A History of Machine Learning

Bottlenecks, Representations, and the Question of Grounding

Flyxion

Independent Researcher

August 2026

Contents

The Argument Before the Field

In 1766, a respected philosopher in Königsberg sat down to write about a man he could not stop thinking about.

The man was Emanuel Swedenborg, a former scientist and engineer who had spent his early career on canals, mining machinery, and treatises on metallurgy and cosmology. In midlife, something changed. Swedenborg began reporting visions: encounters with angels, journeys through spiritual worlds, and most famously, a claim that he had witnessed a fire in Stockholm from three hundred kilometers away, in real time, while sitting at a dinner party in Gothenburg. Word of the episode spread across Europe. It reached Immanuel Kant.

Kant was intrigued, then troubled. He bought Swedenborg's sprawling eight-volume *Arcana Coelestia* at considerable expense and read it looking for evidence, one way or the other, of whether the visions were real. What he found instead was something harder to categorize: a system. Swedenborg's spiritual world was not a jumble of disconnected hallucinations. It was internally coherent, richly detailed, and endlessly extensible. Every new vision fit the architecture of the ones before it. Correspondences bred further correspondences. Nothing in it was obviously self-contradictory. And none of it could be checked against anything outside itself.

Kant's response, *Dreams of a Spirit-Seer*, is a strange piece of writing. On the surface it mocks Swedenborg. Underneath, it is doing something more careful, working toward a single question: when does internally coherent reasoning count as knowledge, rather than merely fluent invention? That question did not leave Kant. Fifteen years later it resurfaced, transformed, as the *Critique of Pure Reason*, the book that tried to specify, once and for all, the conditions under which a claim to knowledge is entitled to be believed.

This book is not about Kant or Swedenborg. It is about machine learning. But it opens here because the question they were fighting over, in 1766, is the same question that has driven nearly every major development in this field, right up to the present: when does internally coherent reasoning count as knowledge?

Two Ways to Go Wrong

Any system capable of producing rich, structured output faces two very different failure modes.

The first is a system that cannot produce enough. It is too constrained, too literal, too dependent on rules someone wrote down in advance. It fails by having nothing to say about situations its designer did not anticipate. Symbolic AI's expert systems, built from thousands of hand-coded if-then rules, failed this way. They were precise and they were brittle. The world kept presenting cases the rules did not cover.

The second failure mode is the opposite, and it is the one Kant was staring at when he read the *Arcana Coelestia*. It is a system that can produce almost anything: fluent, internally consistent, richly elaborated, and only loosely tethered to anything outside itself. Give it a pattern

and it will extend that pattern indefinitely, whether or not the extension is true. Modern language models, when they hallucinate, are doing something structurally similar to what Kant suspected Swedenborg's imagination was doing: sampling forward from an internal model of what a plausible next thing looks like, unconstrained by an external check on whether that next thing actually happened.

Swedenborg was not lying, in the ordinary sense. He was, by every account, sincere. His visions were the output of an extraordinarily generative mind operating without a verification loop back to shared, external reality. That is the useful, non-mystical way to read him: not as a fraud and not as a genuine seer, but as an early, human-scale example of what unconstrained generation looks like when it is fluent enough to be convincing.

Kant's answer was not to demand more rules, in the manner of the symbolic systems that would fail two centuries later. His answer was to ask a different kind of question entirely: not "is this specific claim true," but "what are the general conditions any claim has to meet, before it is even eligible to count as knowledge." Space, time, and causality, in his account, are not things we find in the world. They are the structure a mind imposes on raw experience in order to make experience navigable at all. Reason, for Kant, is not a longer list of facts. It is an architecture that decides which internally coherent stories are allowed to stand.

Why This Belongs in a History of Machine Learning

Every architecture discussed in this book, from the perceptron to the transformer, sits somewhere on the line between these two failure modes. Each one solves a specific bottleneck: a limit on what the previous generation of models could represent, learn, remember, compute, or verify. And nearly every solution to one bottleneck creates or exposes another, in the same way that Kant's boundary-drawing solved the problem of unconstrained metaphysical speculation while creating a new, harder problem: how do you build a system expressive enough to be useful, without it also being expressive enough to make anything sound plausible. Read that pattern carefully enough, across enough chapters, and a stronger claim starts to look unavoidable: every mechanism that increases what a system can express eventually creates a new problem of whether that expression can be trusted. This book's last chapter returns to that claim directly, once there are sixteen chapters of evidence behind it instead of one historical argument.

That tension, expressiveness against grounding, generative capacity against verification, is not a modern invention arriving with GPUs and internet-scale text corpora. It is the oldest problem in the study of minds, artificial or otherwise. Kant and Swedenborg give it a human face and a specific date: an argument conducted in letters and treatises across the 1760s and 70s, between a man who saw too much and a man who was afraid of seeing anything he could not later justify.

The chapters that follow trace how the field of machine learning has, generation after generation, reinvented versions of that argument: sometimes as a fight between symbolic rules and statistical learning, sometimes as a fight between raw generative capacity and the alignment techniques meant to constrain it, sometimes as a quieter, more technical question about whether a model's internal representation of the world is answerable to anything outside itself. The specific names change. The underlying question, the one Kant could not put down

after reading Swedenborg, does not.

Chapter 1

Two Ways for a Rule-Based System to Fail

Machine intelligence's earliest serious attempt rested on an assumption worth stating plainly before testing it: that reasoning which is exact must also be reasoning that scales, that getting every step provably right is a matter of the same discipline as getting to an answer at all. This chapter tests that assumption directly, with a program small enough to run yourself, and finds it false in a specific, measurable way.

What Expert Systems Promised

By the early 1980s, a new kind of software had convinced a great many serious people that machine intelligence was close at hand. Systems like MYCIN, built to recommend antibiotic treatments, and DENDRAL, built to infer chemical structures from mass spectrometry data, did something that looked, from the outside, exactly like expertise. You gave them facts. They gave you conclusions, and they could explain, step by step, exactly how they got there. No black box, no guessing. Every inference could be traced back through an explicit chain of rules to the facts that triggered it.

This was the core appeal of what came to be called expert systems: a large collection of if-then rules, hand-written by engineers working alongside human domain experts, chained together by an inference engine that applied them mechanically and exhaustively until nothing new could be derived. If the rules were correct, the conclusions were guaranteed to be correct. There was no approximation, no statistical hedging, no sense in which the system might be "mostly right." It was either a valid inference or it wasn't.

That guarantee is worth taking seriously, because it is genuinely valuable and because later chapters will spend a great deal of time on architectures that give it up entirely. It is worth seeing, concretely, what exact symbolic inference looks like and why people believed it would scale.

A Small Working Example

The program below is a Maxima script, `non_sequitur.mac`. It does one thing: given a list of premises and a proposed conclusion, all written in ordinary propositional logic, it determines whether the conclusion actually follows. If it does, the program says so. If it doesn't, the program hands back a counterexample: a specific assignment of true and false to every variable involved, under which every premise holds and the conclusion still fails.

```
non_sequitur(
    [impl(p,q), p],
    q
```

```
);
```

This is modus ponens. The program checks every possible assignment of `p` and `q`, finds that whenever `impl(p,q)` and `p` are both true, `q` is true as well, and reports the argument valid. No exception exists, so none is found.

```
non_sequitur(
    [impl(p,q), q],
    p
);
```

This looks similar but isn't. It is the classic fallacy of affirming the consequent. The program searches the same small space of assignments and this time finds a counterexample: `p` false, `q` true. Both premises hold under that assignment, `impl(p,q)` is true because a false antecedent makes any implication true, and `q` is true by assumption, but the supposed conclusion `p` is false. The argument is invalid, and the program can show you exactly why, in one line.

Run it yourself and you'll see the appeal immediately. There's no ambiguity to argue about, no sense in which the system is being evasive or approximate. It is either right or provably wrong, and when it's wrong it hands you the exact case that breaks it.

How the Program Actually Works

Under the hood, `non_sequitur` does nothing clever. It collects every propositional variable appearing anywhere in the premises or the conclusion, then walks through every possible combination of true and false values those variables could take. For each combination, it checks whether all the premises come out true. If they do, it checks the conclusion. The first time it finds a combination where every premise holds but the conclusion fails, it stops and reports that case as a counterexample. If it exhausts every combination without finding one, the argument is valid.

This is exhaustive verification in the most literal sense. The program doesn't reason about the structure of the argument the way a human logician might, spotting the fallacy pattern by eye. It simply tries everything.

Where the Guarantee Breaks Down

Here is the problem, and it is visible the moment you count. If an argument involves `n` distinct propositional variables, there are `2^n` possible combinations of true and false to check. Ten variables: 1,024 combinations, instantaneous. Twenty variables: over a million, still fast on modern hardware. Forty variables: over a trillion. Every additional variable doubles the work. This is combinatorial explosion, and it is not a flaw in this particular implementation — it is a property of exhaustive search over propositional assignments. No amount of clever coding makes `2^n` grow any slower.

