wu)_c$. This is *exactly* a linear classification head with no bias term, of the kind every chapter since Section 3.4 has been training with labelled data and gradient descent. CLIP does not avoid the linear head. It synthesises the weights from language instead of fitting them to examples, and the figure below makes the equivalence explicit because it is the cleanest statement of what zero-shot transfer actually is.

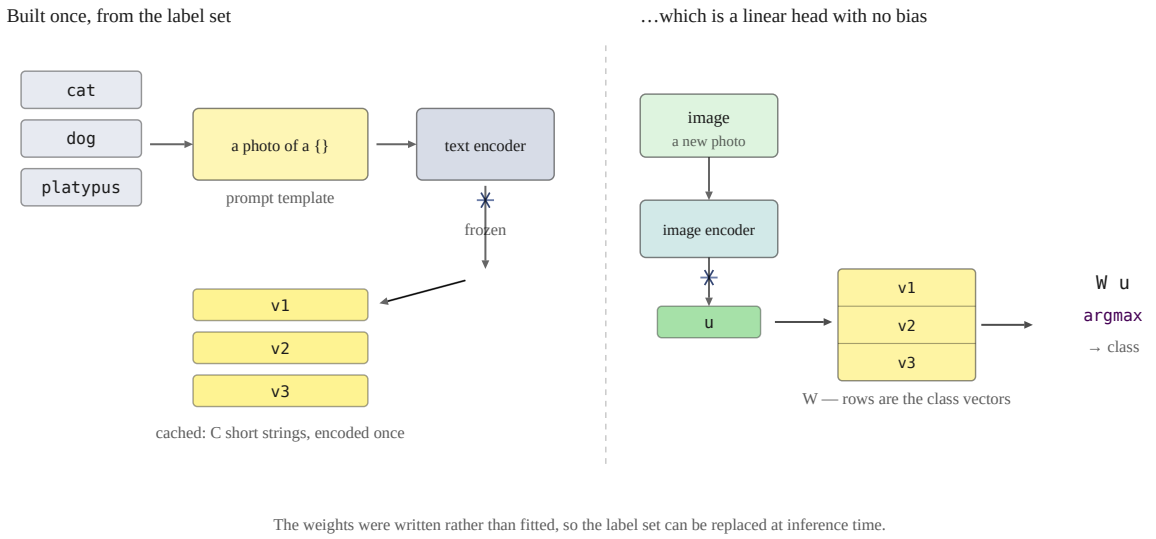


Figure 17.3.: Constructing a zero-shot classifier. The class names are written into prompts, encoded once, and cached; classifying an image is a single matrix–vector product against them. The right-hand panel is the same computation drawn as the bias-free linear layer it is — the weights were written rather than trained, and the classifier can be rebuilt for a new label set in the time it takes to encode C short strings.

Reading it that way also explains the property that makes it useful. A trained linear head is welded to the label set it was fitted on; changing the classes means collecting examples and refitting. A *written* head can be rebuilt for any label set at all, including categories that appeared nowhere in training and including label sets invented at inference time. The vocabulary of the classifier is open, and its cost is a forward pass over a few short strings.

The phrasing of those strings turns out to matter more than it should, and the reasons are instructive. The text encoder was trained on captions, which are sentences, so feeding it a bare token like `boxer` puts it well outside the distribution it learned on — and lands it on a genuine ambiguity, since the word is a dog breed and an athlete and CLIP has seen both. Wrapping the label in `a photo of a {label}` recovers the caption-like form and disambiguates by context, and adding a domain hint — `a satellite photo of a {label}`, `a sketch of a {label}` — helps further on datasets where the images are not ordinary photographs.

Any single template also bakes in a bias, and the fix generalises the trick rather than tuning it. Encode the class through many templates and average the resulting vectors, renormalising the result to unit length:

$$\bar{v}_c = \frac{\frac{1}{K}\sum_{k=1}^{K} t\big(\text{prompt}_k(c)\big)}{\Big\|\frac{1}{K}\sum_{k=1}^{K} t\big(\text{prompt}_k(c)\big)\Big\|}. \tag{17.4}$$

CLIP ensembles 80 context prompts on ImageNet this way, worth 3.5 percentage points over the single default template, with prompt engineering and ensembling together contributing almost five points. The detail that makes it practical is *where* the averaging happens. Ensembling in probability space would mean running K classifiers per image and averaging their outputs, so inference cost would scale with K. Averaging in embedding space collapses the K templates into one cached vector per class before any image arrives, so the ensemble costs K text encodings once and nothing at all per image thereafter. The two are not the same computation and do not give the same answer — Equation 17.4 commits to a single direction per class rather than to a mixture — but the one that is free is also the one that works.

17.4. What the zero-shot number was hiding

Trained on 400 million pairs, CLIP reaches 76.2% zero-shot top-1 on ImageNet, matching the original ResNet-50 while using none of ImageNet's 1.28 million labelled training images. That is a striking sentence and, taken alone, a slightly boring result: a great deal of machinery to arrive where supervised training arrived years earlier. The interesting part is what happens when a supervised model matched to the same 76.2% — the paper's comparison uses a ResNet-101 — is asked about anything else.

The benchmark suite for this measures *distribution shift*: datasets built from the same thousand ImageNet categories, rendered in ways the original photographs were not. ImageNetV2 is a fresh collection following the original protocol, ImageNet Sketch replaces photographs with line drawings, ImageNet-R with artistic renditions, ObjectNet photographs household objects at unusual angles and in unusual settings, and ImageNet-A collects natural photographs that standard classifiers get wrong. Nothing in any of them requires a concept the model has not been asked about. Only the pixel statistics move.

The gaps in Figure 17.4 are not the usual few points of benchmark noise. On ImageNet-R the supervised model scores 37.7% and CLIP 88.9%. On ObjectNet, 32.6% against 72.3%. On ImageNet-A, 2.7% against 77.1% — a supervised classifier performing several times *worse* than

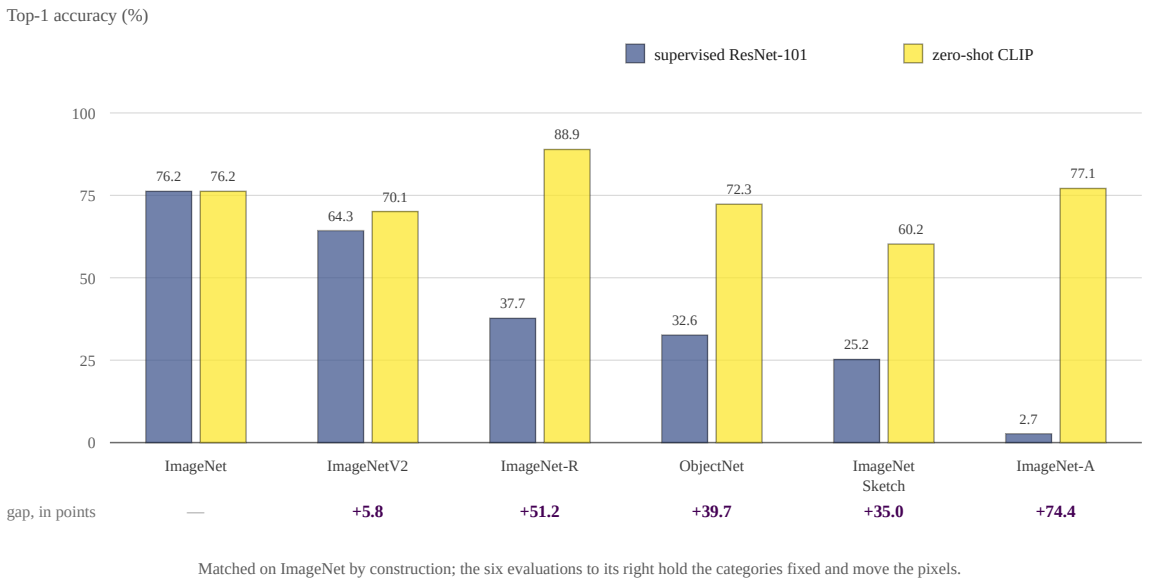


Figure 17.4.: Two models matched exactly on ImageNet and separated by up to 74 points everywhere else. Bars are top-1 accuracy for a supervised ResNet-101 and for zero-shot CLIP on the six evaluations; the annotation on each pair is the gap. The supervised model's accuracy tracks how closely a dataset resembles ImageNet's photographs, which is the definition of having learned the dataset rather than the categories.

random guessing among a thousand classes would, because the images were selected adversarially against exactly that kind of model. The supervised network's accuracy degrades in near-perfect proportion to how far a dataset drifts from ImageNet's particular photographic conventions, which is the operational definition of having fitted the dataset rather than the categories. Its 76.2% was never a measurement of whether it knows what a teapot is; it was a measurement of whether it knows what an ImageNet teapot looks like. CLIP, having never seen ImageNet's training distribution, has nothing of that kind to overfit.

This is the result that ended the argument, and it did so by reframing what the argument was about. The case for supervised pretraining had always been that labels are the most informative signal available per example, which is true. The case against is that a fixed label set caps what can be learned per example at $\log_2 1000 \approx 10$ bits, and worse, that it teaches the model the *photographer's* conventions along with the category. Captions are noisier per example and unbounded in what they can express, and there are three orders of magnitude more of them.

The scaling evidence points the same way. CLIP's zero-shot ImageNet accuracy rises steadily with both model size and data volume across the range they trained, with no sign of the saturation that supervised ImageNet training hits once the labels are exhausted — the ceiling in the supervised case is a property of the dataset, not of the method.

CoCa then showed that the contrastive objective, for all its efficiency, is leaving something on the table. Its architecture keeps CLIP's two towers and splits the text side in two: the first block of decoder layers has no cross-attention and produces a unimodal text embedding, which is contrasted against the image embedding exactly as before, and the remaining layers cross-attend to the image features and are trained to *generate* the caption. One forward pass, two losses. The captioning term is a strictly harder requirement — predicting each word given the image and the preceding words demands that the image features retain the information a description

needs, not merely enough to be ranked above the other images in a batch. CoCa reaches 86.3% zero-shot top-1 on ImageNet, 90.6% with a frozen encoder and a learned head, and 91.0% after fine-tuning. Contrastive pretraining is cheap and generative pretraining is informative, and the two compose.

17.5. The failure the objective permits

It would be easy to read the previous section as a straightforward win, and the lecture is careful not to let it stand that way. CLIP-style models are cheap to train relative to what they can do, trivial to retarget at a new label set, and robust in a way that supervised classifiers are not. They are also, in a specific and well-documented sense, not doing what their zero-shot numbers suggest.

Start from what Equation 17.1 actually demands. To drive the loss down, the image encoding of a photograph must score higher against its own caption than against the $N - 1$ other captions *in that batch* — and those captions are drawn at random from the web. A batch containing a golden retriever, a plate of sushi and a suspension bridge poses a problem that is completely solved by detecting which nouns are present. Nothing about word order, nothing about which adjective attaches to which noun, and nothing about the relation between two objects is ever load-bearing, because no random negative ever differs from the positive on those axes alone. The gradient has no reason to encode structure it is never tested on, and it does not.

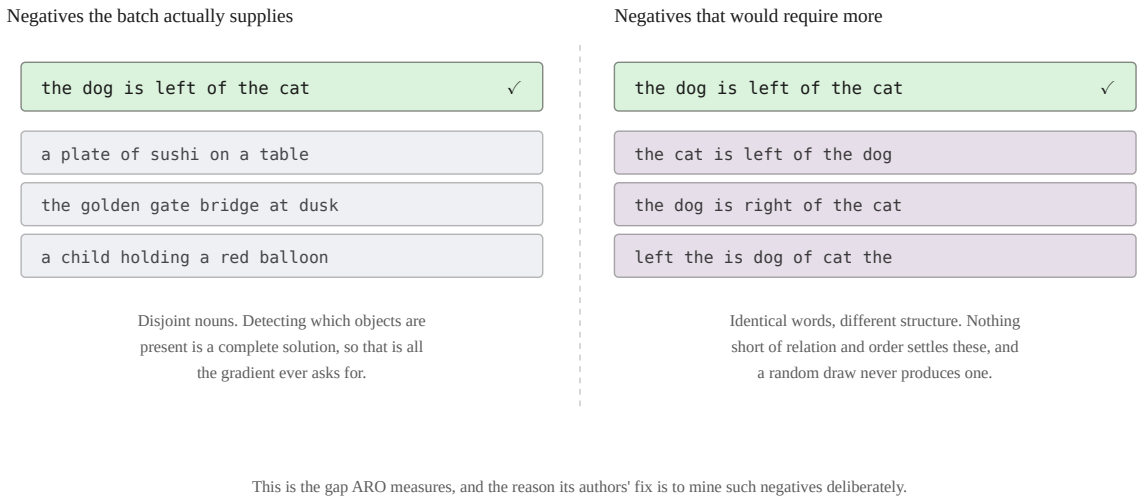


Figure 17.5.: Why random negatives never force compositional structure. Against the batch on the left, matching nouns is a complete solution and every gradient step rewards it. The hard negatives on the right — same words, different structure — are the ones that would require more, and the probability of drawing one from a web-scale corpus at random is effectively zero.

The ARO benchmark makes this precise by constructing the negatives the training distribution never supplies. Its cases are built from Visual Genome and COCO by perturbing a true caption in exactly one structural way — swapping the arguments of a relation, reassigning an attribute to the wrong object, or shuffling word order — so that the correct and incorrect captions contain the same words and differ only in how those words compose. Across more than fifty thousand such cases, state-of-the-art vision–language models behave like bags of words: relational understanding is poor, attribute-to-object binding blunders, and order sensitivity is close to

absent. The authors' diagnosis is the same as the one above, and their fix follows from it — mine hard negatives that differ compositionally, so that the objective finally has to represent what it was never asked about.

The related Winoground probe is smaller and sharper: two images and two captions built from *identical* words in different orders, where the task is to match them correctly. Models that score respectably on ImageNet perform at or below chance on it.

The other limitation is structural rather than statistical, and it constrains everything the rest of the chapter builds. CLIP emits one vector per image. Whatever grounding the model has — which pixels correspond to which word — is destroyed by the pooling that produces that vector, and there is no output head through which it could be recovered. The model can tell you a photograph is consistent with `two dogs on a sofa`. It cannot tell you where either dog is, and it is unreliable about whether there are two, because counting requires exactly the compositional structure the objective declined to learn. A single similarity score is a narrow interface, and the next three sections are each a different answer to the question of what to attach to a vision encoder so that something richer can come out.

17.6. Fusion by concatenation

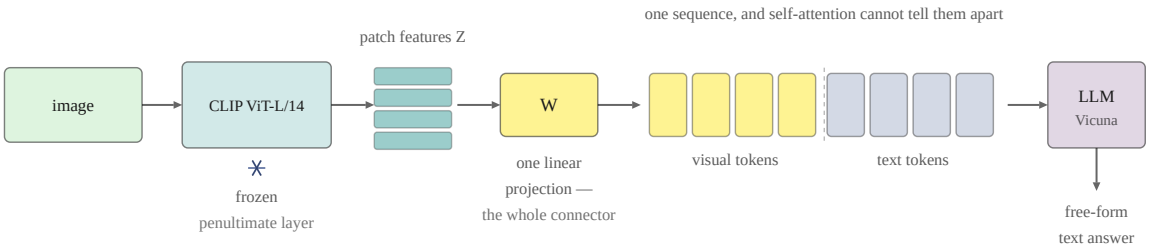
If a single vector is too narrow an interface, the obvious thing to want is a model that takes an image and a question and answers in free text. By 2023 one half of that was solved — instruction-tuned language models could hold a conversation, follow directions, and reason in text — and the other half, a vision encoder producing semantically rich features, had been solved by CLIP. The question was how to join them, and there are two good answers. LLaVA takes the simpler one.

Recall that a transformer language model does not consume words. It consumes a sequence of vectors in $\mathbb{R}^d$, produced by looking each token up in an embedding table, and self-attention is indifferent to where in the sequence any particular vector came from. That indifference is the opening. If image content can be written as a handful of vectors in that same d-dimensional space, it can be *concatenated into the prompt* and the language model will attend to it exactly as it attends to words — with no architectural change whatsoever.

LLaVA does precisely that. A frozen CLIP ViT-L/14 encodes the image into patch features Z; a single trainable projection W maps each patch feature into the language model's embedding dimension, giving visual tokens $H_v = WZ$; those tokens are prepended to the embedded text tokens; and the whole sequence goes into Vicuna, which generates the response autoregressively. The connector is one matrix.

Two choices inside that simplicity carry weight. The first is which layer of the vision encoder to read. The final layer of a CLIP image encoder is trained to produce the single pooled vector that enters Equation 17.1, which means it has been optimised to discard everything not needed for that ranking — precisely the spatial detail a question-answering model wants. The penultimate layer's patch features have not yet been through that bottleneck, and LLaVA uses them. It is the same instinct as taking features from before the classifier in transfer learning, applied to a bottleneck that is contrastive rather than categorical.