A real diagnostic or planning system doesn't reason about two or three variables. It reasons about hundreds or thousands of facts, symptoms, and rules interacting at once. Exhaustive

truth-table verification of an argument at that scale is not merely slow. It is not something any computer, now or in the foreseeable future, will finish before the sun burns out.

This is worth separating cleanly from a second, unrelated problem that expert systems also ran into, because the two are often blurred together and they call for entirely different fixes.

The first problem, the one `non_sequitur.mac` makes visible, is computational: exhaustive reasoning over a large rule base costs more than any amount of hardware can pay for. The second problem is about where the rules come from in the first place. MYCIN's rules had to be extracted, one at a time, through extensive interviews with physicians, then encoded, tested, and maintained by engineers as medical knowledge changed. This is the knowledge acquisition bottleneck, and it has nothing to do with how fast a computer can search. A system with all the computational power in the world is still only as good as the rules a human bothered to write down, and every rule not written down is a case the system simply cannot handle. Expert systems were narrow not because search was slow inside their domain, but because outside their domain there was no knowledge at all, encoded or otherwise.

Keep these two failures distinct. The chapters ahead will show statistical and neural approaches solving the knowledge acquisition problem, learning patterns from data instead of requiring an engineer to write each one down by hand, largely independently of how those same approaches handle computational cost. Conflating the two failures now would blur a distinction that matters again later, when deep networks solve one of these problems dramatically and the other, in a different form, resurfaces as a live concern in the largest language models.

A Sidebar: When the Verifier Needs Verifying

There is one more lesson worth drawing out before moving on, and it did not need to be invented for the purpose. The first version of `non_sequitur.mac` used to teach this chapter did not work correctly. It contained two bugs, and both are worth walking through, because they say something sharper about exact reasoning than any hypothetical example would.

The first bug was immediate and loud. The function that collects a formula's variables passed ordinary lists to Maxima's `union` operation, which requires actual sets. The program crashed on the first example, before it could check a single valuation. This kind of failure is the easy kind: it announces itself, and it gets fixed before anyone mistakes the program's silence for correctness.

The second bug was quieter, and more dangerous for exactly that reason. Three separate functions used the pattern "loop through a list, and return early the moment you find what you're looking for, otherwise fall through to a default." In Maxima, an early `return` inside a `for` loop only escapes the loop, not the surrounding block, unless that loop happens to be the very last statement the block contains. In all three functions here, it wasn't. There was always a fallback statement written after the loop: an error message, a default value of `true`, a block that prints "Result: VALID." Every one of those fallbacks silently overrode the early return. The practical effect was that the function meant to check whether every premise holds under a given valuation always reported `true`, no matter what the premises actually said, and the main search loop, even after finding and printing a genuine counterexample, would fall through and

announce "Result: VALID" immediately afterward, contradicting the counterexample it had just shown you.

That is worth sitting with. A program built specifically to demonstrate exact, mechanical, unambiguous inference was, until corrected, silently wrong about whether its own inferences were valid. It never crashed. It never complained. It printed confident, well-formatted output every time, including output that flatly contradicted itself two lines apart. Nothing about propositional logic was at fault. The rules the program was checking were exactly the rules described earlier in this chapter, and they were exactly right. The failure lived entirely in the software implementing those rules, in a control-flow detail so easy to miss that it survived being written, read, and executed against three worked examples before showing up as a contradiction.

This is the deeper version of the chapter's point. Exact symbolic reasoning gives you a guarantee about the relationship between premises and conclusions inside a formal system. It gives you no guarantee at all about whether the machine executing that reasoning is doing what its author believes it is doing. Trusting a system because it is deterministic and rule-based, rather than statistical and approximate, only pushes the trust problem down one level, to the correctness of the rules and the code that applies them. That problem does not go away as the field moves toward learned, statistical methods in the chapters ahead. It changes shape, and in some ways it gets harder, because a model's internal reasoning is no longer written down in a form a human can read line by line and check against a specification. But it was never absent, not even here, not even in the simplest, most exact, most explainable kind of system this field has produced.

A Postscript on Modern Solvers

It would be a mistake to conclude from any of this that exhaustive symbolic reasoning was simply a dead end that statistical learning made obsolete. The truth is closer to the opposite: symbolic reasoning kept improving on its own track, and that track is still active and important today.

`non_sequitur.mac` checks every one of the `2^n` assignments because that is the clearest possible way to teach the idea. It is not how a serious modern solver works. Beginning with the DPLL algorithm in the 1960s and maturing into the CDCL (conflict-driven clause learning) solvers used today, symbolic satisfiability engines learned to avoid exploring most of the search space rather than avoiding the problem's underlying difficulty. Unit propagation forces the values of variables that are already determined by the current partial assignment, without guessing. Clause learning records the specific combination of choices that led to a contradiction, so the solver never repeats that exact mistake again. The worst-case complexity of Boolean satisfiability is still, and will likely always be, NP-complete: no known algorithm avoids exponential blowup in the worst case. But the practical difference is enormous. Modern SAT and SMT solvers routinely handle problems with millions of variables, verifying hardware designs, checking software for security vulnerabilities, and solving planning problems that a brute-force truth table, run on any computer that will ever be built, could never finish.

The lesson to carry forward is not that symbolic methods failed and statistical methods won. It is that exact reasoning and scalable reasoning are different properties, satisfied independently,

and this chapter's program had one without the other; exhaustive search and clever search are different things, and the history of both symbolic and statistical AI is, in large part, a history of finding ways to avoid looking at most of a search space while still finding what you're looking for in it. That theme returns, in a very different guise, when we reach scaling laws and the training of very large models.

Chapter 2

The Perceptron

The Bottleneck It Answered

Chapter 1 ended with a working diagnosis. Exhaustive symbolic reasoning is exact, but it does not scale, and expert systems on top of it were only as capable as the rules a human sat down and wrote out by hand. That second problem, not the computational one, is where this chapter picks up, and it splits into two separate questions that are easy to conflate. Can a rule be learned instead of hand-written? And, separately, can the system in question even express that rule at all, once found? This chapter answers the first question yes, cleanly, and then discovers the second question has its own, entirely independent answer.

In 1958, Frank Rosenblatt proposed a device that sidestepped hand-written rules entirely. Instead of asking an engineer to specify the conditions under which an input belonged to a category, the perceptron would look at labeled examples and adjust its own internal weights until it got them right. The rule wouldn't be written. It would be found.

What a Perceptron Actually Is

Strip away the biological metaphor and a perceptron is a short piece of arithmetic. Given an input vector, multiply each component by a weight, add a bias term, and check whether the total is positive. If it is, output one category; if not, output the other. That's the entire model at inference time. Learning happens through a rule almost as simple: run an example through the current weights, compare the output to the correct label, and if they disagree, nudge every weight in the direction that would have made the correct answer more likely. Repeat over the whole dataset, multiple passes, until nothing is left to correct.

```
class Perceptron:
    def __init__(self, n_inputs, lr=0.1):
        self.w = [0.0] * n_inputs
        self.b = 0.0
        self.lr = lr

    def activate(self, x):
        s = sum(wi * xi for wi, xi in zip(self.w, x)) + self.b
        return 1 if s > 0 else 0

    def train(self, data, epochs=20):
        for epoch in range(epochs):
            errors = 0
            for x, y in data:
                pred = self.activate(x)
                err = y - pred
```

```
                if err != 0:
                    errors += 1
                    for i in range(len(self.w)):
                        self.w[i] += self.lr * err * x[i]
                    self.b += self.lr * err
            if errors == 0:
                return epoch + 1
        return None
```

Forty lines, no libraries, nothing hidden. Train it on the logical AND function, four examples, and watch what happens.

```
Training perceptron on AND
  epoch 1: 1 misclassified, weights=[0.1, 0.1], bias=0.1
  epoch 2: 3 misclassified, weights=[0.2, 0.1], bias=0.0
  epoch 3: 3 misclassified, weights=[0.2, 0.1], bias=-0.1
  epoch 4: 2 misclassified, weights=[0.2, 0.2], bias=-0.1
  epoch 5: 1 misclassified, weights=[0.2, 0.1], bias=-0.2
  epoch 6: 0 misclassified, weights=[0.2, 0.1], bias=-0.2
  Converged after 6 epochs. Final accuracy: 4/4
```

Six passes over four examples, and the perceptron has found weights that correctly implement AND, without anyone telling it the rule "output 1 only when both inputs are 1." It found that rule by trial and correction. OR converges even faster, in four epochs. Nobody wrote if-then logic for either of these. The network discovered a linear rule that happened to match one.

This is worth pausing on, because it is a genuinely different way of building a system than anything in Chapter 1. The expert system's designer had to already know the rule and transcribe it. The perceptron's designer only had to supply correctly labeled examples and a procedure for adjusting weights. Knowledge acquisition, the second bottleneck from Chapter 1, stops being a transcription problem and becomes a data collection problem. That shift, learn the rule from examples rather than write the rule down, is the single idea that the rest of this book is, in one form or another, an extended elaboration of.

The Convergence Guarantee

The perceptron's early reputation rested on more than empirical success on toy examples. Rosenblatt, and later Novikoff, proved something stronger: if a dataset actually can be separated by some straight line (or, in higher dimensions, some flat hyperplane), the perceptron learning rule is guaranteed to find such a line in a finite number of steps. Not "usually." Not "with high probability." A hard guarantee, provable in a few lines of linear algebra, that training will terminate at a correct answer whenever a correct linear answer exists.

That guarantee is why the perceptron generated so much excitement. It combined the appeal of exact, provable behavior, the same appeal that made expert systems attractive, with the very different appeal of learning from data instead of requiring hand-specified rules. It looked, briefly, like the best of both worlds.

Where the Line Runs Out

Run the identical code, unmodified, on a fourth function: exclusive or, XOR. It outputs one when its two inputs differ, zero when they agree.

```
Training perceptron on XOR
  epoch 1: 2 misclassified, weights=[-0.1, 0.0], bias=0.0
  epoch 2: 3 misclassified, weights=[-0.1, 0.0], bias=0.1
  epoch 3: 4 misclassified, weights=[-0.1, 0.0], bias=0.1
   ...
  epoch 20: 4 misclassified, weights=[-0.1, 0.0], bias=0.1
  Converged: None. Final accuracy: 2/4
```

It does not converge. Not at 20 epochs, not at 20,000. The weights don't drift toward a solution and stop improving; run it longer and the accuracy simply oscillates, never settling above half the examples. This isn't a bug in the code and it isn't a matter of tuning the learning rate or running more epochs. Plot the four points of XOR on a two-dimensional grid, one input on each axis, and color them by output. The two positive examples sit on opposite corners from each other, and so do the two negative examples. No single straight line has all the positives on one side and all the negatives on the other, because none exists. The perceptron's convergence guarantee is conditioned on exactly this possibility, and XOR is the simplest possible function that lacks it.

Marvin Minsky and Seymour Papert made this observation precisely, and at length, in their 1969 book *Perceptrons*. Their argument was not that Rosenblatt's device was poorly designed or poorly trained. It was a proof: certain functions, XOR chief among them, cannot be represented by a single-layer perceptron at all, regardless of how the weights are set. No amount of additional training data or additional training time closes a gap that is representational rather than statistical. The perceptron is not failing to find the rule. There is no linear rule to find.

The Bottleneck This Exposes

Chapter 1 showed a system limited by how much knowledge a human could hand-encode. The perceptron dissolves that specific limit: yes, a rule can be learned instead of written down, and the AND and OR examples above prove it directly. But learnability and representability turn out to be separate ceilings, satisfied or violated independently of each other. A single-layer perceptron can only represent decision boundaries that are straight lines (or their higher-dimensional equivalent, hyperplanes), and no learning procedure, however good, raises that ceiling. Anything that requires a more complex boundary, XOR, and the far more complex boundaries most real-world categories actually have, sits permanently out of reach, not because it wasn't learned well, but because it isn't there to be learned.

Minsky and Papert's book is often, and somewhat unfairly, credited with single-handedly causing the collapse in neural network funding and research interest that occupies the next chapter. The fuller picture involves funding politics, unmet expectations from machine translation projects, and a general Cold War-era retreat from ambitious AI claims. But the mathematical result at the center of it was real, and it identified a genuine representational ceiling, not a temporary engineering inconvenience. The path past that ceiling, adding a second layer

of learned weights between input and output, was known in principle within a few years. What was missing was an efficient way to train it. That gap, and how it got closed, is where Chapter 4 picks up, once Chapter 3 has walked through what happened to the field in the meantime.

Chapter 3

The AI Winter

A Different Kind of Chapter

The first two chapters each ended at a technical wall. Exhaustive symbolic reasoning ran into combinatorial explosion. The perceptron ran into a representational ceiling it couldn't cross with any amount of additional training. This chapter looks, on the surface, like a different kind of chapter, one about funding, reputation, and institutional support rather than mathematics. It isn't, and that's the point worth stating before the history: both of the collapses this chapter covers trace back to a specific, nameable technical limit from the first two chapters, not to a field simply falling out of fashion. The question this chapter answers is whether AI's history is a story of failed vision or a story of predictable technical bottlenecks arriving on schedule and colliding with promises that had outrun them. The evidence points firmly at the second.

It's tempting to read AI's history as a straight line: perceptron fails, the field waits, backpropagation arrives, deep learning follows. The actual sequence is messier and more useful to know, because the pattern it reveals, ambitious promises outrunning what the current bottleneck-resolution actually allows, then the funding retreat that follows, then a quieter period in which the next resolution gets worked out largely unnoticed, is not unique to the 1970s and 80s. It has recurred, in compressed form, several more times since, and later chapters will point back to this one when it does.

The First Winter

Two reports did most of the damage.

In 1966, a US government committee called ALPAC evaluated a decade of heavily funded machine translation research, most of it aimed at automatically translating Russian scientific and military text for Cold War intelligence purposes. The committee's actual findings were more measured than their reception; ALPAC still saw value in machine-assisted translation and recommended continued work in computational linguistics generally. What survived in the public and political memory was a blunter message: fully automatic translation had not delivered on ten years of promises, and continued funding at the current level was hard to justify. American machine translation research went dark for most of the next two decades.

Three years later, Minsky and Papert's *Perceptrons* supplied a second, more theoretical blow, the one Chapter 2 walked through directly. It didn't kill research into learning machines by itself, but it gave funders and skeptics a precise, mathematically airtight reason to doubt that the perceptron's simple learning rule was a path to anything more general. A representational ceiling, proven rather than merely observed, is a hard thing to argue with.

Then, in 1973, the British government commissioned applied mathematician Sir James Lighthill,

who had no background in AI research, to assess the field for the Science Research Council. His report concluded that AI had failed to produce the major practical impact its proponents had promised, and it singled out combinatorial explosion, the same wall Chapter 1 described in `non_sequitur.mac`, as a fundamental barrier the field had not found a way around. The British government responded by cutting AI funding across nearly every university department working on it. Researchers left the field or left the country. The report's influence didn't stop at the English Channel; it gave American funding skeptics ammunition too, and in 1974 DARPA, which had been AI research's largest and most patient funder through the 1960s, began sharply reducing its support.

By the mid-1970s, AI research in both countries had lost most of its institutional funding, and "artificial intelligence" had become, for a working researcher, a label that made it harder rather than easier to get a grant approved. This period, roughly 1974 to 1980, is what's usually meant by the first AI winter.

An Irony Worth Naming Early

Here is a detail that rarely makes it into the shorter tellings of this story, and it's worth sitting with before moving on, because Chapter 4 depends on it. The mathematical technique that would eventually let networks with more than one layer be trained efficiently, resolving exactly the representational ceiling Minsky and Papert had proven, was first worked out, in close to its modern form, by Paul Werbos in his 1974 Harvard doctoral thesis. That is the same year DARPA began its retrenchment. The solution to the problem that had helped freeze the field's funding existed in a dissertation on a library shelf while the freeze was setting in. It would take more than a decade, and a different set of researchers working largely independently, before the technique was rediscovered, popularized, and connected back to neural networks widely enough to matter. Funding cycles and technical progress do not always move together, and this is the first of several points in this book's history where the useful idea was already sitting somewhere, waiting for the field's attention to come back around to it.

The Second Boom and the Second Winter

AI funding and enthusiasm did recover, but not by returning to neural networks. The 1980s boom was built on the expert systems described in Chapter 1. XCON, deployed at Digital Equipment Corporation starting in 1978 to configure VAX computer orders, was the clearest commercial success story the field had yet produced, reportedly saving the company tens of millions of dollars a year by encoding the expertise needed to translate a customer order into a correct, buildable system configuration. Stories like that one drove real investment. Companies built and sold expert system "shells," specialized Lisp machines optimized to run this kind of software commanded premium prices, and by the mid-1980s the AI industry had grown from a niche research pursuit into a market worth well over a billion dollars.

It did not last. Two separate problems converged around 1987. The first was economic: general-purpose workstations from Apple, Sun, and IBM caught up to and then undercut the price and performance of specialized Lisp hardware, and the market for that hardware collapsed within a couple of years, taking several companies down with it. The second was the same knowledge

acquisition bottleneck from Chapter 1, now showing up at commercial scale. Expert systems built from thousands of rules were expensive to maintain, could not learn from new cases on their own, and broke, often badly, the moment a situation fell outside the narrow domain their rules had anticipated. Companies that had invested heavily discovered the maintenance cost of keeping a large rule base current, in constant collaboration with expensive human domain experts, exceeded the value the system provided. DARPA cut funding again. The second AI winter, running roughly from 1987 into the early 1990s, was in some ways deeper and more discouraging than the first, because it followed a period of genuine, well-documented commercial success rather than merely unmet promises.