The second is where the training data comes from, and it is the part that generalised. There was no corpus of multimodal instruction-following conversations, and collecting one by hand is expensive. So LLaVA manufactured it: take images from COCO with their existing captions and bounding boxes, feed *those text annotations* — not the images — to a language-only GPT-4, and ask it to write conversations, detailed descriptions, and reasoning questions about the scene



Stage 1 trains W alone; stage 2 trains W and the language model on GPT-4-written instruction data.

Figure 17.6.: LLaVA's fusion. Patch features from a frozen CLIP encoder are mapped by a single linear projection into the language model's embedding space and concatenated with the text tokens, so self-attention mixes the two modalities with no change to the language model's architecture. The connector is the only new parameter block in stage one.

it can infer from them. The teacher never sees a pixel; the symbolic annotations stand in for the image, and the student learns to produce the same responses from the image itself. Training then runs in two stages, first aligning the projection with the language model frozen, then fine-tuning the projection and the language model together on the generated instruction data. The result reaches 85.1% of GPT-4's score on a synthetic multimodal instruction benchmark and, ensembled with GPT-4, 92.53% on ScienceQA.

The honest caveat is that this is distillation. The ceiling of a model trained this way is set by the teacher that wrote its training data, which is the specific problem Section 17.8 exists to solve.

17.7. Fusion by cross-attention

Flamingo answers the same question differently, and the differences are motivated by a goal LLaVA does not pursue: keeping the language model *exactly* intact while teaching it to see, so that its in-context learning ability survives the operation.

The first problem is quantity. A vision encoder's output grid varies with resolution, and video makes it worse — frames sampled at one per second turn a clip into a spatio-temporal grid whose size depends on duration. Cross-attention over that is expensive and its cost is unpredictable. Flamingo inserts a **Perceiver Resampler**, which holds a fixed set of learned latent queries that cross-attend to the visual features and emit a fixed 64 visual tokens no matter how many features came in. Variable-size input, constant-size output, with the compression learned rather than imposed by pooling — and in ablation it beats both a plain transformer and an MLP in that position.

The second problem is the one that dictates the architecture. Flamingo's language model is a frozen Chinchilla, and freezing it is the point: its few-shot ability is what Flamingo is trying to inherit, and fine-tuning would put that at risk. But new layers initialised at random, inserted between pretrained blocks, corrupt the residual stream immediately — the first forward pass injects noise into representations the frozen model spent its entire training run learning to rely on.

The fix is a gate. Each inserted `GATED XATTN-DENSE` block multiplies its output by $\tanh(\alpha)$ before adding it to the residual stream, where α is a learnable scalar *initialised to zero*:

$$y \leftarrow y + \tanh(\alpha_{\text{xattn}}) \cdot \text{attn}(q = y,\ kv = x), \qquad y \leftarrow y + \tanh(\alpha_{\text{dense}}) \cdot \text{ffw}(y). \tag{17.5}$$

Since $\tanh(0) = 0$, the new block contributes nothing at step zero and the conditioned model's output is *identical* to the frozen language model's. Vision does not intrude; it is admitted, gradually, as far as the gradient finds it useful, and each layer sets its own α independently. Removing the zero-initialised gate costs 4.2 points on the aggregate score and destabilises training — this is a load-bearing detail, not a flourish, and the same idea recurs wherever a pretrained network has to be extended without being disturbed.

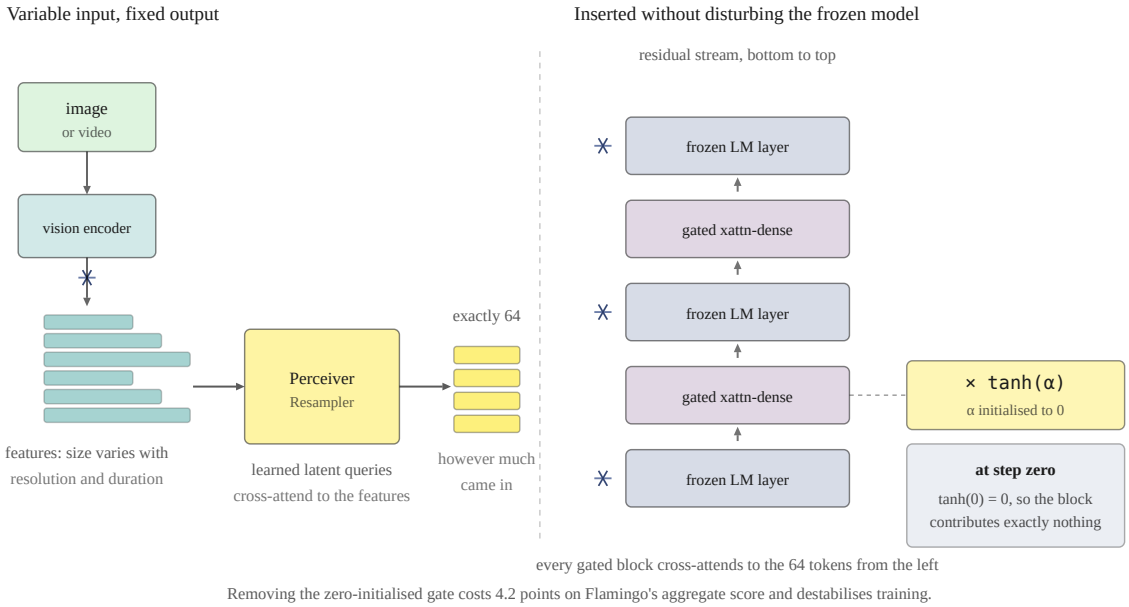


Figure 17.7.: Flamingo's architecture. The Perceiver Resampler turns a variable-size feature grid into exactly 64 visual tokens; gated cross-attention blocks are interleaved between the frozen language model's layers, each scaling its contribution by $\tanh(\alpha)$ with α initialised to zero. At initialisation the composite model computes exactly what the frozen language model computes.

The third piece is the training data, and it is what produces the capability the model is remembered for. Flamingo trains on web pages with images and text *interleaved* in their original order, rather than on isolated image–caption pairs. A sequence therefore looks like several images, each with surrounding prose — which is structurally identical to a few-shot prompt. The attention mask then does something deliberately severe: each text token may attend to the visual tokens of only the single most recent preceding image, not to all images in the sequence. Cross-attention is strictly local; the frozen language model's own self-attention, which sees the whole sequence, is what carries information across examples.

The constraint is what makes the arrangement work. Letting every token attend to every image would blur the image–text correspondence the model needs; forbidding it keeps each pairing crisp while the unmasked text pathway still lets the model recognise that the earlier pairs demonstrate a pattern. Trained this way, Flamingo does in-context learning across modalities: shown three image–answer pairs in a novel format and then a fourth image, it continues the pattern, with no gradient step and no task-specific head. It sets a new few-shot state of the art on 16 benchmarks,

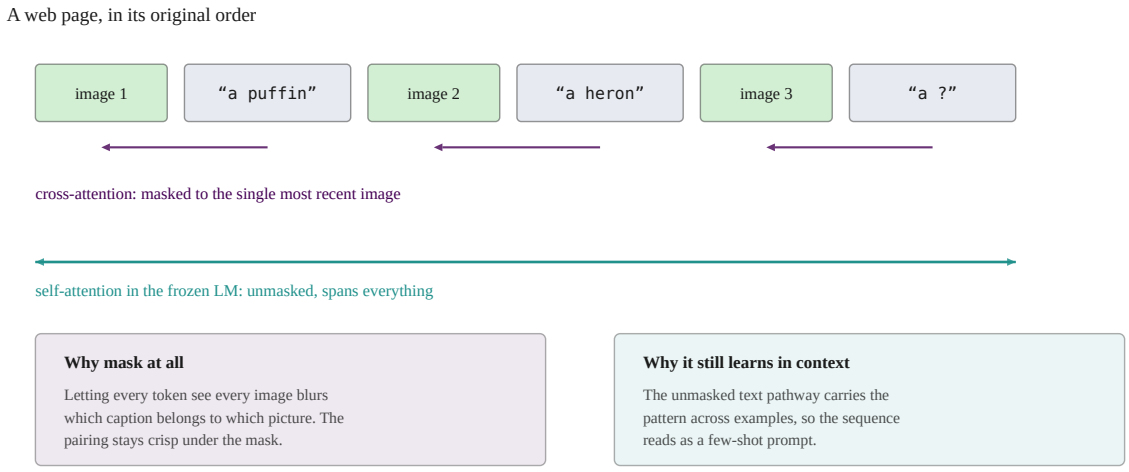


Figure 17.8.: Flamingo’s per-image masking on an interleaved sequence. Each text token cross-attends only to the visual tokens of the image immediately preceding it, so the association between an image and its own text is never diluted. Longer-range structure — the pattern that makes a sequence of examples a few-shot prompt — is carried by the language model’s self-attention, which is unmasked.

and on six of them a few-shot Flamingo surpasses models that were fine-tuned on the full training set — orders of magnitude more task-specific data.

17.8. Open weights are not open data

By 2024 the frontier multimodal models were proprietary, and the open replications shared a dependency that limited how good they could get. Their instruction data came from GPT-4V — the pipeline of Section 17.6, run against a stronger teacher — which is distillation, and a distilled student is bounded by its teacher. It also means the open community’s models improve only when a closed model does, which is a strange position for open research to occupy.

Molmo argues that the gap was never architectural. Its model is deliberately unremarkable: a vision encoder, a connector, a language model, assembled the way Section 17.6 describes. The claim is that the difference between the open models and the closed ones is *data*, and that the right response is to collect it rather than to distill it.

The difficulty is that the data you want is dense description — long, detailed, exhaustive text about what is in an image — and internet alt-text is the opposite. Alt-text is incidental: it exists to serve some other purpose, it is short, it is often a filename or an SEO fragment, and it describes almost nothing about the image’s actual contents. Asking crowdworkers to type dense captions instead runs into a plain human fact, which is that people will not type for very long, and the captions come back short no matter how the task is worded.

The workaround is the part of Molmo most worth remembering, because it solves an annotation problem by changing the *modality of annotation* rather than the incentive. Annotators are asked to **describe the image out loud for 60 to 90 seconds**, and the speech is transcribed. Speaking is fast and low-effort where typing is slow and effortful, and a person who will type twelve words will happily produce two hundred. Prompts direct attention to what a caption normally omits — spatial relationships, counts, text appearing in the image, background detail

— and the resulting PixMo-Cap dataset holds 712,000 images with captions far longer than anything scraped. No VLM was involved at any point, so the ceiling is human rather than GPT-4V.

The second dataset is a genuine addition to what these models can do. Grounding — saying *where* something is — has historically meant bounding boxes or segmentation masks, both slow and awkward to annotate. Molmo asks annotators to **point**, emitting 2D coordinates, and collects 2.3 million pointing annotations. A point is much cheaper than a box and turns out to be the more useful primitive.

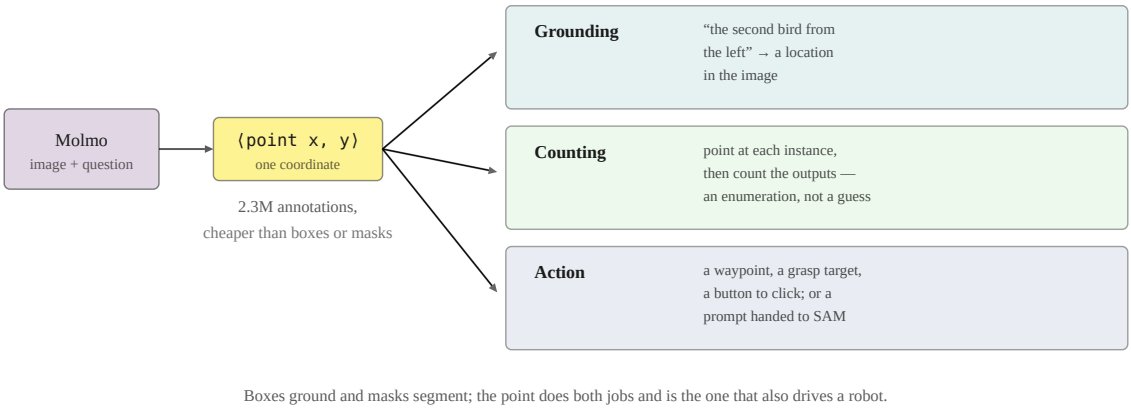


Figure 17.9.: Pointing as a single output primitive. The same coordinate output grounds a referring expression, answers a counting question by enumerating instances rather than estimating a number, and hands a target to a downstream system — a segmentation model that turns the point into a mask, or a robot that turns it into a waypoint. Boxes and masks each serve one of these; a point serves all three and is the cheapest of the three to annotate.

Pointing unifies three things that were separate. It grounds language, by attaching a phrase to a location. It fixes counting: a model asked *how many* now answers by pointing at each instance and counting its own outputs, which converts a perceptual estimate into an enumeration and is measurably more accurate. And it is an action interface — a point is a navigation waypoint, a grasp target, or a UI element to click, so the same output that answers a question can drive a robot or an agent. A point is also composable with a model that turns points into regions, which is exactly what Section 17.9 provides; chaining the two gives segmentation masks for anything Molmo can name, with neither model trained for the combination.

Molmo's 72B model ranks second on human-preference evaluation among all models tested, open and closed, and achieves the highest academic benchmark score in its comparison — from a conventional architecture and better data. The lecture's summary is the correct one: quality over quantity, and intentional collection over incidental scraping, even for pretraining.

17.9. Promptability as the pretraining task

Every model so far outputs text. Segmentation asks for something else — a mask, per pixel — and the question of what a *foundation model* for segmentation would even be is not obvious, because segmentation has no equivalent of a caption to supervise it at scale.

SAM resolves this by choosing the task first. Rather than segmenting a fixed set of categories, it defines **promptable segmentation**: given an image and a prompt indicating what to segment — a point, a box, a rough mask, or text — return a valid mask. The prompt vocabulary is open, and the task is general enough to serve both as a pretraining objective and as the interface downstream users get.

Making it work requires confronting a problem that supervised segmentation datasets define away by fiat. A prompt is frequently ambiguous, and irreducibly so. A single point on a person's shirt might mean the shirt, the person, or the group the person is standing in. All three are correct segmentations of that point, and no amount of additional training data settles which one was intended, because the intent is not in the image. A model forced to emit one mask learns to average the three, which is the worst answer available — a mask that is nobody's intended object.

SAM's response is to predict **three masks per prompt** with a confidence score each, corresponding roughly to the part, the whole, and the containing group. During training the loss is computed against the best-matching of the three only, so the three heads specialise rather than converging; at inference the caller picks. Ambiguity is represented rather than resolved.

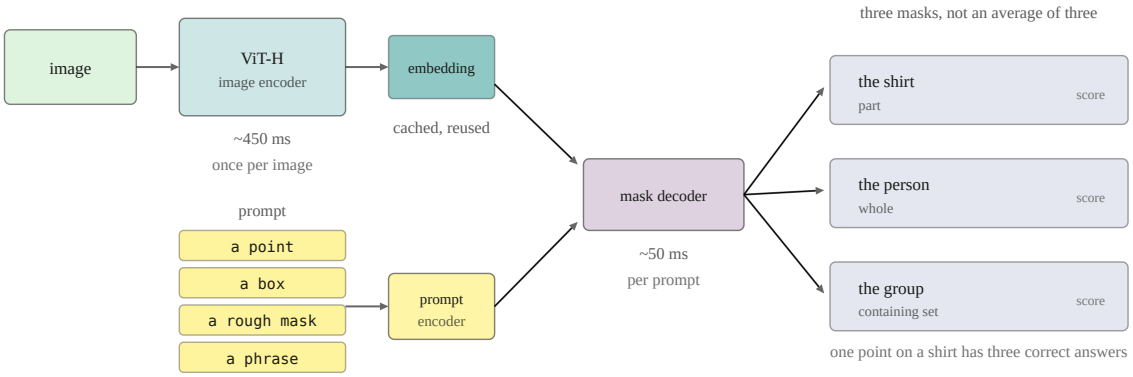


Figure 17.10.: SAM's architecture, drawn along the axis that its runtime behaviour depends on. The heavy ViT encoder runs once per image and its embedding is reused; each prompt then costs only the lightweight decoder, which is why interactive use with many prompts stays real-time. The decoder emits three masks with confidence scores, so an ambiguous point returns the part, the whole and the group rather than an average of them.

The cost structure in Figure 17.10 is a design decision, not a consequence. Interactive segmentation means many prompts against one image — a user clicking repeatedly, or a downstream model probing a grid of points — and the asymmetry between a heavy encoder run once and a tiny decoder run per prompt is what makes that affordable. Encode once, decode cheaply, many times.

That leaves the data, and here the promptable task pays for itself in a way worth following, because the model and the dataset are built by the same process. There is no web-scale source of segmentation masks the way there is for image–text pairs, so SAM's authors ran a **data engine** in three stages. First, annotators segment interactively with model assistance, and the model

retrains on what they produce. Then the model proposes confident masks itself and annotators only fill in what it missed, which shifts human effort towards the model's blind spots. Finally, the model annotates fully automatically: prompted with a dense grid of points across each image, filtered for confidence and stability, it generates masks with no human in the loop. The result, SA-1B, holds over 1 billion masks across 11 million images — roughly 400 times more masks than any prior segmentation dataset — and the great majority were produced by the model the dataset was collected to train. Each stage's model makes the next stage's annotation cheaper, and the loop is only possible because the task was defined to be promptable in the first place.

What SAM does not do is name anything. It segments what you point at and has no notion of what the region is, which is precisely why it composes so well with the models of the previous sections: CLIP or Molmo supplies the semantics and the location, SAM turns the location into a boundary. Neither was trained for the pairing.

17.10. Chaining, and the case for not training

Everything above answers a new demand by training something. There is a fourth move, and by the end of the lecture it is the one that looks most like where the field went: leave the models frozen and *compose* them.

The small version of the idea is CuPL, which revisits the prompts of Section 17.3. Those templates are hand-written, generic, and identical for every class — `a photo of a {label}` says nothing about what distinguishes the label from anything else. But a language model has read a great deal about these categories and knows what they look like. So CuPL asks one: *what does a platypus look like?* The answers come back as descriptive sentences naming the discriminating features — the bill, the flat tail, the fur — and those sentences become the prompts. The classifier's weights are now written by a model that knows something about the class, rather than by a template that knows only its name. It gains over a percentage point on ImageNet, requires no training and no hand-written templates, and its ingredients are two frozen models and a prompt.

VisProg generalises the move as far as it goes. Given a natural-language instruction — something compositional, of the kind a single model handles poorly, like *tag the seven dwarfs in this image* or *hide the face of the person on the left* — it uses a language model's in-context learning to emit a short Python-like **program**. Each line calls a module: an object detector, CLIP, a segmentation model, an inpainting model, an image-processing routine, or plain arithmetic. An interpreter then executes the program line by line, feeding each module's output to the next.

Nothing is trained. The language model has never been fine-tuned for this; it is shown a handful of instruction-and-program pairs in context and generalises to new instructions from there. Adding a capability means registering a module and adding one example to the prompt, which is a considerably shorter path than collecting a dataset. And the interpretability is genuine rather than post-hoc: the program *is* the reasoning, the intermediate outputs can be inspected at every line, and when the answer is wrong you can see which step produced the wrong thing.

The costs are real and the lecture states them. Every module invocation is a full forward pass, so latency compounds down the program; if any module is a hosted API, each line is a paid call and a network round trip. Errors compound too, since a mistake early in the program is faithfully propagated by every step after it, and the system has no way to notice. Composition buys flexibility and interpretability at the price of speed and reliability.

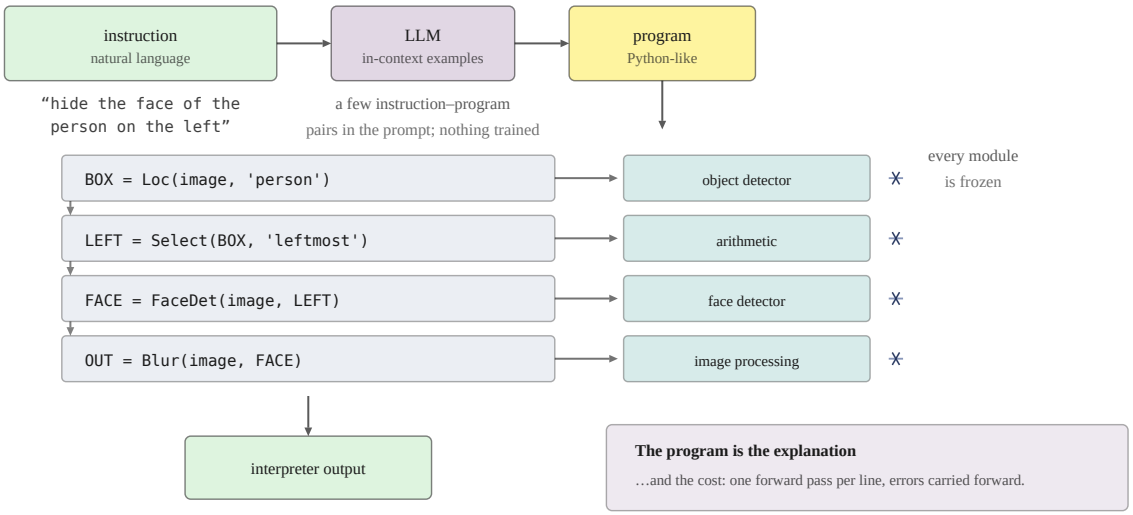


Figure 17.11.: VisProg's execution. A language model, given a few examples of instruction-and-program pairs, compiles a new instruction into a short program over frozen modules; an interpreter runs it, threading intermediate results through. Nothing is trained — extending the system to a new capability means registering a module and adding an example to the prompt — and the program itself is the explanation of how the answer was produced.

17.11. What the chapter adds up to

The trajectory here has a shape, and it is worth naming because it is not the shape one would guess from the architectures alone. Nothing in this chapter turned on a new layer type. CLIP is a contrastive loss the field already had, pointed at a supervision signal that was already lying on the internet. LLaVA is a matrix. Flamingo's one real invention is a scalar initialised to zero. Molmo's architecture is conventional by design, and its contribution is a microphone. SAM's is a task definition that makes its own dataset collectable. VisProg trains nothing at all.

What each of them changed instead was the *interface*. CLIP replaced a fixed label set with an open vocabulary, at the cost of a representation too coarse to say where anything is or how many there are. LLaVA and Flamingo replaced a similarity score with free-form text, by two different routes — concatenate into the prompt, or cross-attend through a gate — and the choice between them comes down to whether the language model may be modified. Molmo replaced the caption with the point, which grounds, counts, and acts through one output. SAM replaced the category with the prompt. And chaining replaced the model with a program over models.

The limitation from Section 17.5 survives all of it, which is the note to end on rather than a summary. A model trained to rank captions against random negatives learns which words are present and not how they compose, and every downstream system in this chapter inherits some of that. Better data narrows it, generative objectives narrow it, explicit programs route around it — but the compositional structure of a visual scene is still the thing these models represent least well, and it is the thing the next chapter's robots cannot do without.

Part VII.

Embodied AI

18. Robot Learning

Lecture 17

Based on Lecture 17 of CS231n, Stanford University, Spring 2025, given by Yunzhu Li.

18.1. Closing the loop

Every model in this book so far has been a function. An image goes in, a label or a box or a mask or a sentence comes out, and the transaction ends there: nothing the model produces affects what it will be shown next. That property is so deeply assumed that it is rarely stated. It is what licenses splitting a dataset into train and test, what makes the samples independent enough for the loss to be a sum over them, and what lets a benchmark number mean anything at all.

A robot breaks it. The action a robot takes changes the state of the world, and the changed world is the next observation. The data the robot will be trained on tomorrow is a consequence of the policy it is running today, which means the training distribution is not a fixed thing to be sampled but a moving object the learner itself is dragging around. Almost everything difficult in this chapter follows from that one fact, and it is worth holding on to as the organising idea rather than as a caveat.

The framework that replaces the function has four parts, and the whole field is phrased in them. At each time step the agent observes a **state** s_t, chooses an **action** a_t, and the environment responds with a new state s_{t+1} and a scalar **reward** r_t measuring how well things are going. A **policy** π is what maps states to actions, and the objective is not to be right about any single step but to maximise the reward accumulated over a whole episode:

$$\pi^\star = \arg\max_{\pi} \; \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t \;\middle|\; \pi\right] \tag{18.1}$$

where $\gamma \in (0, 1]$ discounts rewards that arrive later, and the expectation is over whatever randomness the environment and the policy contain. The discount is not only a modelling convenience; it is what keeps the sum finite when the episode has no natural end.

The formulation is worth taking seriously partly because of how far it stretches. Balancing a pole on a cart makes the state the pole's angle and angular velocity together with the cart's position and speed, the action a horizontal force, and the reward one unit for every step the pole stays upright. Legged locomotion makes the state the angles and velocities of every joint, the action a torque per joint, and the reward one unit per step taken while still standing. Playing Atari makes the state the raw screen and the action a joystick direction, with the reward the change in score. Go makes the state the board, the action where to place the next stone, and the reward a single bit delivered at the end of the game. Even a language model fits: the state is

the text so far, the action is the next token, and a chatbot optimised against human preference is an agent whose reward is a person's satisfaction with the reply.

That range is the formulation's strength and also the source of a mistake worth heading off early. The fact that Go and cloth folding can be *written down* the same way does not make them the same problem, and the difference is not merely one of difficulty. In Go the state is exactly and completely observable, the transition function is known perfectly, and the reward is unambiguous. In cloth folding none of the three holds. The state is whatever a few cameras can see of a self-occluding deformable object; the transition function is the physics of fabric, which nobody can write down at the resolution needed; and the reward is a human judgement about whether the result looks folded. Three different people will fold a shirt three different ways, and each will be satisfied with the outcome. There is no unique goal state to reach and therefore no unique reward function to optimise, only a family of them corresponding to what different users happen to want — smallest folded area, fewest creases, fastest completion. The reward is a design decision, and pretending otherwise is how a well-optimised policy ends up doing something nobody asked for.

So the honest way to state the difference from the rest of the course is this. Computer vision, in the form the previous chapters treat it, is a representation-learning problem: given high-dimensional input, produce a useful description of it. Robot learning is a constrained optimisation problem in which the constraints are physics, the objective is defined over a goal, and the variable being solved for is a sequence of actions. The perception is still there, and it is still hard, but it has stopped being the answer and become a component inside a loop.

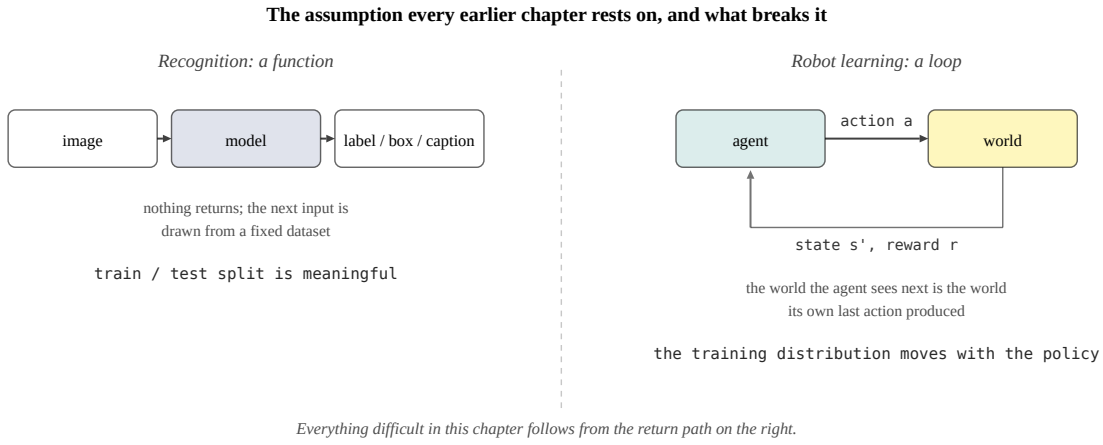


Figure 18.1.: The structural difference the chapter turns on. A recognition model is a function: input enters, output leaves, and nothing returns. A robot's output is an action executed in the world, so the world's response becomes the next input — the training distribution is generated by the policy being trained, not sampled from a fixed dataset.

18.2. What makes robot perception a different problem

The last chapter ended by observing that its models represent the compositional structure of a scene least well, and predicting that this is exactly what a robot cannot do without. This section is where that prediction starts to be paid, and the reason is that a robot needs a different *kind* of answer than a caption or a mask.

Consider what Section 17.8 established. Molmo's contribution was to make **pointing** a first-class output: the model emits 2D coordinates rather than text, and the same coordinate grounds a referring expression, enumerates instances for counting, and — the part that matters here — hands a target to something downstream. A point is a navigation waypoint or a grasp target. That is already an action interface, and it is the cleanest bridge between the two chapters: it is the moment a vision model stopped producing a description of the world and started producing something a controller can consume.

But a point is only the beginning of what a manipulation policy needs, and the gap between them is the substance of robot perception. A grasp needs the object's 3D pose, not its image coordinates. A push needs to know how the object will move when pushed, which depends on its mass distribution and its friction against the table. Folding a shirt needs the configuration of a surface that is mostly hidden behind itself. None of those are recoverable from a single labelled frame, and all of them are compositional in the strict sense: they are facts about *parts and their relations*, not about the identity of a whole.

Four differences separate this from computer vision as the course has treated it, and it is worth being precise about each because each one closes off a technique that worked earlier.

The first is that robot vision is **3D and metric**. A detector that reports a bounding box in pixels has said nothing a gripper can use; the robot needs a pose in metres, in its own coordinate frame, with an uncertainty attached. This is why depth is not an optional extra channel in robotics the way it often is in vision benchmarks, and it is why the geometry of Section 16.1 is closer to the working practice of robotics than object detection is.

The second is that the robot is an **active perceiver**. A vision dataset is a fixed collection of images somebody else chose; a robot chooses its own viewpoints and can act in order to see. If an object is occluded, the robot can move the camera, or move the occluder, or rotate the object in its hand — and these are the same kind of decision as the manipulation itself, made by the same policy against the same objective. Perception stops being a preprocessing stage and becomes one of the things the policy is for.

The third is that it is **real-time and closed-loop**. A model that takes two seconds per frame is unusable for a controller that must issue commands at tens of hertz, no matter how accurate it is. This constraint is unfamiliar from the rest of the course, where latency appears as an engineering footnote, and it is severe enough to be an architectural constraint: it is why the models of Section 18.7 are measured in control frequency as much as in success rate, and why a 35-million-parameter policy running at 3 Hz was a headline result rather than an embarrassment.

The fourth is that it is **task-driven**, and this is the one that most changes what "good perception" means. There is no general-purpose representation that is correct independently of what the robot is about to do. Segmenting every object in the scene to pixel accuracy is wasted computation if the task is to push one pile of coffee beans into a square, and a representation that captures the beans as a coarse distribution is better in the strict sense of leading to better actions. Perception and control are co-designed, and the criterion for a representation is whether a policy or a planner built on it works — not whether it scores well on a perception benchmark.

The sensor suite reflects all four. Cameras and depth sensors supply the scene, but proprioception — the joint encoders reporting the robot's own configuration — is the signal a controller most depends on, because it is fast, accurate, and never occluded. Force and torque sensing at the wrist reports contact, which vision cannot see at all: whether the gripper is actually holding something, and how hard. Tactile sensors in the fingers report whether a grasp is slipping. These are complementary in an important way, since the moments when vision is least informative —

when the hand is on the object, occluding it — are exactly the moments when contact sensing is most informative.

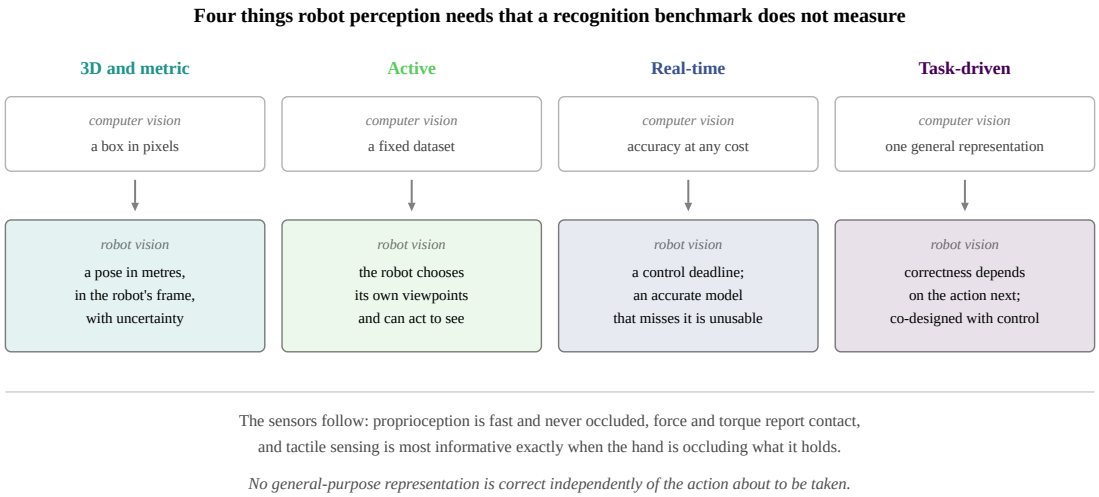


Figure 18.2.: The four axes on which robot perception departs from the recognition pipeline. Each closes off a technique that worked earlier: a pixel-space box is not a graspable pose, a fixed dataset cannot be re-photographed from a better angle, an accurate model that misses the control deadline is not usable, and there is no task-independent notion of a good enough representation.