What This Chapter Sets Up

Two winters, separated by about a decade, driven by two different technologies and two different failure modes, funding collapse after unmet promises the first time, funding collapse after real but unmaintainable success the second time. It would be easy to read this as a field that simply couldn't get out of its own way. The more accurate reading is narrower and more specific: both winters followed directly from one of the two bottlenecks named in the first three chapters of this book, computational explosion in the first case, knowledge acquisition in the second, colliding with expectations that had outrun what either paradigm could actually deliver at scale.

Neither winter meant research stopped entirely. Small groups kept working on neural networks through the years when the label was most unfashionable, refining exactly the technique Werbos had described in 1974. Chapter 4 picks up there, with the paper that finally brought that technique to the field's attention and showed, concretely, that a network with more than one layer could learn to solve problems a single-layer perceptron provably never could.

Chapter 4

Backpropagation and Multilayer Networks

The Question Left Open

Does a second layer of weights add power the way more rope adds length, simply by being more of the same thing, or does it add power in some qualitatively different way? Chapter 2 ended with a proof, not a suggestion: no single-layer perceptron can represent XOR, because no straight line separates its positive examples from its negative ones. Adding a second layer of learned units between input and output was known, even at the time, to remove that ceiling in principle. A big enough network with a hidden layer can represent XOR, and a great deal more besides. Minsky and Papert said as much themselves in 1969. What they did not have, and what nobody else had in a practical, general form for another decade and a half, was an efficient way to train such a network.

The difficulty is not obvious until you try to state it precisely. Training a single-layer perceptron is straightforward because you always know exactly how wrong the output was, and you can nudge the one layer of weights directly in the direction that would have helped. Add a hidden layer, and the units in that hidden layer have no target of their own to be wrong about. Nobody labeled the correct hidden-layer activations. The only error you can measure sits at the very end of the network, after two layers of transformation, and the question this chapter answers is how to take that single number, the output error, and turn it into a specific, useful correction for every weight in both layers, including the ones with no direct connection to the output at all.

Credit Assignment

This is usually called the credit assignment problem: when a system with many internal parts produces a wrong answer, which parts deserve to be blamed, and by how much. Backpropagation solves it with an application of the chain rule from calculus, propagating the output error backward through the network, one layer at a time, converting "the final answer was off by this much" into "this specific hidden unit contributed to that error in this specific way."

The mechanics are more approachable in code than in the notation. Below is a complete, minimal multilayer network: two inputs, one hidden layer of two units, one output, using the same kind of sigmoid activation that dominated the field for its first few decades.

```
def sigmoid(x):
    return 1.0 / (1.0 + math.exp(-x))

def dsigmoid(y):
    return y * (1.0 - y)   # derivative, given the sigmoid's own output
```

```
class MLP:
    def __init__(self, n_in, n_hidden, n_out, lr=0.5):
        self.lr = lr
        self.w1 = [[random.uniform(-1, 1) for _ in range(n_in)] for _ in
        range(n_hidden)]
        self.b1 = [random.uniform(-1, 1) for _ in range(n_hidden)]
        self.w2 = [[random.uniform(-1, 1) for _ in range(n_hidden)] for _ in
        range(n_out)]
        self.b2 = [random.uniform(-1, 1) for _ in range(n_out)]

    def forward(self, x):
        h = [sigmoid(sum(w*xi for w, xi in zip(row, x)) + b)
             for row, b in zip(self.w1, self.b1)]
        o = [sigmoid(sum(w*hi for w, hi in zip(row, h)) + b)
             for row, b in zip(self.w2, self.b2)]
        return h, o

    def train_step(self, x, y):
        h, o = self.forward(x)

        o_err = [oi - yi for oi, yi in zip(o, y)]
        o_delta = [e * dsigmoid(oi) for e, oi in zip(o_err, o)]

        h_err = [sum(o_delta[k] * self.w2[k][j] for k in range(len(o_delta)))
                  for j in range(len(h))]
        h_delta = [e * dsigmoid(hi) for e, hi in zip(h_err, h)]

        for k in range(len(self.w2)):
            for j in range(len(self.w2[k])):
                self.w2[k][j] -= self.lr * o_delta[k] * h[j]
            self.b2[k] -= self.lr * o_delta[k]

        for j in range(len(self.w1)):
            for i in range(len(self.w1[j])):
                self.w1[j][i] -= self.lr * h_delta[j] * x[i]
            self.b1[j] -= self.lr * h_delta[j]
```

Two things are worth pointing at directly in this code, because they are the entire idea.

The line computing `o_delta` is the only place the network's actual error appears. It is the discrepancy between what came out and what should have come out, scaled by how sensitive the sigmoid was at that point.

The line computing `h_err` is backpropagation itself. Each hidden unit's share of the blame is a weighted sum of the output deltas, weighted by exactly the connection strengths, `self.w2`, that carried that hidden unit's activation forward to the output in the first place. A hidden unit that fed strongly into an output that was very wrong gets a large share of the blame. A hidden unit that barely influenced the output gets almost none. This is the chain rule, applied mechanically: the error flows backward along the same connections the activation flowed forward along, scaled at each step by how much influence that connection actually had.

Everything after those two lines is bookkeeping: use each delta to nudge the weights that

produced it, in the direction that would have reduced the error.

Watching It Solve the Problem Chapter 2 Couldn't

Train this exact network, two inputs, two hidden units, one output, on XOR.

```
Training a 2-2-1 network on XOR
  epoch 1: total squared error = 1.08602
  epoch 10: total squared error = 1.08085
  epoch 100: total squared error = 1.07547
  epoch 500: total squared error = 0.78151
  epoch 1000: total squared error = 0.04895
  epoch 2000: total squared error = 0.00660
  epoch 4000: total squared error = 0.00224

Final predictions:
  input=[0, 0]  target=0  raw_output=0.0210  rounded=0
  input=[0, 1]  target=1  raw_output=0.9776  rounded=1
  input=[1, 0]  target=1  raw_output=0.9775  rounded=1
  input=[1, 1]  target=0  raw_output=0.0280  rounded=0
```

All four cases correct, with the raw outputs sitting confidently close to 0 or 1 rather than hovering near the decision boundary. This is the exact function that made Chapter 2's single-layer perceptron oscillate forever between two and four correct answers, never settling, because no line could separate its classes. A network with one hidden layer of just two units settles on a correct answer within a few thousand passes over four examples.

It's worth pausing on the shape of that error curve rather than just the ending. For the first several hundred epochs, barely anything happens: the error sits close to where it started, drifting slightly. Then, between epoch 500 and epoch 1000, it collapses by more than an order of magnitude. This is not a smooth, steady descent, and it's not an artifact of this particular toy example either; it's a first, small glimpse of a pattern that recurs at every scale this book will discuss, up to and including the largest language models in later chapters. A network spends a long stretch of training apparently making little progress while its weights are actually rearranging themselves through a difficult region, and then, once they cross into a region where a good solution is reachable by mostly local, incremental improvement, error drops quickly. What the hidden layer is doing during that quiet stretch is worth naming directly: it is finding a useful internal representation, a way of recoding the two raw inputs into two new intermediate quantities that happen to be linearly separable, even though the original inputs were not. Chapter 2 proved no line separates XOR in its original two-dimensional input space. Nothing in that proof said anything about whether a line might separate it in some other, learned space. Backpropagation's real contribution isn't just fitting weights; it's discovering a representation the original problem didn't hand you.

What This Resolves, and What It Doesn't

The representational bottleneck from Chapter 2 is genuinely gone, in a strong sense. A network with even one hidden layer of sufficient width, given enough training and the right data,

can approximate essentially any function of practical interest; this is a real, provable result, not marketing language, and it's usually called the universal approximation theorem. That is worth sitting with for a moment, because it means the ceiling Minsky and Papert identified was specific to the single-layer architecture they were analyzing, not to neural networks as a category.

It does not follow that the problem is solved outright, and the field did not treat it as solved, even after this technique became widely known in the mid-1980s through the work of David Rumelhart, Geoffrey Hinton, and Ronald Williams, who popularized and clearly demonstrated the method that had sat in Werbos's 1974 thesis largely unnoticed. Universal approximation says a suitable network exists. It says nothing about how large that network needs to be, how much data it takes to find good weights for it, or how long training takes as networks and datasets grow past the toy scale of this chapter's four-example XOR problem. Those turn out to be separate, harder questions, and this book will spend several later chapters, on convolutional networks, sequence models, and the deep networks that took another two decades to become practical, working through exactly those questions: not whether a big enough network can represent what you need, but how to actually find the weights, at a scale where "try every combination" and even "run gradient descent for a few thousand epochs on four examples" both stop being options.