18.3. Four things the reward has to pay for

Reinforcement learning is the most direct response to Equation 18.1: let the agent act, observe what reward comes back, and shift its behaviour towards whatever produced more of it. Nothing supervises the individual action. The only signal is a scalar arriving from the environment, and everything the agent knows about which of its thousands of decisions were good has to be extracted from that scalar. Four properties separate this from the supervised loop of the earlier chapters, and each one is a place where a technique that worked there stops working.

The first is that **the environment is stochastic**, so the same action twice does not produce the same outcome. Push a box across a table and the distribution of contact forces under it — which no sensor reports — decides whether it slides straight or rotates, and by how much. The consequence is that reward is a random variable rather than a measurement, and a single trajectory is a single sample from it. An action that was correct can be followed by a poor reward and an action that was wrong by a good one, so the learner cannot trust any individual outcome and has to average over many.

The second is **credit assignment**, and it is the one the field has spent the most effort on. In supervised learning the loss attaches to the prediction that caused it: the gradient of a cross-entropy against a label points at the specific output that was wrong. In sequential decision-making the reward is delayed, sometimes until the very end. A game of Go returns one bit after roughly two hundred and fifty moves, and the move that lost the game may have been the fortieth. The learner must distribute a single scalar over a long sequence of decisions with nothing telling it which of them deserved it.

The third is that **the environment is not differentiable**. The supervised chain from input through model to loss is differentiable end to end, which is what makes backpropagation applicable

at all. The chain from action to reward passes through physics, or through a game engine, or through a human's judgement, and none of those admits a derivative. So the gradient of reward with respect to action has to be estimated by sampling — by trying variations and inferring a direction from the differences in return. Zeroth-order estimates like this have variance that grows with the dimension of the action space, which is the technical root of why reinforcement learning needs so much data. It is not that the algorithms are wasteful; it is that a sampled gradient carries far less information per interaction than a computed one.

The fourth is **non-stationarity**, and it is the closing of the loop from Section 18.1 restated in the language of training. In supervised learning a prediction does not affect which example comes next, so the data distribution is fixed and the i.i.d. assumption holds. Here the states the agent visits are generated by the policy the agent is currently running, so improving the policy changes the distribution the policy is being evaluated on. The target moves because the learner moved it.

The classical answer to credit assignment is to learn a function that answers the counterfactual directly. The **action-value function** $Q^\pi(s, a)$ is the discounted return expected from taking action a in state s and following π afterwards:

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s,\ a_t = a,\ \pi\right] \tag{18.2}$$

If this function were known for the optimal policy, acting would be trivial: evaluate $Q^\star(s, a)$ for every available action and take the largest. That is the whole of the decision rule, and it is why Q is worth the trouble. What makes it learnable without waiting for episodes to finish is that it satisfies a consistency condition relating each state to its successor,

$$Q^\star(s,a) = \mathbb{E}_{s'}\left[\, r + \gamma \max_{a'} Q^\star(s', a') \,\right] \tag{18.3}$$

so a single observed transition (s, a, r, s') gives a target for $Q(s, a)$ built out of Q evaluated one step later. Learning becomes regression of the network against its own slightly-more-informed estimate, and the reward propagates backwards along the trajectory one transition at a time. Credit assignment is solved not by tracing responsibility through the sequence but by defining a quantity whose value at each step already contains the future.

Deep Q-learning made this work on raw pixels, and the architectural detail that matters is small and easily missed. Rather than taking a state and an action and returning one number — which would need a separate forward pass per action — the network takes only the state, four stacked frames of the Atari screen so that velocity is recoverable from a static input, passes them through convolutional layers and a fully connected head, and emits one Q value per action in a single pass. Choosing an action is then an $\arg\max$ over the output vector. The same network, the same hyperparameters and no game-specific features were applied to seven Atari 2600 games, and the result outperformed all previous methods on six of them after ten million frames.

What that trained agent does is the best argument for the whole approach. On Breakout, ten minutes of training produces a paddle that occasionally touches the ball. Two hours produces one that reliably returns it. Several hours produce something nobody wrote down: the agent learns to dig a tunnel up the side of the brick wall and send the ball behind it, where it bounces along the top clearing rows without further intervention. That strategy was not demonstrated, not rewarded specially, and not anticipated. It is what exploration under a scalar objective can find when the objective is the only thing specified, and it is the reason the framework is worth its costs.

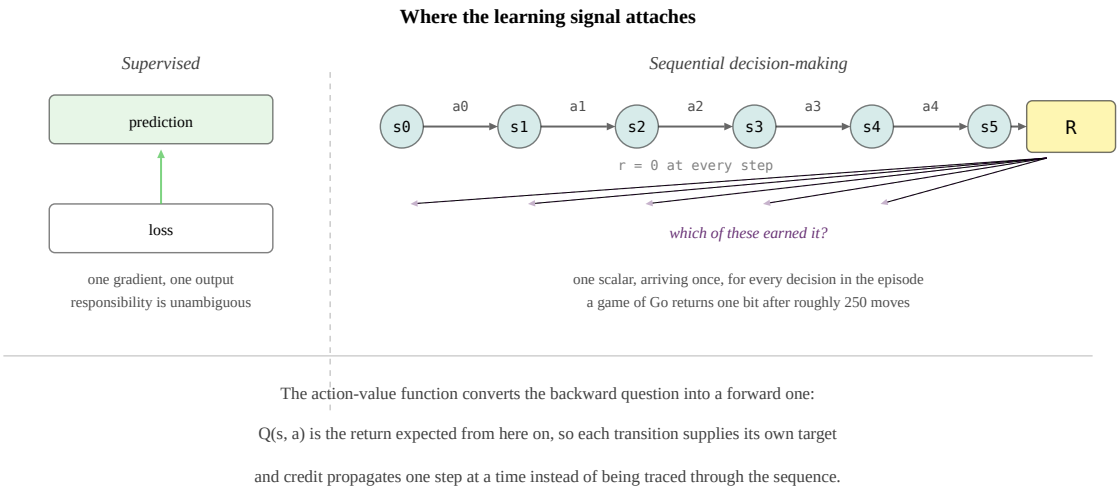


Figure 18.3.: Credit assignment. A supervised loss attaches to the prediction that caused it, so the gradient points at a specific output. A reward arriving at the end of an episode must be distributed over every decision that preceded it, with nothing indicating which of them earned it — the action-value function of Equation 18.2 is the device that converts the backward-looking question into a forward-looking one each step can answer locally.

18.4. Where scale worked, and what it cost

Go is the canonical demonstration, and the sequence after it is instructive because each step removed something rather than adding it. AlphaGo beat a professional player in January 2016 using a policy network bootstrapped from human games, a value network, and Monte Carlo tree search. AlphaGo Zero removed the human games — it learned from self-play alone — and was stronger. AlphaZero removed the Go-specific parts and applied the same algorithm to chess and shogi. MuZero removed the rules of the game, learning a latent dynamics model good enough to plan in without ever being told how the environment transitions — which is the first appearance in this chapter of the idea Section 18.5 is built on.

The lesson usually drawn from that sequence is Sutton's *bitter lesson*: the methods that win in the long run are the ones simple enough to absorb more computation, and hand-engineered structure tends to be a ceiling rather than a floor. It is a fair reading of the games results, and it is worth stating precisely because the rest of this chapter is largely about the conditions under which it does not transfer. In November 2019 Lee Sedol retired from professional Go, saying that an entity that cannot be defeated had appeared. Between then and now the same recipe, scaled further, produced strong agents for StarCraft II and Dota 2, games with vastly larger state spaces than Go. The empirical claim is close to unconditional: given a well-specified game and sufficient compute, reinforcement learning will produce a superhuman player.

The phrase carrying all the weight is *well-specified game*. A game has a perfect simulator — the rules — that runs faster than real time, resets instantly, never wears out, and is exactly the environment the agent will be evaluated in. Every one of those properties fails for a robot, and what happens when they fail is best seen in the two robotics results the lecture uses.

The first is legged locomotion. ANYmal trained by Lee and colleagues walks over terrain that defeated previous controllers — loose rubble, mud, snow, running water — using only proprioception, with no vision at all. The interesting part is the training structure, which exists

precisely because the simulator and reality diverge. Training a rough-terrain controller directly from the sensors the real robot has is too slow to converge, so the work splits it in two: a **teacher** policy is trained in simulation with access to privileged information the real robot could never have — the exact terrain profile, the ground-truth contact state of each foot — and then a **student** policy that sees only the proprioceptive history available on hardware is trained to imitate the teacher. The privileged information makes the problem easy enough to solve; the imitation step converts the solution into one that can actually be run. The student has to infer from the history of joint positions and torques what the teacher was told directly, and it does, recovering contact and slip events from proprioception alone.

The second is dexterous manipulation, and it is the honest one. OpenAI trained a five-fingered hand to manipulate a Rubik's cube, using automatic domain randomisation — a distribution over simulated physics whose difficulty is increased whenever the policy is doing well, so the policy is forced to become robust to a range of dynamics wide enough to contain reality. The result is genuinely impressive and the accounting is sobering. The policy consumed roughly **thirteen thousand years of simulated experience**, trained on 64 V100 GPUs alongside 29,440 CPU cores for rendering and simulation, continuously for several months. And it does not reliably solve the cube: on a fixed fair scramble repeated ten times, it completed a sequence requiring fifteen face rotations 60 % of the time, and the full twenty-six-rotation scramble 20 % of the time. A more recent controller for in-hand reorientation of novel shapes reorients objects it never trained on with a median time near seven seconds, and drops a duck-shaped test object in 56 % of trials.

Three bottlenecks explain the gap between the games results and these, and they are the reasons the rest of the chapter exists.

Sample cost. The number of interactions model-free reinforcement learning needs is enormous — Figure 18.4 puts it against the alternatives — and it is not an implementation defect but a consequence of the zeroth-order gradient described above. In a simulator that cost is a compute bill. On hardware it is impossible: thirteen thousand years does not fit in a laboratory, and a real robot executing millions of trials wears out its own actuators long before convergence.

Reward design. A game hands you the reward. A manipulation task does not, and writing one is a design problem with no correct answer, as Section 18.1's folded shirt already showed. Worse, the optimiser is adversarial towards any specification that is even slightly wrong: reward the robot for lifting the object and it will learn to nudge the object up and drop it repeatedly, because that is what the reward literally asked for. Every practical result above rests on a reward function that took considerable human effort to shape, which means the human labour was not eliminated but moved.

The simulation-to-reality gap. Training in simulation is what makes the sample cost survivable, and it introduces the problem that the policy is optimal for a physics that is not the one it will be deployed in. Friction, compliance, actuator delay, sensor noise and contact dynamics are all approximations, and contact is the worst of them — precisely the regime manipulation lives in. Domain randomisation is the standard answer, and it works by asking for a policy robust across a family of dynamics wide enough to contain reality. The cost is that robustness is bought with conservatism and with more samples, since the policy must solve many environments rather than one.

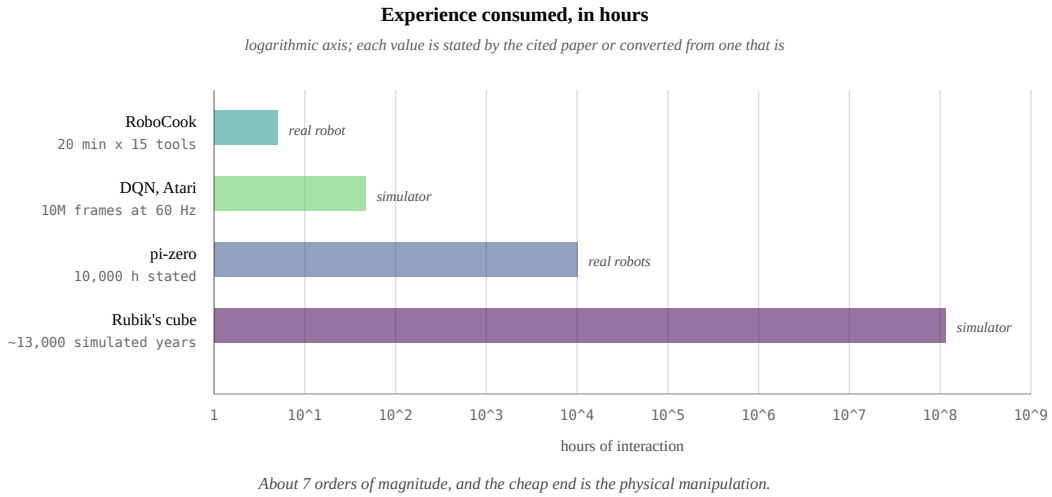


Figure 18.4.: Experience consumed by four results this chapter cites, converted to hours on a logarithmic axis. The spread is about seven and a half orders of magnitude, and it does not track difficulty — the five-hour result is real dough manipulated by a real robot, and the hundred-million-hour one is a simulated hand. What separates them is how much supervision each interaction carried. Every value is stated by the corresponding paper, or converted from a stated one by the arithmetic printed beside the bar.

18.5. Learning the model instead of the policy

Reinforcement learning discards something on every trial. The agent acts, receives a scalar, and updates a policy — but the transition it just observed, the fact that *this* push moved *that* object *this* far, is thrown away once its contribution to the return has been absorbed. That transition is a far richer piece of information than the reward attached to it, and it is available for free on every interaction.

This is also not how people work. Asked to push a mug towards the edge of a table, nobody tries it a thousand times to find out what happens. They know roughly what will happen, because they carry a predictive model of everyday physics accurate enough to plan against, learned from a lifetime of ordinary interaction rather than from a reward signal. The proposal of this section is to give a robot the same thing: learn a **forward model**

$$\hat{s}_{t+1} = \hat{f}_\theta(s_t, a_t) \tag{18.4}$$

by ordinary supervised regression on observed transitions, and then get the actions by inverting it. Note what has happened to the learning problem. Fitting $\hat{f}_\theta$ is supervised learning with dense targets: every step of every trajectory supplies an input–output pair, whether or not the trajectory succeeded, and there is no credit-assignment problem because the label for each transition is the very next observation. Failure data is as informative as success data. The delayed-scalar difficulty of Section 18.3 has not been solved so much as routed around.

Planning is then the inverse problem. Given the current state and a goal, search for the action sequence whose predicted outcome is closest to the goal:

$$a^\star_{0:H-1} = \arg\min_{a_{0:H-1}} \; d\big(\hat{s}_H,\, s^{\text{goal}}\big) \quad \text{subject to} \quad \hat{s}_{t+1} = \hat{f}_\theta(\hat{s}_t, a_t) \tag{18.5}$$

where d is a distance in whatever space the state lives in and H is the planning horizon. Because $\hat{f}_\theta$ is a neural network, this objective is differentiable in the actions and can be optimised by gradient descent through the unrolled model — or, in practice more often, by sampling thousands of candidate action sequences in parallel on a GPU and keeping the best, which costs no gradients and copes with objectives that are not smooth.

The model will be wrong, and the standard defence is to not trust it very far. Only the first action of the optimised sequence is executed; the true next state is then measured, and the whole optimisation is redone from there. This is **model-predictive control**, and it converts an open-loop plan into a closed-loop policy: prediction error does not accumulate over the horizon because the horizon is restarted at every step. It is the same receding-horizon idea that reappears, for a quite different reason, in Section 18.6.

None of that is the hard part. The hard part, and the actual research question, is the one Equation 18.4 quietly assumes an answer to: **what is s?** A dynamics model is only as good as the state it is a function of, and the history of this area is best read as a sequence of answers to that question.

The first answer was **pixels**. Deep visual foresight learns an action-conditioned video prediction model — given the current frame and a candidate arm motion, predict the frame that results — and plans by choosing the motion whose predicted frame moves a designated pixel closest to its target. The appeal is that nothing needs annotating. The robot collects its own pushing data autonomously, the supervision is the next frame, and no notion of an object appears anywhere in the system. The cost is that the model has to spend its capacity predicting the appearance of everything in view, including the tabletop and the shadows, when what the plan depends on is the motion of one object. Prediction in pixel space is also blurry over any useful horizon, and blur is precisely the failure mode that destroys a distance metric.

The second answer was **keypoints**. Keypoints into the future replaces the image with a small set of 3D points, obtained from self-supervised dense correspondence, that track the same physical locations on an object across time. The latent state becomes a few dozen numbers instead of a few hundred thousand, the dynamics model becomes correspondingly easier to fit, and the goal is specified directly as a target configuration of the same keypoints — which makes the distance in Equation 18.5 a distance in metres rather than in pixel intensities. The limitation is in the assumption: a fixed set of keypoints presumes an object with a fixed and roughly rigid structure. A pile of rice has no keypoints.

The third answer is **particles**, and this is where the chapter's connection to the last one gets paid. Represent the scene as a set of points with no fixed identity, build a graph over them by connecting points that are near enough to interact, and learn the dynamics as message passing on that graph. Each edge computes an interaction from the pair of nodes it joins, and each node updates itself from its own state and the sum of the interactions arriving at it:

$$e_{ij} = f_{\text{rel}}(v_i, v_j), \qquad v_i' = f_{\text{node}}\Big(v_i, \textstyle\sum_{j \in \mathcal{N}(i)} e_{ij}\Big) \tag{18.6}$$

Two properties follow from the fact that f_{rel} and f_{node} are *shared across every particle and every pair*, and both are the reason this representation works where the previous two do not. The model is indifferent to how many particles there are, so a model trained on one pile of coffee beans transfers to a larger pile without retraining. And it is indifferent to which particle is which, so there is no correspondence to maintain when a deformable object changes shape or a pile splits in two.

This is compositional structure in the strict sense that Section 17.8's models lacked it. The last chapter closed by observing that a model trained to match captions against random negatives

learns which entities are present without learning how they compose, and that the omission would matter here. A particle graph is the opposite arrangement: it represents *only* entities and their relations, and the physics it predicts is assembled from a local rule applied everywhere rather than recognised as a whole. That is why it generalises across object count, object size and object shape — the composition is in the architecture rather than in the training distribution.

The results follow the representation. Dynamic-resolution particle models manipulate piles of granular material — coffee beans, almonds, candy, granola, rice, corn — gathering them, sorting two mixed kinds, and redistributing a pile into complicated target shapes including letters of the alphabet. The refinement worth naming is that the resolution is not fixed: a regressor predicts how coarse the particle representation should be at each control step, because a task's early stages need only a rough sense of where the mass is while its final stages need fine detail to match a shape's boundary. Choosing granularity per step outperforms any fixed choice.

RoboCook is the most complete version of the argument and it makes a dumpling. The robot has four RGB-D cameras reconstructing the workspace into a point cloud, fifteen 3D-printed tools it can pick up and exchange, and a graph dynamics model of the dough learned from real interaction. Producing a dumpling from a lump of dough takes nine stages, and control operates at two levels: a classifier chooses which of the fifteen tools suits the current state and the goal, and a low-level module chooses the motion to execute with it. That low-level module is where the model earns its place — candidate actions and tools are sampled, the graph model predicts the resulting dough shape for each, and the one whose prediction lands closest to the target is selected. Because sampling at test time is slow, the sampling is done offline to generate a dataset and a policy is distilled from it, so the deployed system runs fast while still being derived entirely from the learned model's predictions.

Two things about that system are worth more than the demonstration. The first is what happens when a person interferes: a human repeatedly deforms the dough while the robot works, and at one point flattens a shape the robot had just cut. The robot does not fail and does not blindly continue. Because the tool classifier is a function of the current observation rather than of a step counter, an observation matching an earlier stage is simply routed back to that stage, and the robot redoes the work. Recovery is a consequence of having framed control as a function of state rather than as a script, which is worth noticing because nothing in the system was designed for disturbance rejection.

The second is a comparison the authors ran and the result is not the expected one. The obvious alternative to learning dough dynamics is to simulate them with a physics engine built for deformable material — the material point method — and identify its parameters from data. They did that, with extensive system identification, and the identified simulator's predictions were noticeably *less* accurate than the model learned directly from real interaction. This is worth sitting with, because it inverts the usual framing of the sim-to-real gap from Section 18.4. There, simulation was the cheap source of experience and reality the thing to transfer to. Here, twenty minutes of real interaction per tool produced a better model of this particular dough than a principled simulator fitted to the same data, because the learned model is free to absorb everything the physics engine's assumptions leave out. Sim-to-real is not a fixed toll to be paid; whether simulation helps depends on how well its assumptions match the material, and for contact-rich deformable manipulation they match poorly.

18.6. Imitation, and two ways it goes wrong

Reinforcement learning learns the policy and pays for it in interactions. Model learning learns the dynamics and pays for it in the difficulty of choosing a state representation. The third option

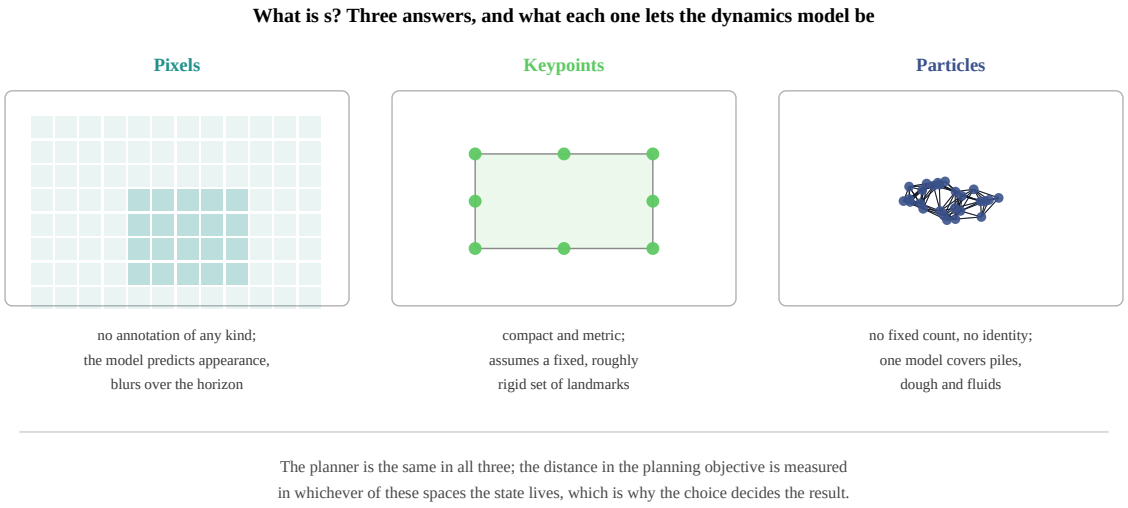


Figure 18.5.: The same scene under three state representations, and what each one lets the dynamics model be. Pixels need no annotation but spend capacity on appearance and blur over the horizon; keypoints are compact and metric but assume a fixed, roughly rigid structure; particles have no fixed identity and no fixed count, which is what lets one model cover piles, dough and fluids.

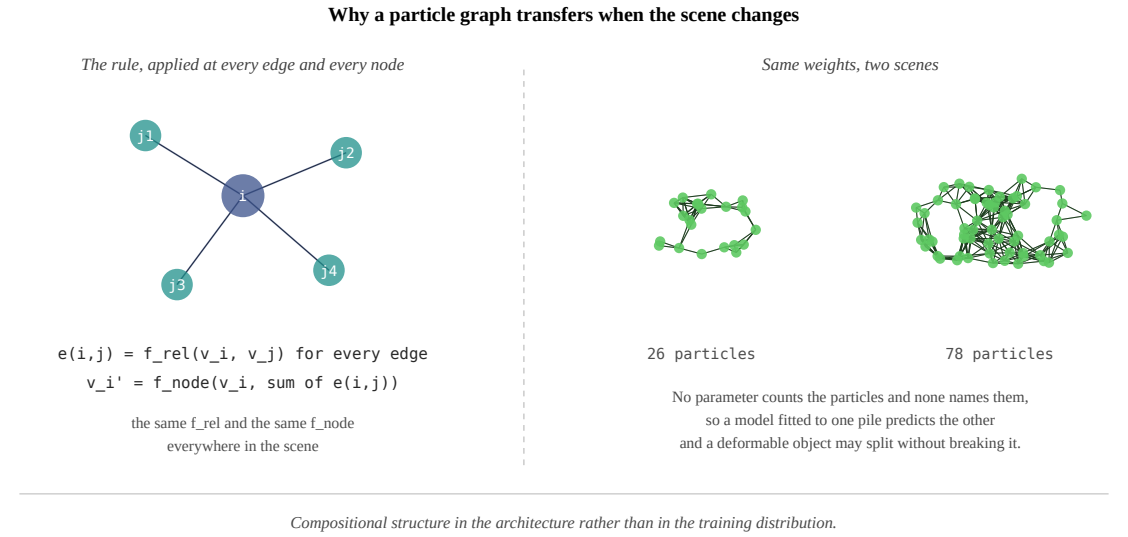


Figure 18.6.: Message passing on a particle graph, Equation 18.6. Edges are formed between particles close enough to interact; every edge applies the same relation function and every node the same update function, so the model is defined independently of how many particles the scene contains or which particle is which. That sharing is the compositional structure the previous chapter's models were missing.

is the one that sounds too simple to work: have a person demonstrate the task, record what they did, and fit a policy to the recording by supervised learning. This is **behaviour cloning**, and it is a straightforward regression of actions on observations,

$$\theta^\star = \arg\min_\theta \sum_{(o,a)\in\mathcal{D}} \big\|\pi_\theta(o) - a\big\|^2 \tag{18.7}$$

over a dataset $\mathcal{D}$ of demonstrated observation–action pairs. No reward function has to be designed, no exploration is needed, nothing has to be simulated, and the task specification is carried implicitly by the demonstrations. It is by a wide margin the fastest route to a robot that does something in the physical world: collect data in the morning, train overnight, and have a working policy by the following afternoon.

Two things go wrong with it, and they are worth separating because they have different causes and different fixes.

18.6.1. The distribution the policy makes for itself

The first failure is the closed loop of Section 18.1 arriving in its most concrete form. Equation 18.7 is fitted on the states the *expert* visited, but at deployment the states are the ones the *learner* visits, and those diverge. A small error moves the robot slightly off the demonstrated trajectory, into a state slightly less well covered by the training data, where the policy is slightly less reliable and makes a larger error, which moves it further off. The training loss can be small and the deployed behaviour still leave the data distribution entirely.

The size of that effect is worth computing rather than gesturing at, because the standard statement of it is imprecise in a way that matters. Take the worst case behind DAgger: a policy that errs with probability ε per step on the expert's distribution and, having erred once, stays off the distribution and pays unit cost for the rest of the episode. Its expected total cost over horizon T is the probability of having erred by each step, summed:

$$J_{\mathrm{BC}}(T,\varepsilon) = \sum_{t=1}^{T}\Big(1-(1-\varepsilon)^t\Big) = T - \frac{(1-\varepsilon)\big(1-(1-\varepsilon)^T\big)}{\varepsilon} \tag{18.8}$$

The generator behind Figure 18.7 evaluates both sides independently and checks that they agree, which is worth doing because the expansion everyone quotes is derived from the left-hand side and loses something on the way. For small ε the sum is $\varepsilon T(T+1)/2$, not εT^2: the growth is quadratic, but the leading constant is one half and the $+1$ is real. And the expansion is an upper bound that the exact cost outgrows its resemblance to — the exact cost can never exceed T, since an episode cannot cost more than its own length, so the quadratic regime holds only while εT is small and after that the curve bends into total failure rather than continuing to accelerate. At $\varepsilon = 0.01$ and $T = 50$ the exact cost is already only 85 % of its quadratic approximation.

The contrast with training on the learner's own distribution is stark. If the policy is corrected at the states it actually visits, the errors do not compound and the cost is $O(\varepsilon T)$ — linear. At $\varepsilon = 0.01$ and a horizon of 100 steps, that is a cost of 1 against 37, and by 400 steps the ratio is 76.

That is precisely what DAgger does. Train a policy on the demonstrations, run it, and record the states it visits; ask the expert what they would have done at each of those states; add those corrections to the dataset and retrain. The point is not that the expert demonstrates the task better the second time but that they demonstrate it *somewhere else* — on the states the learner's

own mistakes produce, which is exactly the region the original dataset omits. In practice this iterative cycle, collecting corrections where the deployed policy fails, is not an optional refinement but the standard working loop of every imitation-learning system that survives contact with hardware.

A different response to the same observation is to recover the *intention* rather than the actions. **Inverse reinforcement learning** takes the demonstrations and infers a reward function under which they would be optimal, then runs ordinary reinforcement learning against that reward. This makes the task explicit rather than leaving it implicit in the data, and it generalises differently: a reward function transfers to a new initial condition where a cloned trajectory would not. The classic demonstration is Abbeel and Ng's autonomous helicopter, which learned aerobatic manoeuvres from a human pilot's flights and then flew them, in some cases better than the demonstrations it learned from — which is possible precisely because it optimised the inferred objective rather than reproducing the inputs. The cost is that the inference is ill-posed: many reward functions explain the same behaviour, including trivial ones, and pinning down a useful one requires assumptions that are themselves a modelling problem.

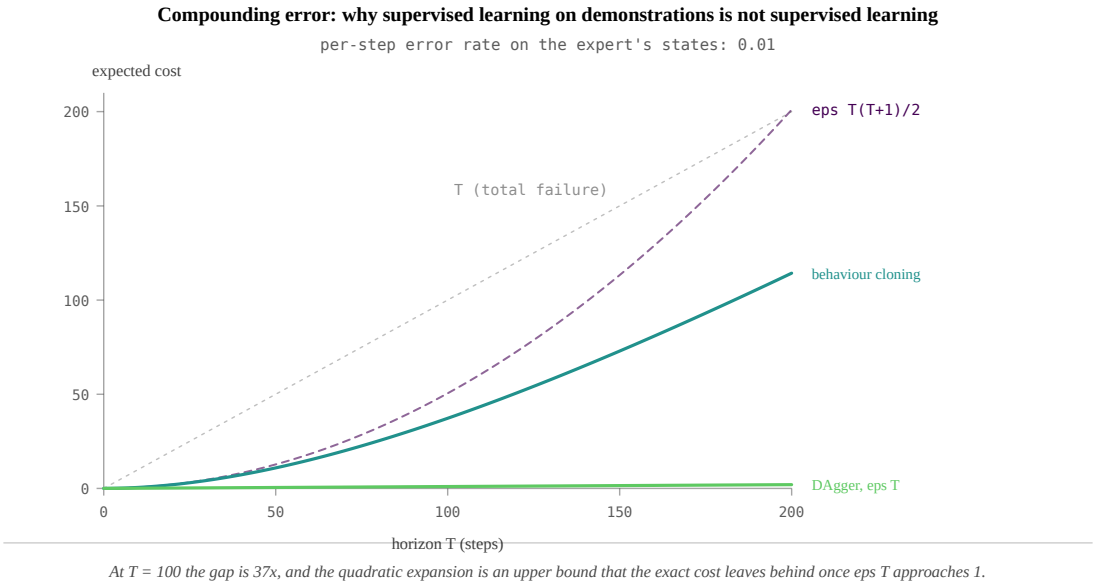


Figure 18.7.: Expected cost against horizon at a per-step error rate of 0.01, evaluated by the generator rather than quoted. Behaviour cloning compounds; correcting on the learner's own states does not. The dashed purple curve is the quadratic expansion the familiar bound comes from, and the dotted line is the ceiling that expansion ignores — an episode cannot cost more than its own length, so the quadratic regime ends once εT approaches 1.

18.6.2. Averaging two good answers

The second failure has nothing to do with distribution shift, and it is present even with unlimited demonstrations collected from the learner's own states. It is a defect in Equation 18.7 itself.

Squared error has a unique minimiser and it is the conditional mean. If the demonstrations at some state are unimodal, that is exactly what is wanted. But human demonstrations are routinely multimodal: shown an obstacle, one demonstrator goes left and another goes right, and both are correct. The conditional mean of *go left* and *go right* is *go straight*, into the obstacle. The regression has not made an error in any sense the loss can see — it has returned the value

that minimises expected squared error — and the action it returns is one no demonstrator ever took and no demonstrator would endorse.

It is tempting to leave the claim there, and the claim as stated is not quite true. The generator behind Figure 18.8 makes the toy case explicit: two demonstrated actions at $\pm d/2$, each Gaussian with standard deviation σ, mixed with equal weight. The L2-optimal prediction is the mixture mean, which the generator confirms two ways — analytically, and by minimising expected squared error numerically over a grid. But whether that mean is a *bad* action depends on d, and the dependence has a sharp threshold: an equal-weight mixture of two Gaussians is bimodal if and only if $d > 2\sigma$. Below that separation the two demonstrations are not distinguishable modes at all, the density has a single peak, and the mean sits on it. Regression is then the right answer.