That isn't where the book goes next, though, and it's worth saying why. Backpropagation's arrival didn't make every other approach to learning obsolete overnight; for the better part of two decades, a different family of methods, built on probability rather than gradient descent, delivered results neural networks of the era couldn't match, often the quiet, working alternative in the very winters Chapter 3 described. Understanding what those methods offered, and precisely where they ran out of road, turns out to matter for understanding what deep learning eventually had to displace. Chapter 5 picks that thread up first.

Chapter 5

Statistical Learning

A Question the First Four Chapters Couldn't Ask

Every system built so far in this book has lived in a world of certainty, and none of them could do something this chapter's central example does easily: say "probably," and mean it precisely. `non_sequitur.mac` checked whether a conclusion follows with absolute rigor; there was no notion of "probably follows." The perceptron and the multilayer network learned decision boundaries, but the problems they were shown, AND, OR, XOR, have no ambiguous cases. Every input has exactly one correct output, and there is no such thing as a message that is a little bit spam.

Real classification problems are rarely that clean. Consider a spam filter. The word "free" shows up constantly in advertising junk, but it also shows up in perfectly ordinary email: "are you free for a meeting tomorrow." No fixed rule, "contains the word free implies spam," survives contact with real inboxes. What you actually want is something more like: given everything I've seen about how spam and legitimate email tend to differ, how likely is this specific message to be spam? That is a different kind of question than anything the first four chapters were built to answer, and it needed a different kind of tool.

Naive Bayes

One of the oldest and most durable answers is embarrassingly simple: count words, and use those counts as evidence. For every word in a message, ask how often that word tends to appear in spam versus in legitimate mail, based on past examples. Combine that evidence across every word in the message, and pick whichever class the combined evidence favors.

The "naive" part refers to a simplifying assumption doing a lot of the work: it treats every word's presence as independent of every other word's presence, given the class. That assumption is obviously false; words like "free" and "prize" tend to show up together, not independently. The method works well in practice anyway, often surprisingly well, and that gap between a flawed assumption and a useful result is itself a recurring theme in statistical learning that's worth noticing early.

```
class NaiveBayesSpamFilter:
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Laplace smoothing constant
        self.class_counts = defaultdict(int)
        self.word_counts = {"spam": defaultdict(int), "ham": defaultdict(int)}
        self.vocab = set()
        self.total_words = {"spam": 0, "ham": 0}

    def train(self, examples):
```

```
        for text, label in examples:
            self.class_counts[label] += 1
            for word in text.lower().split():
                self.word_counts[label][word] += 1
                self.total_words[label] += 1
                self.vocab.add(word)

    def word_prob(self, word, label):
        count = self.word_counts[label][word]
        return (count + self.alpha) / (self.total_words[label] + self.alpha *
        len(self.vocab))

    def classify(self, text):
        n_spam = self.class_counts["spam"]
        n_ham = self.class_counts["ham"]
        n_total = n_spam + n_ham

        log_prob = {
            "spam": math.log(n_spam / n_total),
            "ham": math.log(n_ham / n_total),
        }
        for word in text.lower().split():
            for label in ("spam", "ham"):
                log_prob[label] += math.log(self.word_prob(word, label))

        return log_prob
```

Trained on ten short, obviously toy examples (five spam, five ordinary email) and tested on four new messages it never saw during training:

```
"free meeting tomorrow"
  log P(spam)=-11.051  log P(ham)=-10.977  predicted=ham  margin=-0.073
"win a free prize now"
  log P(spam)=-16.033  log P(ham)=-20.820  predicted=spam  margin=+4.787
"send me the report please"
  log P(spam)=-20.638  log P(ham)=-18.335  predicted=ham  margin=-2.303
"claim your free money"
  log P(spam)=-12.960  log P(ham)=-16.101  predicted=spam  margin=+3.141
```

Look closely at the first result before the last three. ”Free meeting tomorrow” gets classified as ham, correctly, but the margin between the two log-probabilities is tiny, seventy-three thousandths, compared to margins of nearly five and over three for the other three messages. The word ”free” is genuinely ambiguous evidence in this training set; it appeared in spam examples and, thanks to ”are you free,” it would appear in ham examples too if the training set were larger. The model isn’t hiding that ambiguity behind a confident-looking label. It’s telling you, if you look at the margin rather than just the prediction, exactly how sure it is. Nothing in the first four chapters had a way to express ”probably, but not by much.” This does, and that is the actual advance this chapter is about, not the specific accuracy on four toy examples.

Why the Smoothing Term Matters

Look again at `word_prob`, and specifically at the `alpha` term added to both the numerator and denominator. Without it, any word that appeared zero times in a given class's training data would get an estimated probability of exactly zero for that class. Try removing it and classifying a message containing a word like "prize" that appeared only in the spam training examples:

```
ERROR: ValueError: math domain error
```

The program crashes, because it tries to take the logarithm of zero. This is not a bug in the implementation the way Chapter 1's sidebar described. It's the correct, if unhelpful, behavior of the underlying math: a probability of exactly zero for the ham class means the model has concluded, with total certainty, that no legitimate email could ever contain that word, based on nothing more than the fact that this particular ten-example training set happened not to include it. Laplace smoothing, the `alpha` term, is a small, principled adjustment that reserves a sliver of probability for words the model hasn't seen yet in a given class, on the reasoning that "I haven't seen this yet" and "this is impossible" are different claims and should never be treated as the same one. It's a small detail with a large consequence: without it, the model's confidence in what it doesn't know is indistinguishable from its confidence in what it does.

The Bottleneck This Era Runs Into

Naive Bayes, and the broader family of statistical methods that grew up alongside it through the 1970s, 80s, and 90s, decision trees, logistic regression, hidden Markov models, solved something the earlier chapters genuinely could not: how to make a good decision under uncertainty, quantify how confident that decision is, and update gracefully as more evidence arrives. This is not a small achievement. Some of the era's quietest, most durable successes, statistical machine translation, speech recognition systems built on hidden Markov models, spam filters much like this toy one, were running in production while the field's more visible symbolic and connectionist projects were living through the winters described in Chapter 3.

But look again at what this classifier actually operates on: word counts. Somebody, in this case the person writing the training examples, already decided that "which words appear" was the relevant signal, split the message into words, and handed the model a clean list of countable units. The model never touches raw text as a sequence of characters. It never has to figure out that "free" and "FREE" and "free!!!" probably mean the same thing, or that word order carries information the bag-of-words representation throws away entirely. All of that judgment, deciding what counts as a meaningful feature of the input, happened before the algorithm ever ran, and it happened by hand.

This is the feature-engineering bottleneck, and it is what defines the next several chapters as much as it defines this one. Support vector machines, taken up next, offer a mathematically elegant way to work with cleverer, implicitly defined feature spaces. Probabilistic graphical models, after that, offer a principled way to encode structural assumptions about how variables relate. Both are real advances. Neither one removes the underlying requirement that a human being decides, in advance, what the raw input should be turned into before any

learning begins. That requirement doesn't get resolved by a smarter algorithm within this paradigm. It gets resolved, a good deal later, by a different one entirely, once networks become deep enough and are given enough raw data to learn their own features directly. Keep the shape of this bottleneck in mind. It's the thread connecting a 1990s spam filter to why anyone bothered building deep convolutional networks at all.

Chapter 6

Support Vector Machines and Kernel Methods

Solving XOR Without a Hidden Layer

Chapter 4 solved XOR by adding a hidden layer and letting backpropagation discover, through training, an internal representation in which the problem became linearly separable. That was a real achievement, but it raises an obvious question. What if you didn't need to learn that representation at all? What if you could simply decide, in advance, on a fixed transformation of the input that makes the problem linearly separable, and then use nothing more sophisticated than the humble perceptron from Chapter 2 in that transformed space?

This turns out to work, and seeing it work is the cleanest way into this chapter's central idea.

A Kernel Is a Shortcut Through a Higher-Dimensional Space

Take the two-dimensional inputs XOR gives you, and imagine mapping each one into a higher-dimensional space using some fixed, nonlinear function, before running an ordinary linear classifier on the result. Done naively, this means picking a mapping, computing it explicitly for every input, and hoping the transformed points happen to be linearly separable. The kernel trick is the observation that, for a useful family of mappings, you never actually need to compute the transformed coordinates at all. You only need a function, called a kernel, that tells you the dot product two inputs would have had if you'd transformed them both and multiplied. Every algorithm that can be written using only dot products between examples, and the perceptron turns out to be one of them, can be rewritten to use a kernel function instead, and gets the benefit of an implicit, often very high-dimensional feature space for free.