The threshold is exact and the generator checks it by counting local maxima of the density on a fine grid while sweeping the separation, recovering $2.002\,\sigma$ against the closed-form 2σ. Above it, the situation inverts in a way stronger than "the mean is a compromise": the mean becomes a local *minimum* of the density. At $d = 4\sigma$ a demonstrated action is 3.7 times likelier than their average; at $d = 6\sigma$ it is 45 times likelier. The prediction that minimises squared error is, in its own neighbourhood, the least likely thing to do.

The fix is to stop asking the policy for a single number and ask it for a distribution. Implicit behaviour cloning does this with an energy function: rather than $\pi_\theta(o) \mapsto a$, learn a scalar $E_\theta(o, a)$ trained so that demonstrated actions have low energy, and select actions at inference by minimising it,

$$a^\star = \arg\min_a E_\theta(o, a) \tag{18.9}$$

An energy landscape can have two minima, and the arg min picks one of them rather than averaging them. It also copes with discontinuity: an explicit network is a continuous function of its input and therefore cannot represent a policy that must switch abruptly as the state crosses a boundary, whereas the location of an energy minimum can jump while the energy itself stays smooth.

Diffusion Policy applies the same insight through the machinery of Section 15.6. The policy is a conditional denoising diffusion process over actions: start from noise and iteratively refine it, conditioned on the observation, until it lands on an action. Because a diffusion model represents an arbitrary distribution rather than a point estimate, multimodality is handled by construction, and because Section 15.14's framing makes it a gradient field being followed rather than a function being evaluated, it scales to the high-dimensional output that the next idea needs. The policy does not predict one action but a short *sequence* of future actions, executing only the first few before replanning — the receding-horizon idea of Section 18.5 reappearing, but for a different reason: here it buys temporal consistency, since a policy that recommits to a whole plan at every step can dither between two modes, choosing left at one timestep and right at the next and going straight after all. Across 15 tasks from four manipulation benchmarks, this averages a 46.9 % improvement over the previous state of the art, and it works on tasks with genuinely fine-grained contact — spreading butter, peeling a potato, sliding a book onto a shelf.

18.7. Robotic foundation models

A robotic foundation model is, structurally, the least novel thing in this chapter. It has no explicit state representation and no transition function; it does not model the environment at all.

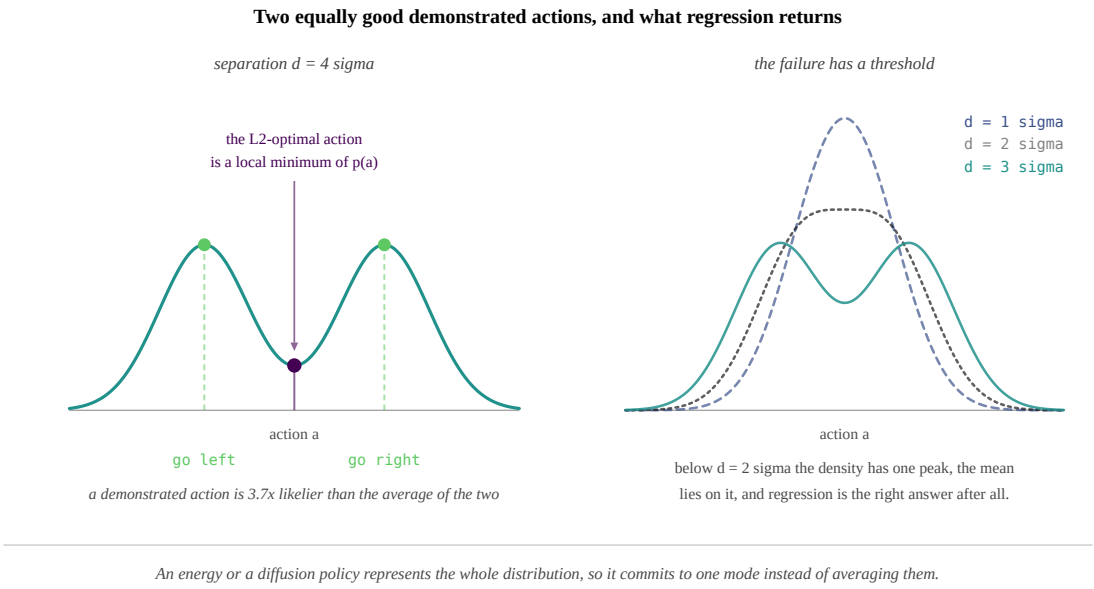


Figure 18.8.: Two equally good demonstrated actions and what squared error returns. Left: at a separation of four standard deviations the L2-optimal prediction sits at a local minimum of the action density, and a demonstrated action is 3.7 times likelier than their average. Right: the failure has a threshold. Below a separation of two standard deviations the mixture has a single peak, the mean lies on it, and regression is correct — which is why the objection to behaviour cloning is about *separated* modes rather than about averaging as such.

It is a policy mapping an observation and a goal to an action — exactly what Section 18.6 was fitting, trained by exactly the same supervised objective on exactly the same kind of teleoperated demonstrations. What is different is the demand placed on it, and the demand is Section 17.1's: the same weights should work across many tasks, many objects and many environments, including ones nobody anticipated.

The lecture's definition of the target is an analogy, and it is a good one because it names both the achievement and the failure mode in the same breath. A vision-language model's answer may not be correct, but it is always *reasonable* — fluent, on-topic, plausibly shaped. The corresponding claim for a robot policy is that the synthesised action may not be optimal, but the trajectory will always be smooth, continuous and responsive to the instruction. That is a real property and it is what makes these models feel different from a task-specific policy, which fails by jittering or freezing. It should also be read as a warning, for the same reason the language analogy is apt: a system that always produces something plausible has removed the most reliable signal that it is wrong. The names attached to this class — vision-language-action models, large behaviour models — describe the same object.

The line of work is short and fast. RT-1 in December 2022 established the recipe, and its numbers are the honest measure of what this costs: roughly 130,000 demonstration episodes covering over 700 tasks, collected with a fleet of 13 robots over 17 months. The model itself is small — 35 million parameters, running at 3 Hz — because the constraint is the control deadline of Section 18.2 rather than capacity. RT-2 made the change that matters conceptually: instead of training a policy that happens to consume vision and language, it takes a vision-language model already pretrained on the web and fine-tunes it to emit actions as if they were tokens, co-trained on the original vision-language data so the semantics are not overwritten. What that buys is generalisation that the robot data does not contain — asked to pick up an improvised

hammer, it reaches for a rock, because the web taught it what a hammer is for and the robot data only taught it how to pick things up. RT-X and OpenVLA followed, and 2025 brought a crowd of them.

0 is the one the lecture works through, and its architecture is the clearest statement of what the class has settled on. A pretrained vision-language backbone — PaliGemma, 3 billion parameters — supplies semantics from web-scale data. Bolted to it is a smaller **action expert** of about 300 million parameters that consumes the backbone's representation together with the robot's joint states and emits actions. The action expert does not emit discretised tokens; it produces continuous actions by **flow matching**, the straight-line variant of diffusion from Section 15.7, and it produces not one action but a *chunk* of consecutive future actions in a single forward pass, which is what allows control at up to 50 Hz from a 3.3-billion-parameter model. The relationship between the two halves is worth stating plainly, because it is the whole design: the backbone supplies knowledge that no volume of robot data could contain, and the expert supplies a control rate the backbone could never reach. Pretraining used about 10,000 hours of demonstrations across 7 robot configurations and 68 tasks, together with open cross-embodiment data.

The training is split the way language models are, and for the same reason. Pretraining produces a base model broad enough to handle simple tasks that resemble what it has seen. Anything harder — and anything genuinely new — requires **post-training** on task-specific data. This is not a shortcoming of the particular model; it is the shape of the field. The base model is a prior over reasonable behaviour rather than a solution to any specific task, and the honest reading of the demonstration videos is that the impressive long-horizon results are post-trained. 0.5 extends the claim furthest, cleaning kitchens and bedrooms in homes that were not in its training data, with multi-stage behaviours lasting ten to fifteen minutes.

Two caveats deserve as much attention as the results, and both come from the lecturer rather than from the papers.

The first explains something anyone watching these videos notices, which is that the robots are slow. The reason is not inference latency and not planning. The demonstrations were collected by a human **teleoperating the same robot**, and teleoperation is slower than using one's own hands even after hours of practice — the operator is driving an unfamiliar embodiment, at a distance, with occlusions that force them to shift viewpoint before they can tell whether a stage is complete. The policy is therefore imitating a human who has been artificially slowed down, and it inherits that speed exactly. The ceiling is in the data collection process, not in the model, which is why making demonstration collection faster is itself an active research direction rather than an engineering chore.

The second is a limit on the whole approach, and it returns to where this chapter started. Asked whether one large policy will eventually cover a household, the lecturer's answer is no. Folding a box is already a long-horizon task and it is genuinely impressive that a single policy handles it; but folding shirts *and* making beds *and* clearing floors, in an unfamiliar house, is a different kind of problem, and his expectation is that some higher-level abstraction — a scene graph, a symbolic decomposition — will have to sit above the vision-language-action model and steer it. That is the compositional structure of Section 18.5 reappearing at the level of tasks rather than of objects, and it is the same gap the last chapter identified: these models represent entities well and the relations between them poorly, and a policy that must plan across a house is a policy that must reason about relations.

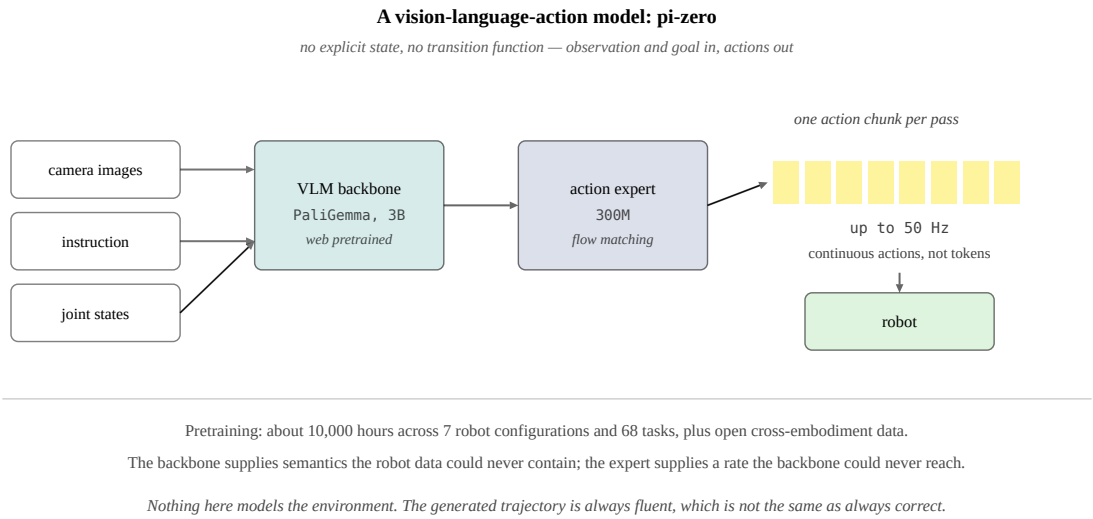


Figure 18.9.: The 0 stack. A web-pretrained vision-language backbone supplies semantics; a small action expert consumes its representation together with the robot's joint states and emits a chunk of continuous future actions by flow matching, which is what makes 50 Hz control possible from a 3.3-billion-parameter model. Nothing in the diagram models the environment — there is no state estimate and no transition function, only observation and goal in, actions out.

18.8. What is not solved

The most serious unsolved problem in robot learning is not a modelling problem. It is that **nobody can measure anything reliably**, and this follows from the same closed loop that started the chapter.

Evaluation is conducted in the real world, on real hardware, because nothing else predicts real performance. That makes it expensive and slow: a large laboratory runs a grid of teleoperation rigs for evaluation and waits two days for results, and the description offered of their situation — *we have a large enough budget that we can still make progress* — is a statement about how the bottleneck is being managed rather than solved. It also makes it noisy in a way that defeats comparison across groups. Two labs evaluating the same policy will get different numbers, because the initial configuration of the objects, the lighting, and the friction of the particular table are all part of the environment, and none of them is written down in a paper.

Underneath that is the deeper problem, and it is the cleanest structural break from every other chapter in this book. **The training loss does not predict task success.** In supervised learning the validation loss is the thing you care about, up to a generalisation gap; a lower loss is a better model, and the whole methodology of the field rests on that being true. A policy's training loss measures one-step action prediction accuracy on demonstration data. Task success measures whether a hundred-step rollout, from an initial state the policy chose to enter, accomplishes something. The compounding-error analysis of Section 18.6.1 is exactly why these come apart: a policy with a lower one-step error can have a higher deployed cost if its errors are correlated in the wrong way, and Equation 18.8 says the gap between per-step accuracy and episode cost grows with the horizon. There is no held-out set that fixes this, because the states the policy will be evaluated on do not exist until the policy generates them.

Simulation is the obvious escape and it has not yet worked as one. Large simulated benchmarks

exist — BEHAVIOR-1K at Stanford, Habitat 3.0 at Meta — and every one of them inherits the sim-to-real gap of Section 18.4, now in its most demanding form, because a benchmark has to simulate rigid objects, deformable objects and cloth all convincingly enough that the ranking of policies is preserved. Around that sit problems that are unglamorous and genuinely hard: generating enough assets, digitising real environments, procedurally generating scenes that are both realistic and diverse. The property that would make all of it worthwhile is the one thing not yet demonstrated, which is *correlation* — that a policy which scores better in simulation is actually better on hardware. The comparison the field reaches for is ImageNet, and it is the right one for a precise reason. ImageNet mattered not because the images were interesting but because, for several years, progress on that benchmark was progress in the field. Robot learning has no such object, and building one is arguably a larger contribution than any policy would be.

Three other threads are worth naming as where this goes next.

The first closes a loop within the chapter. The community is collecting **action-conditioned interaction data** at unprecedented scale in order to train policies — and using it only for policies discards most of what it contains. Every one of those transitions is a statement about how the world responds to being acted on, which is precisely the training signal Section 18.5 needed and never had at scale. A **foundation world model**, in the sense of action-conditioned future prediction, is the natural other thing to build from the same data, and the open questions about it are the ones this chapter has already met at small scale: should it predict in 3D, how much structural prior should it carry, and how should learning and physics be divided. The interesting prospect is not either artefact alone but the interplay — a policy proposes, a world model evaluates.

The second is that the foundation models being borrowed were not built for this. A language or vision-language model has read a great deal about the world and interacted with none of it, and its understanding of geometry, contact and physical consequence is correspondingly thin. The proposed correction rhymes with how language models were aligned: reinforcement learning from human feedback made models useful by optimising against judgements they could not derive from text, and reinforcement learning from *embodied* feedback would do the same against outcomes that only the physical world can supply.

The third is that none of this survives as a static artefact. A deployed robot meets new scenarios, users with preferences that differ from the ones in the training data, and its own accumulating experience — so adaptation and lifelong learning are requirements rather than extensions. And every result in this chapter is a systems result before it is a modelling result: delays, compute budgets, and modules that have to talk to each other on a deadline decide what works, and a policy that is correct but late is not a policy.

The chapter began by observing that a robot's output changes its next input, and every difficulty since has been that fact wearing a different costume. It is why the reward is delayed and the credit ambiguous; why a cloned policy drifts off the distribution it was fitted on; why the training loss stops predicting success; and why evaluation cannot be made cheap by holding data back. Reinforcement learning attacks it with interactions and pays in samples. Model learning attacks it with a predictive model and pays in the difficulty of choosing what to predict. Imitation attacks it with human demonstrations and pays in the cost of collecting them and in the horizon over which small errors grow. Foundation models attack it by borrowing everything the web already knows and paying only for the last mile in robot data. None of them removes the loop, because the loop is what the problem is.

Part VIII.

Impact

19. Human-Centered AI

Lecture 18

Based on Lecture 18 of CS231n, Stanford University, Spring 2025, given by Fei-Fei Li.

19.1. What the field was trying to match

The course ends where it began, with the human visual system, but asking a different question about it. Section 2.1 used human vision as evidence that the problem is hard: a person watching a rapid sequence of never-before-seen images can still spot a target in one of them, and can separate an animal from a non-animal within about 150 milliseconds of the image appearing (Thorpe, Fize and Marlot, 1996). That was offered as a benchmark to be reached. The closing lecture treats the same fact as a *specification* that has now largely been met, and asks what the specification was for.

Two things converged to meet it, and they are worth separating because only one of them is usually named. The first is the familiar triple of data, compute and learned representations, which arrived together around 2012 and which Section 2.9 covers. The second is the source of the problems themselves. Categorisation, attention, scene understanding, the distinction between recognising an object and understanding a situation — none of these were posed by engineers looking for tractable benchmarks. They came from psychology and neuroscience, which had spent a century establishing that these are separable capacities in the first place. The traffic still runs both ways: vision models are now used as instruments for studying biological vision, and the questions that biological vision raises continue to supply the field with problems it would not otherwise have thought to pose.