The perceptron, rewritten this way, is usually called the dual-form or kernel perceptron. Instead of tracking a weight vector directly, it tracks how many times each training example was misclassified, and expresses its decisions purely in terms of kernel evaluations against the training set.

```
class KernelPerceptron:
    def __init__(self, kernel):
        self.kernel = kernel

    def train(self, X, y_labels, epochs=50):
        y = [1 if l == 1 else -1 for l in y_labels]
        n = len(X)
        self.alpha = [0] * n
        self.X = X
        self.y = y

        for epoch in range(epochs):
```

```
            mistakes = 0
            for i in range(n):
                s = sum(self.alpha[j] * self.y[j] * self.kernel(self.X[j],
                X[i])
                        for j in range(n))
                pred = 1 if s >= 0 else -1
                if pred != y[i]:
                    self.alpha[i] += 1
                    mistakes += 1
            if mistakes == 0:
                return epoch + 1
        return None

    def decision(self, x):
        return sum(self.alpha[j] * self.y[j] * self.kernel(self.X[j], x)
                   for j in range(len(self.X)))
```

Run this with an ordinary linear kernel, `kernel(a, b) = a · b`, on XOR, and it behaves exactly the way Chapter 2 predicts it must:

```
Kernel perceptron on XOR, linear kernel (should fail, same as Ch2)
  converged: None
  x=[0, 0] target=0 decision_value=+0.00 predicted=1
  x=[0, 1] target=1 decision_value=-1.00 predicted=0
  x=[1, 0] target=1 decision_value=-1.00 predicted=0
  x=[1, 1] target=0 decision_value=-2.00 predicted=0
```

No surprise there; the linear kernel gives back exactly the ordinary perceptron in a different notation, and Chapter 2 already proved this can't work. Now swap in a degree-2 polynomial kernel, `kernel(a, b) = (a · b + 1)^2`, changing nothing else about the algorithm:

```
Kernel perceptron on XOR, degree-2 polynomial kernel
  converged after: 10 epochs
  x=[0, 0] target=0 decision_value=-1.00 predicted=0
  x=[0, 1] target=1 decision_value=+2.00 predicted=1
  x=[1, 0] target=1 decision_value=+2.00 predicted=1
  x=[1, 1] target=0 decision_value=-3.00 predicted=0
```

All four correct, and converged, in the strict sense of the mistake count reaching zero, after ten passes. The polynomial kernel implicitly expands each two-dimensional input into a space that includes terms like the product of the two original coordinates, and in that expanded space, XOR's four points genuinely do fall on opposite sides of a hyperplane. Nothing about the perceptron's training rule changed. Only the notion of "similarity" between two examples changed, from an ordinary dot product to whatever the polynomial kernel computes, and that alone was enough to cross the representational ceiling Chapter 2 proved was uncrossable for the original, unmapped input space.

What a Real Support Vector Machine Adds

It's worth being precise about what this example has and hasn't shown. The kernel perceptron above finds *a* separating boundary in the implicit feature space, using the same mistake-driven update rule as Chapter 2's perceptron. It makes no attempt to find the *best* one. A support vector machine, properly speaking, adds a further, genuinely valuable idea on top of the kernel trick: among all the boundaries that correctly separate the training data, choose the one that maximizes the margin, the distance from the boundary to the nearest training example on either side.

That choice isn't cosmetic. A boundary that squeezes tightly past a couple of training points is more likely to misclassify a new example that lands close to where those points were, purely by chance of sampling. A boundary with a wide margin on both sides has more room for a new, unseen point to fall on the correct side even if it doesn't land exactly where a training example did. Finding the maximum-margin boundary is a constrained optimization problem, quadratic programming rather than a simple mistake-driven update loop, and it's genuinely more involved to implement than the perceptron shown here. But the kernel trick applies to it in exactly the same way: the optimization problem, like the perceptron's update rule, can be expressed entirely in terms of dot products between training examples, so the same swap, replace the dot product with a kernel function, gives you the same implicit access to high-dimensional, nonlinearly-separable feature spaces, now combined with a principled reason to prefer one separating boundary over another.

The Bottleneck This Chapter Doesn't Resolve

Chapter 5 closed by naming the feature-engineering bottleneck: statistical learning methods only ever operate on features a human decided to hand them, word counts, in that chapter's example. It's tempting to read the kernel trick as an escape from that bottleneck, since it seems to conjure an enormous, even infinite-dimensional feature space out of nowhere, without anyone explicitly engineering it.

Look again at what actually changed between the two runs above, though. Nothing about the training procedure changed. What changed was the choice of kernel function, degree-2 polynomial instead of linear, and that choice was made by a human, in advance, based on foreknowledge that a degree-2 expansion happens to be enough to make XOR linearly separable. For a real problem, nobody hands you that foreknowledge. Choosing a kernel, linear, polynomial of some degree, radial basis function, or one of the many other families in use, is itself a form of feature engineering, one level more abstract than choosing which words to count, but no less a human decision made before training begins and no less capable of being wrong for a given problem. The kernel trick is a genuinely elegant piece of mathematics, and support vector machines were, for a long stretch of the 1990s and 2000s, the strongest general-purpose classifiers available for many practical problems. But the bottleneck Chapter 5 identified is still standing at the end of this chapter. It moved up one level of abstraction, from "which words matter" to "which similarity function matters," and it would take a different kind of architecture entirely, one that learns its own feature representations directly from raw data rather than requiring any of them to be specified in advance, to actually resolve it. That is where Chapter 8 picks up, once Chapter 7 has covered one more major statistical-learning-era approach: encod-

ing what you already know about a problem's structure directly, using probabilistic graphical models.

Chapter 7

Probabilistic Graphical Models

What Naive Bayes Assumed Away

Chapter 5's spam filter made a strong simplifying assumption: every word's presence is independent of every other word's presence, given the class. That assumption is exactly what makes Naive Bayes unable to do something this chapter's model can: let one piece of evidence compete with another to explain the same outcome, rather than simply piling evidence up. The assumption bought a great deal of simplicity, and it worked well enough for the toy example, but it is also, transparently, false. Words are not independent. "Free" and "prize" show up together far more often than chance would predict, precisely because spam is written by people reusing the same small vocabulary of manipulative phrases. A model that cannot represent that dependency at all is missing something real about how language, and the world more generally, actually works.

Probabilistic graphical models exist to fix exactly this. Instead of assuming every variable is independent of every other variable given the class, a graphical model lets you specify, directly and explicitly, which variables actually influence which other ones, and leaves everything else independent by default. The graph is the specification. An edge from one variable to another means "this one directly influences that one." The absence of an edge is itself a claim: "these two don't influence each other directly, only through whatever they're both connected to."

A Small Network, Built by Hand

Here is one of the field's standard teaching examples, small enough to reason about completely. Three variables: whether it rained, whether the sprinkler ran, and whether the grass is wet. Rain influences whether the sprinkler needed to run (nobody turns the sprinkler on in a downpour), and both rain and the sprinkler directly influence whether the grass ends up wet.

```
P_RAIN = {True: 0.2, False: 0.8}

P_SPRINKLER_GIVEN_RAIN = {
    True:  {True: 0.01, False: 0.99},
    False: {True: 0.40, False: 0.60},
}

P_WETGRASS_GIVEN_SPRINKLER_RAIN = {
    (True,  True):  {True: 0.99, False: 0.01},
    (True,  False): {True: 0.90, False: 0.10},
    (False, True):  {True: 0.90, False: 0.10},
    (False, False): {True: 0.00, False: 1.00},
}
```

```
def joint_prob(rain, sprinkler, wetgrass):
    return (P_RAIN[rain]
            * P_SPRINKLER_GIVEN_RAIN[rain][sprinkler]
            * P_WETGRASS_GIVEN_SPRINKLER_RAIN[(sprinkler, rain)][wetgrass])
```

Every number here is a probability someone had to supply, based on domain knowledge, not learned from data. That's worth flagging immediately, and the chapter will come back to it. But notice what the structure already buys you before any inference happens: the joint probability of all three variables factors into exactly three small, manageable pieces, one per variable, each conditioned only on its direct causes in the graph, rather than one enormous table covering every combination of every variable at once. That factorization is the graphical model's central mechanical idea, and it's what makes exact inference tractable for networks with far more than three variables, as long as the graph stays reasonably sparse.

Querying the network means asking for the probability of some variable given whatever you've observed, summing out everything else:

```
def enumerate_query(query_var, evidence):
    variables = ["rain", "sprinkler", "wetgrass"]
    results = {}
    for query_val in (True, False):
        total = 0.0
        assignment = dict(evidence)
        assignment[query_var] = query_val
        free_vars = [v for v in variables if v not in assignment]
        for combo in product([True, False], repeat=len(free_vars)):
            full = dict(assignment)
            full.update(zip(free_vars, combo))
            total += joint_prob(full["rain"], full["sprinkler"],
            full["wetgrass"])
        results[query_val] = total
    z = results[True] + results[False]
    return {k: v / z for k, v in results.items()}
```

Explaining Away

Here is where the payoff shows up, and it's a genuinely different kind of reasoning than anything in Chapter 5. Start with no evidence at all. The prior probability of rain is simply 0.2, exactly what was specified going in.