That framing sets up the rest of the lecture, which is organised as three questions rather than three techniques. Can we build machines that see what people see? Can we build machines that see what people *cannot*? And — the question that turns out to carry the weight — can we build machines that see what people *want* seen? The first is a matter of capability, and it is substantially answered. The second and third are not questions about capability at all, and most of this chapter is about why.

19.2. Seeing what no person could

Human visual competence is uneven in a specific way: it is excellent at the level of categories that mattered for survival and social life, and poor below that level. Almost anyone can identify a bird; almost nobody can name the species. There are on the order of ten thousand bird species and, since the 1970s, thousands of distinguishable car models once make, model and year are all counted. This is the regime of fine-grained categorisation, and it is one of the few places where

a vision model can be straightforwardly superhuman, because the human baseline is close to untrained.

The lecture is candid that the problem is not solved. Error rates on fine-grained bird classification climb steadily as the label becomes more specific — coarse taxonomic levels are easy and the species level is where the errors concentrate — and the problem has drifted out of fashion during the shift of attention towards multimodal language models without having been finished.

What makes it worth more than a note about accuracy is what fine-grained recognition becomes when it is pointed at a large enough collection of images. A classifier for car make, model and year, run over some 50 million Google Street View images from 200 American cities, recovers the demographic composition of neighbourhoods: income, education level, race, and voting patterns all correlate strongly with what is parked on the street (Gebru et al., 2017). No person and no feasible team of people could conduct that survey. The census does it every ten years at enormous expense; this does it from imagery already collected for another purpose.

The pivot buried in that result is the one the rest of the chapter turns on. A model used this way is no longer a component inside a product — it is a **measuring instrument**, and the standards that apply to instruments are not the standards that apply to products. When a classifier that is 92 % accurate sits behind a photo-tagging feature, the remaining 8 % is an annoyance distributed across users. When the same classifier is an instrument, its errors are not noise: they are *measurement bias*, they are structured, they correlate with exactly the demographic variables being measured, and they propagate into whatever conclusion the measurement supports. The question stops being how often the model is right and becomes what its being wrong is systematically a function of.

19.3. Where human vision fails

Before the lecture turns to what machines get wrong, it spends time on what people get wrong, and the two failures it picks are of genuinely different kinds.

The first is capacity. Reading the word *red* is automatic, and naming the ink colour it is printed in is not, so a list of colour words printed in mismatched inks is slow and effortful to work through — the Stroop effect. In the same family, two versions of a photograph alternating with a brief blank between them can differ in something as large as a vehicle's engine, and observers hunt for many seconds before finding it. Change blindness of this kind is not a defect of the eye; it is a consequence of attention being a scarce resource that has to be pointed somewhere, so that what is not attended to is substantially not represented.

The second is systematic misperception. In Adelson's checkerboard, two squares printed at identical luminance appear obviously different, and — this is the part that matters — they continue to appear different after the illusion has been demonstrated, with the proof sitting in front of you. The visual system is not reporting the light arriving at the retina. It is reporting its best inference about surface reflectance given assumptions about illumination, shadow and the physics of the scene, and those assumptions are wired deep enough that knowing about them changes nothing.

These two failures point in opposite directions, and the lecture's remaining sections follow one each. Limited attention is a *capacity* problem, and a machine that does not tire is a genuine remedy for it — that thread runs into the clinic in Section 19.6. Systematic misperception is not a capacity problem, and no amount of additional machine attention fixes it, because the bias is in the inference rather than in the sampling. Bias of that second kind is what the lecture means when it says that human bias — inherited from evolution, from social experience, and

from whatever data a person happens to have encountered — becomes harmful at the point where it makes a system unfair to some group of people.

That sentence is where this chapter has to slow down, because it is the load-bearing claim of the lecture and it is left undefined.

19.4. What “unfair” would have to mean

The lecture’s concrete example is face analysis, and the canonical measurement behind it is Gender Shades, which evaluated three commercial gender classifiers on a benchmark constructed to be balanced across skin tone and gender. The headline is a disparity rather than an average: error rates reached 34.7 % on darker-skinned women, against a worst case of 0.8 % on lighter-skinned men. The lecture’s reading is that the field went from not noticing this to taking it seriously, which is true and is a real achievement of the last decade.

But suppose you accept the finding and set out to fix it. What, precisely, is the target? “Make the system fair” is not yet a specification, and the moment it is made into one, a difficulty appears that no amount of attention or engineering effort removes.

Everything follows from one object. Fix a group of people and a binary decision — flag or do not flag — made about each of them, and record the outcome against the truth in a two-by-two table.

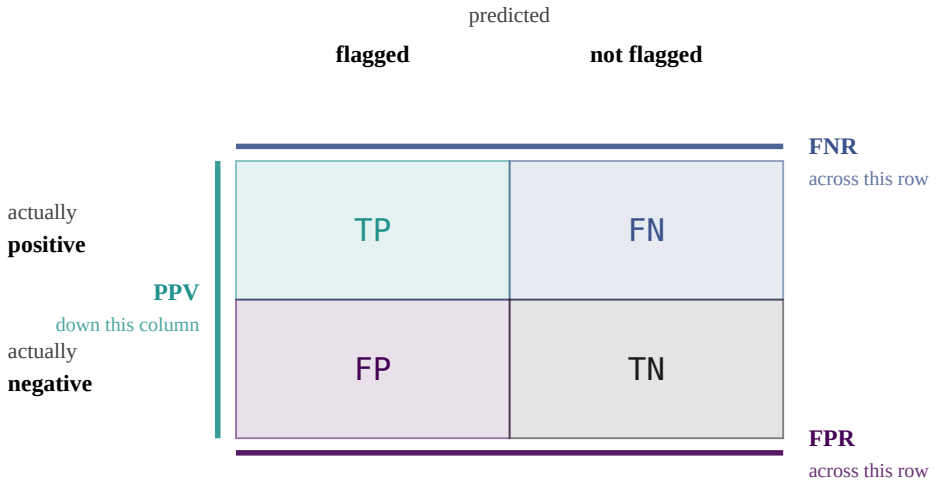


Figure 19.1.: The contingency table underneath every claim in this chapter. Three quantities are read from it, and each conditions on a different margin: the predictive value looks down the column of people the system flagged, the false negative rate looks across the row of people who were genuinely positive, and the false positive rate looks across the row of people who were not. Because all three are ratios of the same four cells, they are not independent — fixing two of them determines the third.

Write p for the prevalence of the condition in the group, that is, the fraction of its members for whom the truth is positive. Three quantities can be read off the table, and each corresponds to something a reasonable person means by fairness.

The first is the **positive predictive value**, $\mathrm{PPV} = P(Y = 1 \mid \hat{Y} = 1)$: among the people the system flagged, the fraction who genuinely were positive. Requiring it to be equal across groups is the demand that a flag *mean the same thing* whoever it is attached to. A decision-maker who cannot assume this has an explicit incentive to discount the score for members of one group, which is discrimination reintroduced at the point of use. In the fairness literature this is the coarse form of calibration, sometimes called test fairness.

The second is the **false negative rate**, $\mathrm{FNR} = P(\hat{Y} = 0 \mid Y = 1)$: among people who genuinely were positive, the fraction the system missed. Equalising it across groups is balance for the positive class.

The third is the **false positive rate**, $\mathrm{FPR} = P(\hat{Y} = 1 \mid Y = 0)$: among people who genuinely were not, the fraction wrongly flagged. Equalising it is balance for the negative class, and it is the criterion the Gender Shades style of audit is closest to, since it asks whether the burden of the system's mistakes falls evenly.

Each is defensible. Each is what somebody means. The difficulty is that they are three ratios built from the same four cells, so they cannot be set independently. Writing the predictive value out in terms of the other quantities,

$$\mathrm{PPV} = \frac{p\,(1 - \mathrm{FNR})}{p\,(1 - \mathrm{FNR}) + (1 - p)\,\mathrm{FPR}} \tag{19.1}$$

and solving for the false positive rate gives a relation due to Chouldechova (2017):

$$\mathrm{FPR} = \frac{p}{1 - p} \cdot \frac{1 - \mathrm{PPV}}{\mathrm{PPV}} \cdot (1 - \mathrm{FNR}) \tag{19.2}$$

This is worth being clear about: Equation 19.2 is not a statistical result, an approximation, or a claim about any particular model. It is an algebraic identity on the four cells, and it holds for every two-by-two table whatever produced it. The generator for this chapter checks it against direct cell ratios over 28,561 integer tables and finds the worst disagreement to be 1.4×10^{-15}, which is floating-point noise.

Now take two groups that differ in prevalence and ask for all three criteria at once. Equal predictive value and equal false negative rate mean the last two factors of Equation 19.2 are shared, so the false positive rates stand in the ratio of the *odds* of the two prevalences:

$$\frac{\mathrm{FPR}_1}{\mathrm{FPR}_2} = \frac{p_1/(1 - p_1)}{p_2/(1 - p_2)} \tag{19.3}$$

The third criterion is therefore not available to be chosen. It has already been determined by the first two together with facts about the world that the model does not control, and it takes the value the model wanted only when $p_1 = p_2$.

The general statement, for real-valued risk scores rather than binary flags, is due to Kleinberg, Mullainathan and Raghavan (2016). In their formulation the three conditions are calibration within groups, balance for the negative class, and balance for the positive class, with the balance conditions stated as equality of the *average score* assigned to the negative and positive members of each group. Their Theorem 1.1 is that any instance admitting all three must either allow

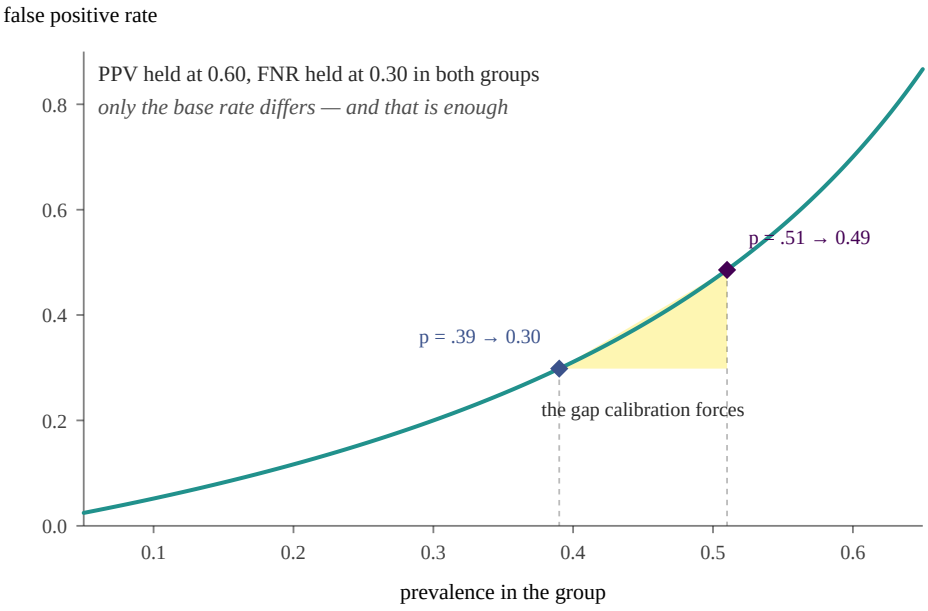


Figure 19.2.: The false positive rate implied by Equation 19.2 as prevalence varies, with predictive value and false negative rate held fixed at the same values in both groups. Nothing about the classifier differs between the two marked points — only the base rate of the population it is applied to — and the gap between them is what equalising the third criterion would have to erase. Beyond a prevalence of about 0.68 the implied rate exceeds one, which is to say no classifier with these properties exists there at all.

perfect prediction — every individual's outcome known with certainty in advance — or have equal base rates across groups. They also prove an approximate version, which closes the obvious escape route: satisfying the three conditions even approximately forces the instance to be approximately one of those two degenerate cases. There is no useful middle ground where a little unfairness on each axis buys a solution.

Sweeping the residual across pairs of base rates shows the shape of that result directly.

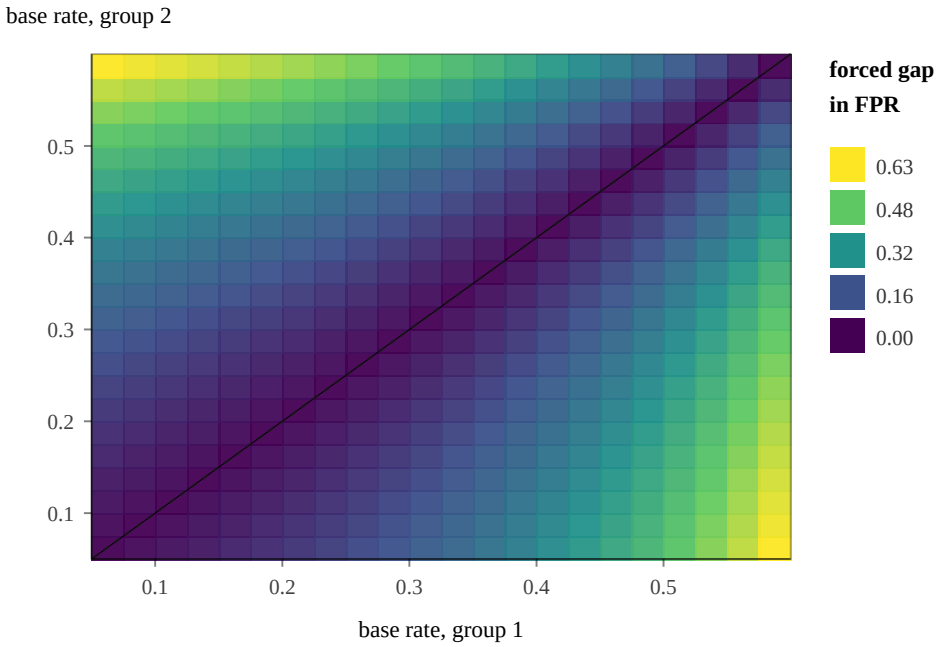


Figure 19.3.: The gap in false positive rate forced between two groups that share predictive value and false negative rate, as their two base rates vary. The zero set is exactly the diagonal. This is not a hard optimisation landscape with a narrow optimum — away from the diagonal there is no solution at all, and the smallest non-zero gap found anywhere off it in the swept grid is 0.005.

The clearest illustration is not synthetic. Chouldechova analyses the COMPAS recidivism score using the Broward County data released by ProPublica, where the two-year recidivism rate is 51 % among Black defendants against 39 % among White defendants. ProPublica's finding was that the score's error rates diverged sharply by race — a false positive rate of 45 % against 23 %, and a false negative rate running the other way at 28 % against 48 %. The instrument's developers replied that the score was calibrated. Both parties were correct, and Equation 19.2 is why.

Taking each group's published prevalence and false negative rate and computing the predictive value that must follow gives 0.625 and 0.591 — near enough equal to substantiate the developers' claim. Running the relation in the other direction is the more interesting check: assume the score is *exactly* calibrated at the average of those two values, use nothing but each group's prevalence and false negative rate, and predict what the false positive rates have to be. The answer is 48 % and 21 %, against published values of 45 % and 23 %. The error-rate gap that the dispute was

about is recovered to within about three points from the base rates alone, without any access to the model, its training data, or the error rates themselves.

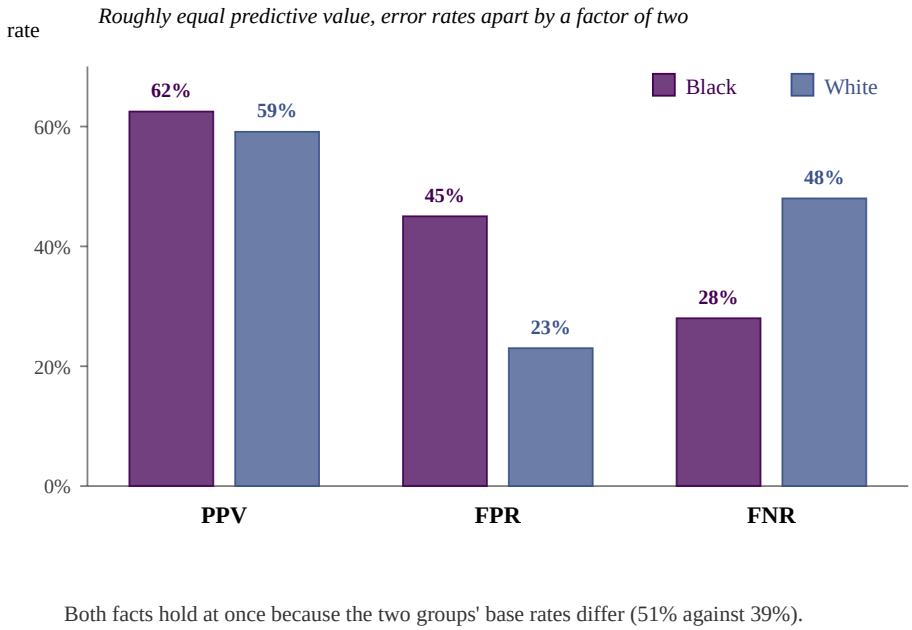


Figure 19.4.: The published COMPAS figures. The predictive values differ by 0.034 — the score means close to the same thing whoever it is attached to — while the error rates differ by roughly a factor of two in opposite directions. Both facts are consequences of the same table, given base rates of 51 % and 39 %.