Now observe that the grass is wet, and nothing else:

```
Query 1: P(Rain | WetGrass=True)
  P(Rain=True  | WetGrass=True) = 0.3849
  P(Rain=False | WetGrass=True) = 0.6151
```

Unsurprising so far: wet grass is evidence for rain, so the probability of rain nearly doubles from its 0.2 prior to 0.3849. Now add a second piece of evidence. Suppose you also learn that the sprinkler was running.

```
Query 2: P(Rain | WetGrass=True, Sprinkler=True)  -- 'explaining away'
  P(Rain=True  | WetGrass=True, Sprinkler=True) = 0.0068
  P(Rain=False | WetGrass=True, Sprinkler=True) = 0.9932
```

The probability of rain doesn't just fail to go up further. It collapses, down to well under a percent, lower even than the original 0.2 prior. Once the sprinkler alone is sufficient to explain the wet grass, the wet grass stops being much evidence for rain at all. This effect, called explaining away, is a real pattern in how evidence should combine, and it is exactly the kind of reasoning a flat, everything-is-independent model like Naive Bayes structurally cannot produce. Naive Bayes has no notion that two pieces of evidence might compete to explain a third; every word's contribution to the final probability is computed and added in complete isolation from every other word's contribution. The graph in this chapter's model isn't decoration. It's what makes this specific, correct inference pattern possible at all.

What Still Has to Be Supplied by Hand

Return to where `P_RAIN`, `P_SPRINKLER_GIVEN_RAIN`, and `P_WETGRASS_GIVEN_SPRINKLER_RAIN` came from. They were written into the code directly, by a person, based on prior knowledge about rain, sprinklers, and grass. So was the graph structure itself: the decision that rain influences the sprinkler and not the other way around, and that both influence the grass, rather than some other arrangement of edges. Nothing in this chapter's approach discovered that structure from data. A human looked at the problem, decided which variables mattered and how they related to each other, and encoded that understanding directly into the model, one conditional probability table at a time.

This should sound familiar. Chapter 1's expert systems asked a human to hand-write if-then rules. This chapter asks a human to hand-write a graph and a set of conditional probability tables instead. The representation is more expressive and more mathematically principled, capable of the explaining-away reasoning that flat rule sets and flat independence assumptions both miss. But the underlying dependency on a human supplying structured domain knowledge in advance never went away. It's still there, one more time, in a new and considerably more elegant notation.

It is possible to learn the numbers in a graphical model's conditional probability tables from data, given a fixed structure; that part of the problem is genuinely more tractable than it might look. Learning the graph structure itself from data, deciding which variables should have an edge between them at all, is a much harder problem, and one the field never fully solved within this paradigm at the scale needed for problems like raw images or free-form text. Both statistical learning's feature-engineering bottleneck from Chapter 5 and the structure-and-parameter bottleneck this chapter has just described point at the same underlying gap: every method covered since Chapter 1 depends, in some form, on a human deciding in advance what the relevant variables, features, or dependencies are. Chapter 8 is where that dependency finally breaks, not because anyone found a cleverer way to hand-specify structure, but because deep networks turned out not to need it specified at all.

Chapter 8

Deep Learning

What the Last Three Chapters Were Waiting On

Chapters 5 through 7 each ran into some version of the same wall. Naive Bayes worked from hand-counted words. Support vector machines worked from a hand-chosen kernel. Bayesian networks worked from a hand-drawn graph and hand-specified probability tables. In every case, a human decided, before training began, what the relevant features or structure of the problem were, and the learning algorithm only ever operated on that human decision, never on the raw material underneath it.

Chapter 4 already showed a way around this, in miniature, without anyone noticing at the time that it was the way around this. A hidden layer, trained with backpropagation, doesn't just fit a boundary. It builds its own internal representation, one the original input space didn't hand it, and it does this without a person specifying what that representation should look like. Nothing in principle stops you from stacking more hidden layers on top of each other, letting each one build a representation out of the layer before it: a first layer might learn something simple, a second layer might learn something built out of combinations of the first layer's output, and so on, each layer a little more abstract than the one beneath it. If that worked reliably, it would resolve the feature-engineering bottleneck outright. Nobody would need to decide in advance which words, which kernel, which graph structure mattered. The network would work that out, layer by layer, directly from raw data.

This is deep learning's core promise, and the idea itself is not new to this chapter. Backpropagation, from Chapter 4, already tells you how to compute the gradients needed to train a network with any number of layers. The genuinely strange part of this chapter is that the idea sat mostly unused for nearly two decades after backpropagation became widely known in the mid-1980s. Deep networks, more than two or three hidden layers, were tried, and they mostly didn't train well. The reason turns out to be a specific, measurable problem, not a vague sense that "deep networks are hard."

Watching Gradients Vanish

Build a network with ten hidden layers, all the same width, using the sigmoid activation function from Chapter 4, and run a single forward pass followed by a single backward pass, exactly the computation Chapter 4's training loop repeats thousands of times. Record the size of the weight gradient at every layer along the way, from the layer closest to the output back to the layer closest to the input.

```
def forward_backward(layers, x, activation, dactivation):
    activations = [x]
    a = x
```

```
    for W, b in layers:
        z = a @ W + b
        a = activation(z)
        activations.append(a)

    delta = activations[-1] - 0.0
    grad_norms = []

    for i in reversed(range(len(layers))):
        W, b = layers[i]
        a_out = activations[i+1]
        a_in = activations[i]
        delta = delta * dactivation(a_out)
        grad_W = a_in.reshape(-1, 1) @ delta.reshape(1, -1)
        grad_norms.append(np.linalg.norm(grad_W))
        delta = delta @ W.T

    grad_norms.reverse()
    return grad_norms
```

With identical random initialization, differing only in whether the activation function is sigmoid or ReLU (`relu(x) = max(0, x)`, a function this chapter hasn't used yet but is about to matter a great deal):

```
A 10-hidden-layer network, 20 units wide, identical random
initialization, differing only in activation function.

Gradient norm at each layer, layer 0 = closest to the input:

 layer        sigmoid           relu
     0     0.00129359   40410.079108
     1     0.00180970   36283.608235
     2     0.00352348   20390.571518
     3     0.00923468   12729.186098
     4     0.02438025   13018.023637
     5     0.05288814    9725.821551
     6     0.11537179    5613.132937
     7     0.20535379    8724.612728
     8     0.47538652    6617.408564
     9     1.09062402    7210.266542

Ratio of layer-0 gradient to layer-9 gradient (closest-to-input vs
closest-to-output):
  sigmoid: 0.0011861032
  relu:    5.6045194543
```

Look at the sigmoid column first. The gradient at layer 9, the layer nearest the output, is a reasonable 1.09. By layer 0, the layer nearest the input, ten layers of backpropagation later, it has shrunk to 0.0013, less than a thousandth of its starting size. The ratio at the bottom makes it exact: the input-side gradient is roughly one nine-hundredth the size of the output-side gradient. That's not a training run that happens to be going slowly. It's a training run

in which the earliest layers of the network are receiving a learning signal too small to meaningfully update their weights, no matter how long you wait. Add more layers and the effect compounds; each additional layer multiplies in another factor smaller than one, because the sigmoid function's derivative is never larger than 0.25 anywhere and is close to zero across most of its range. Stack enough layers and the gradient reaching the earliest ones underflows into a value indistinguishable from noise.

Now look at the ReLU column. The numbers are much larger overall, because ReLU doesn't squash its input into a narrow range the way sigmoid does, but more importantly, the ratio between layer 0 and layer 9 doesn't collapse toward zero. If anything it grows slightly, but it stays within the same rough order of magnitude across all ten layers rather than shrinking by three orders of magnitude. The gradient signal reaches the earliest layers largely intact.

Why This Explains a Twenty-Year Gap

This single property, whether a network's activation function preserves or destroys gradient magnitude across many layers, goes a long way toward explaining why deep networks were mostly considered impractical from the mid-1980s until roughly the mid-2000s to early 2010s, despite backpropagation itself having been available and well understood that entire time. Researchers weren't failing to try deep networks. They were training them with sigmoid and tanh activations, the standard choice inherited from Chapter 4, and running directly into the vanishing gradient problem this section just measured directly. The earliest layers, the ones responsible for learning the simplest, most foundational features everything else would be built on top of, were the layers receiving the weakest training signal, which is close to the worst possible arrangement for a system that's supposed to learn hierarchically.