One caveat belongs here rather than in a footnote, because it is easy to overstate this result. Equation 19.3 predicts a false positive ratio equal to the odds ratio of the base rates, which is 1.63, whereas the published ratio is 1.96. The two do not have to agree, because Equation 19.3 assumes the false negative rates are equal across groups and in COMPAS they are conspicuously not — 28 % against 48 %. The prediction quoted above uses each group's own false negative rate and is the honest version; the odds ratio alone would be the answer to a different question.

Two conclusions follow, and they pull in opposite directions, which is why both need stating.

The first is that a substantial part of what gets reported as algorithmic bias is not a defect that better engineering removes. When two populations differ in the base rate of the thing being predicted, and the predictor is anything short of perfect, the disparities are forced. Demanding that a vendor eliminate all of them at once is demanding something that provably does not exist, and a system audited against one criterion can be made to look scandalous by an audit against another without anything about it having changed.

The second is that this is not a defence of the Gender Shades numbers, and reading it as one inverts the argument. The theorem constrains how *near* to equality the three criteria can jointly be pushed; it says nothing that would license a forty-fold gap in error rate between demographic groups on a task where the label is a property of the image alone. That gap came from training and evaluation sets in which darker-skinned women were scarce, and it narrowed once it was measured, which is precisely the improvement the lecture is pointing at. The impossibility result and the audit are addressed to different failures: the audit catches the disparities that

were nobody's necessity, and the theorem tells you where to stop expecting the audit to be satisfiable.

What survives is that "fair" is not a property a model can be certified to have. It resolves into several incompatible properties, and choosing which of them a given deployment owes the people it is applied to is a question about consequences — whether a false flag or a missed case does more harm here, and to whom — which no amount of accuracy answers. The lecture is right that attention to the problem has grown. Attention produced the measurements; it cannot produce a joint solution that does not exist.

19.5. Designing for what must not be seen

There is a second sense in which a system can fail to see something, and here failure is the goal. A camera placed in a hospital room or a private home to watch for a fall, a missed medication or a hand not washed is, by construction, a camera pointed at people in the least guarded parts of their lives. The requirement is not to see less. It is to see one specific thing and nothing else.

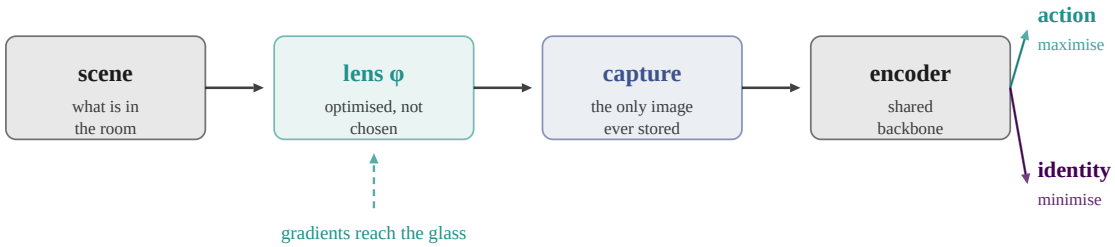
The obvious responses treat this as a post-processing problem: blur the image, mask out faces and bodies, reduce the dimensionality of what is stored, keep the data on the device through federated learning, or encrypt it. Some of these are genuinely useful, and the ones that operate on the *transmission* of data rather than its content — keeping it local, encrypting it in flight — are the least compromised, because they do not degrade the signal at all.

The ones that degrade the image, though, run into a structural obstacle that the lecture states plainly and that deserves to be taken seriously as a design constraint. What identifies a person and what reveals their activity are not separable channels in a photograph. Blur an image until the face is unrecoverable and you have usually blurred it past the point where the difference between reaching for a glass and reaching for a pill bottle survives. The privacy dial and the utility dial are, for naive degradations, the same dial.

The work the lecture singles out attacks that by moving the degradation upstream of the sensor. In PrivHAR, the optics themselves are a learned component: the lens is parameterised, and its parameters are optimised jointly with the recognition network, against an adversary trying to recover identity from whatever the lens produces. The optimisation is pushed towards a lens whose images support action recognition while defeating face recognition, and because the distortion is physical, it happens before anything is digitised.

The consequence is a different kind of guarantee from the usual one. Software that blurs faces is a promise about what a system does with the sensitive image it holds, and that promise can be broken by a configuration error, a software update, a subpoena, or a breach. A lens that never forms the sharp image makes the sensitive recording *not exist*, which is not a policy that can be revoked. That is what makes the approach interesting well beyond the specific accuracy numbers it reports: it converts a promise into a physical fact, and it is a rare instance of a privacy property being enforced by the hardware budget rather than by an audit.

The general lesson the lecture draws from this half of the talk is that the same capability cuts both ways. A model that extends perception can extend it past the point people would consent to; a model trained on human judgements inherits human biases and, applied at scale, amplifies them rather than averaging them out. Neither is a reason to stop building. Both are reasons that the design question — what should this system be able to see, and what should it be constitutionally unable to see — has to be asked while the system is being specified, because it is very difficult to retrofit.



The degradation happens in the optics, so the sensitive image is never captured — *there is no raw recording to leak, subpoena, or mishandle.*

Figure 19.5.: Privacy placed in the optics rather than applied afterwards. The lens parameters are optimised jointly with the recognition head and against an identity adversary, so gradients reach the glass itself. Because the degradation is physical, the only image that ever exists in the system is the already-degraded one.

19.6. The clinic and the base rate

The strongest case in the lecture for machine perception is the one where the human failure being remedied is the capacity failure from Section 19.3 rather than the bias failure. Attention is finite and shifts; a sensor's does not.

The setting is what the lecture calls the dark spaces of healthcare — operating rooms, patient rooms, homes — where consequential things happen continuously and are observed only intermittently, and where the shortfall is not expertise but coverage (Haque, Milstein and Fei-Fei, 2020). The pressure is demographic and will intensify: as life expectancy rises the population needing sustained care grows, while the working-age population available to provide it does not, and health systems already report chronic shortages of nurses. The argument for automation here is unusually clean, because the labour it would substitute for does not exist to be displaced.

The applications follow the same pattern. Hand hygiene compliance matters because hospital-acquired infections are a leading cause of inpatient death — the widely cited estimate is around 99,000 deaths a year in the United States, several times the toll of road accidents (Klevens et al., 2007) — and germs move between rooms on hands. Auditing compliance by having humans watch is unaffordable and, given the fatigue and attention limits already discussed, unreliable; proximity badges answer a different question, since standing near a dispenser is not using it. Depth sensors that resolve people as silhouettes rather than images, feeding an action classifier, answer the actual question while capturing much less than a camera would — the same instinct as Section 19.5, implemented more cheaply (Singh et al., 2020). In intensive care, where getting patients mobilised on schedule is a determinant of how quickly they leave, similar sensors track transfers between bed and chair (Yeung et al., 2019). In homes, the same apparatus is aimed at fall detection, sleep and mobility patterns, and early signs of infection.

The lecture also opens this section with a striking claim: that medical error is the third leading cause of death in the United States. That figure comes from an analysis by Makary and Daniel (2016), which extrapolated death rates from four earlier studies covering 2000 to 2008 across

all US hospitalisations in 2013 to reach 251,454 deaths a year. It is worth reporting accurately, because the estimate has been substantially criticised — the source studies were not designed to be generalised to the whole hospitalised population, several covered only Medicare patients, and the extrapolation applied their rates to every admission including uncomplicated childbirth. The direction of the claim is not seriously disputed: preventable harm in hospitals is common and badly measured. The specific ranking is an artefact of a rough calculation, and repeating it as an established statistic overstates what is known.

That distinction matters more than it looks, because measurement is exactly where these systems either earn their place or fail, and the arithmetic that decides it is the same table as Section 19.4 — now read inside one group rather than across two.

Take the lecture's own operating-room example. Instruments, sponges and needles are counted by hand against a checklist, and a discrepancy stops the operation while the item is searched for, with the pause averaging close to an hour of additional anaesthesia, exposure and bleeding. An automatic count is an obviously attractive target. But retained surgical items are, thankfully, extremely rare — the largest series put the rate at roughly one per 5,500 operations. Write π for that prevalence. Then the probability that a raised alarm is real is

$$\mathrm{PPV} = \frac{\mathrm{sens} \cdot \pi}{\mathrm{sens} \cdot \pi + (1 - \mathrm{spec})(1 - \pi)} \tag{19.4}$$

where sensitivity is the fraction of genuine retentions caught and specificity the fraction of clean operations correctly left alone. The behaviour of Equation 19.4 at small π is the whole problem. A detector at 99 % sensitivity and 99 % specificity — figures that would be reported as excellent — yields a predictive value of 0.018 at this prevalence: about 57 stoppages for every genuine retained item. At 99.9 % specificity it is 0.15, still roughly seven false stoppages per real one. Reaching an even chance that an alarm is real requires specificity above 99.98 %, and reaching 90 % requires 99.998 %.

None of this is an argument against the application, and it is not a claim that these systems do not work. It is a statement about which number decides whether they work. Sensitivity is what a demonstration shows off and what a benchmark rewards; at low prevalence it is nearly irrelevant to whether the system is usable, because the denominator of Equation 19.4 is dominated by false positives drawn from the overwhelming majority of cases where nothing is wrong. A system that stops one operation in a hundred to look for an item that is present once in 5,500 will be switched off within a month, and it will be switched off by people who are behaving entirely rationally. This is the same phenomenon that makes screening programmes for rare diseases difficult, and it sorts the applications in this section into two groups rather than condemning all of them. Falls, infection onset and retained instruments are rare relative to the number of opportunities to alarm about them, and all inherit the problem. Hand hygiene does not, and the contrast is instructive: the event being classified is whether a dispenser was used on a room entry, which happens on a large fraction of entries rather than once in thousands, so the reported operating point of 92.1 % sensitivity at 98.3 % specificity lands in the flat left-hand region of Equation 19.4 rather than the collapsing one. The system works partly because the question was posed at a favourable base rate — a design decision, and one that is easier to make deliberately than to recover from later.

The connection to Section 19.4 is not an analogy. It is the same four cells. Read across two groups they force a choice between incompatible notions of fairness; read against a low base rate they force attention onto the metric that a benchmark is least likely to be reporting. In both cases the failure mode is the same: a number that summarises the model gets mistaken for a number that describes the deployment.

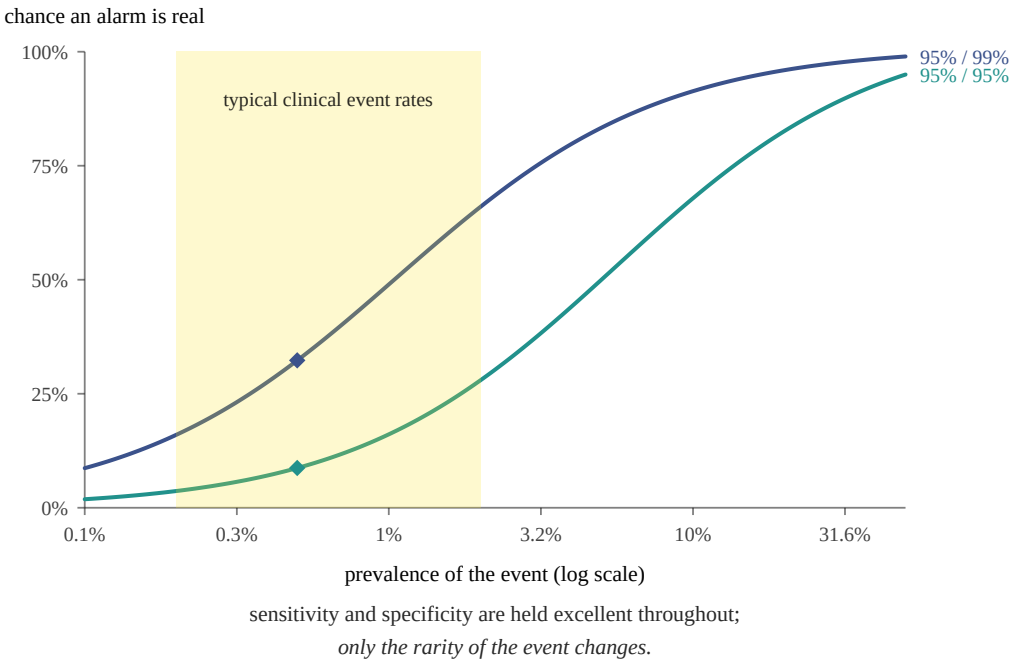


Figure 19.6.: Predictive value against the prevalence of the event, for two detectors whose sensitivity and specificity are held excellent throughout. Only the rarity of what is being detected changes. The shaded band covers the rates at which many clinical events of interest actually occur, and the collapse through it is why deployment is decided by specificity rather than by headline accuracy.

19.7. Letting people choose the task

Sensors gather information and cannot act on it, which is where the lecture turns to embodied systems — and where it poses the question that gives the chapter its title, in its sharpest form. How such systems are trained is the subject of the previous chapter. The question here is prior to that: who decides what they should be trained to do?

The default answer is that researchers decide, and the default has a recognisable signature. Benchmark tasks tend to be the ones that are convenient to simulate, easy to score, and salient to graduate students, which is why so much of the field converges on laundry and dishes. This is not obviously wrong, but it is unexamined, and it quietly determines what the technology will be good at years before anyone outside the field is consulted.

BEHAVIOR-1K inverts the order of operations. Its task list was assembled by starting from government labour and time-use surveys to enumerate thousands of everyday activities, then asking around 1,400 people which of them they would actually want a robot's help with, and ranking by the answers. The resulting thousand tasks are the ones people said they wanted done, and they are supported by fifty scanned real environments — homes, offices, restaurants, shops — and thousands of object assets annotated with the physical properties the activities depend on, such as articulation, deformability and temperature.

What makes the survey interesting is not the top of the ranking, which is unsurprising: cleaning, in most of its forms, is work people are glad to hand over. It is the bottom. People did not want a robot to buy a wedding ring, to play a sport on their behalf, to open a gift, or to prepare an infant's food — and in several of these cases the task is well within reach technically. The refusal is not about feasibility. Some activities carry their meaning in the fact that a person did them, and automating them destroys the thing they were for.

That is a finding no amount of capability research would have produced, because it is not a fact about robots. It is the same structural point as Section 19.4 and Section 19.6 arriving from a third direction: the objective is not derivable from the system. What to optimise, which errors to tolerate, what must remain unseen, and now which tasks are wanted at all — each is a question whose answer has to come from the people affected, and each is a question a benchmark will silently answer by default if nobody asks it.

19.8. Augmentation

The lecture closes on labour, and refuses both of the available slogans. Denying that these systems will change what work looks like is not credible; every general-purpose technology has reorganised labour markets, sometimes with severe and unevenly distributed costs. But the specific prediction that has dominated public discussion — that physical work goes first — has not been how the past few years have run. The measurable disruption so far has landed on white-collar work: software, analysis, writing, the office tasks that were assumed to be furthest from automation.

Against that, the sectors this lecture spends most of its time on are ones where the shortage is real and structural. Elder care, nursing and chronic disease management are short of people now, and demographic arithmetic makes them shorter every year. Machine perception aimed at those settings is not competing with a workforce; it is compensating for one that does not exist and cannot be conjured, and the honest framing of the surgical count, the hygiene monitor and the mobility tracker is that each returns some attention to people who have too little of it.

That is the argument for augmentation over replacement, and the reason it is more than a comforting phrase is that every section of this chapter has run into the same wall from a different side. The fairness criteria could not all be satisfied, so someone has to choose which one the decision owes its subjects. The rare-event detector could not be evaluated by accuracy, so someone has to decide what rate of false alarm a ward will tolerate and for what benefit. The privacy-preserving camera could not be specified without deciding what it must be unable to see. The task benchmark could not be assembled without asking which tasks people want performed. In none of these cases was the missing ingredient a better model, and in every one of them the missing ingredient was a human judgement about consequences that the metric could not supply.

That is what the phrase is pointing at. Not that machines should be kept subordinate for reassurance, but that the systems in this chapter are all incomplete in the same specific way: each computes something, and none of them can tell you what it should have been computing. Which is where a course on how to build them has to stop, and where the responsibility for pointing them somewhere begins.