The eventual fixes arrived in stages, not all at once. Geoffrey Hinton and colleagues showed in 2006 that training a deep network one layer at a time, in an unsupervised pretraining phase, before fine-tuning the whole stack together, could get around the worst of the vanishing gradient problem well enough to train networks meaningfully deeper than had been practical before. This was an important proof that depth could work at all, though it was a workaround rather than a resolution: it sidestepped the vanishing gradient rather than removing its cause. The more direct fix came from simply changing the activation function. Rectified linear units, ReLU, output exactly their input when positive and exactly zero when negative, and unlike sigmoid, their derivative is either exactly 1 or exactly 0, never something small and shrinking. Combined with better-designed random weight initialization schemes tuned specifically to keep signal variance roughly constant from layer to layer, ReLU largely dissolved the vanishing gradient problem for networks of the depth researchers actually wanted to train, as the comparison above shows directly.

What This Actually Unlocks

It's worth being precise about what got resolved here and what didn't. This chapter has not yet shown a network learning features from raw data the way the opening section promised; that demonstration, edge detectors and shape detectors emerging on their own from raw pixels, belongs to the next chapter, where the specific architecture built for that purpose, the convo-

lutional network, finally makes it concrete. What this chapter has shown is narrower and more foundational: the specific mechanical obstacle that made ”just add more layers” impractical for two decades, made visible and measured directly, and the specific, comparatively small change, a different activation function plus more careful initialization, that removed it.

That narrowness is worth sitting with. The representational bottleneck from Chapter 2 was resolved by an architectural idea, adding a hidden layer. The feature-engineering bottleneck from Chapters 5 through 7 is resolved, in principle, by the same idea taken further: stack many hidden layers and let each build on the last. But between having the right idea and having it actually work sat a purely practical, computational obstacle that had nothing to do with whether the idea was correct. Backpropagation was correct in 1986. Depth was the right direction in 1986. Neither fact made deep networks trainable until the activation functions and initialization schemes caught up, decades later. That specific gap, between a correct idea and a trainable one, is this chapter’s claim, and it’s a mechanical, historical fact about depth in particular: what changed between 1986 and 2012 was optimization catching up to representation, nothing broader. Whether that same shape of gap, a correct idea outrunning the practical means to realize it, turns out to be a recurring feature of this whole field rather than a one-time story about depth, is a question worth holding until the very last chapter, once there’s a full history’s worth of evidence to check it against.

Chapter 9

Convolutional Networks

The Demonstration Chapter 8 Deferred

Chapter 8 promised that depth, once made trainable, would let a network learn its own features directly from raw data, finally resolving the feature-engineering bottleneck that Chapters 5 through 7 kept running into. It didn't show that happening. It showed the narrower, prerequisite fact that gradients could now survive passage through many layers. This chapter delivers on the promise directly, with an image classification task small enough to build and inspect by hand.

What a Convolution Actually Computes

Before training anything, look at what a human-designed feature detector looks like, so there's something concrete to compare a learned one against. A small 3x3 kernel, slid across every position of an image, multiplying and summing the pixels underneath it at each position, is a convolution. Here is one specifically designed by hand to respond to vertical edges: positive weights on the left column, negative on the right, zero in the middle.

```
vertical_edge_kernel = np.array([
    [1, 0, -1],
    [1, 0, -1],
    [1, 0, -1],
])
```

Applied to a small image containing a vertical bar, and separately to one containing a horizontal bar:

```
Hand-designed vertical-edge kernel on a vertical-bar image:
[[-3. -3.]
 [-3. -3.]]

Same kernel on a horizontal-bar image:
[[0. 0.]
 [0. 0.]]
```

Strong, uniform response to the vertical bar; almost nothing on the horizontal one. This kernel does exactly one job, detect a left-to-right intensity change, and it does that job at every position in the image using the same nine numbers. That reuse, the same small set of weights applied at every spatial position rather than a separate weight for every pixel, is called weight sharing, and it's the architectural idea this whole chapter is built around. Notice that this is a different kind of move than Chapter 6's kernel trick. A kernel changes where in the problem you look, remapping the input into a space where an existing linear method suddenly works. Weight

sharing changes what the architecture is allowed to assume about the problem before it has seen a single example: that a pattern's identity doesn't depend on where in the image it sits. That assumption, built into the architecture itself rather than chosen per-problem, is called an inductive bias, and it's worth noting immediately why it matters beyond convenience: a fully connected layer looking at even this tiny image would need a separate weight for every pixel, learned independently, with no way to transfer what it learned about detecting a vertical edge in one part of the image to recognizing the same pattern somewhere else. A convolutional filter learns the pattern once and applies that same knowledge everywhere, because the architecture was built to assume it should.

Learning the Filter Instead of Designing It

The kernel above was designed by hand. Now build a network with a single 3x3 filter whose weights start as random noise, and train it, using nothing but labeled examples, to distinguish images containing a vertical bar from images containing a horizontal bar. No one tells it what an edge is.

```
def forward(img, kernel, bias):
    conv_out = conv2d(img, kernel)
    pooled = np.mean(conv_out**2)      # energy pooling
    pred = sigmoid(pooled + bias)
    return conv_out, pooled, pred
```

The pooling step here deserves a note, because the obvious first choice doesn't work, and the reason it doesn't work is itself informative. Simply averaging the raw convolution output collapses back into a plain linear function of the pixels, no better at separating the two classes than Chapter 2's original perceptron, because every image in this dataset contains exactly the same number of lit pixels regardless of orientation, so a purely linear summary can't tell them apart. Squaring the convolution output before averaging, energy pooling, fixes this: it responds strongly whenever the filter finds a strong match anywhere in the image, in either direction, and that turns out to be enough signal to learn from.

```
Training a single learned 3x3 conv filter, energy pooling, lr=0.05
  epoch    1: loss=0.6834  train_acc=0.60
  epoch   10: loss=0.1152  train_acc=1.00
  epoch   50: loss=0.0130  train_acc=1.00
  epoch  100: loss=0.0057  train_acc=1.00
  epoch  200: loss=0.0026  train_acc=1.00
  epoch  300: loss=0.0017  train_acc=1.00
```

Perfect training accuracy by epoch ten, loss still falling steadily afterward. Held out on 30 entirely new images the network never trained on:

```
Held-out test accuracy on 30 new images: 30/30
```

Looking Inside the Learned Filter

The result worth lingering on isn't the accuracy number. It's what the filter's weights look like after training, compared to the hand-designed one from the start of the chapter.

```
Learned 3x3 filter:
[[ 1.963 -2.359  0.095]
 [ 2.154 -2.55   0.19 ]
 [ 2.042 -2.483  0.189]]

column sums: [ 6.159 -7.392  0.474]
row sums:    [-0.301 -0.206 -0.252]
```

Read down each column: the three values in the left column are all close to 2, the three in the middle column are all close to -2.4, the three on the right are all close to 0. In other words, the filter is nearly uniform down each column and sharply different across columns, which is exactly the structural signature of the hand-designed vertical-edge kernel at the top of this chapter, positive on one side, near-zero or negative on the other, constant from top to bottom. The row sums confirm it from the other direction: they're all close to zero and close to each other, meaning the filter genuinely doesn't care which row it's looking at, only which column. Nobody supplied that structure. It emerged from a training loop that saw nothing but example images and a right-or-wrong signal, exactly the way Chapter 4's hidden layer discovered a useful representation for XOR without anyone specifying it, now happening with actual visual structure instead of an abstract logic table.

From a Toy Filter to a Real Vision System

Everything in this chapter has used a single filter on tiny six-by-six images. A real convolutional network stacks many filters per layer, typically dozens to hundreds, each free to specialize in a different pattern, edges at different orientations, then in later layers, combinations of edges into corners and textures, then combinations of those into object parts. This is the hierarchical composition Chapter 8 promised depth would provide, now made concrete: not one hand-designed feature detector applied once, but many learned ones, stacked, each layer building on the vocabulary the layer beneath it discovered.

The historical proof that this works at real scale came in 2012, when a network called AlexNet, designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, entered the ImageNet competition, a benchmark of over a million labeled photographs across a thousand categories. Every strong entry in the years before had relied on hand-engineered visual features, descriptors like SIFT and HOG, built by researchers who had spent years figuring out by hand what made a useful visual feature, feeding those hand-built features into classifiers like the support vector machines of Chapter 6. AlexNet used none of that. Its convolutional filters, five layers of them, were entirely learned from raw pixels, using the ReLU activations and other trainability fixes described in Chapter 8. It won by a startling margin: a top-5 error rate of 15.3 percent, against 26.2 percent for the best entry built on hand-engineered features, in a single year's competition. The feature-engineering bottleneck this book has been tracking since Chapter 5 didn't erode gradually. In computer vision specifically, it broke essentially all at once, in public, in a

single benchmark result that the rest of the field could not ignore.

What's Still Missing

Convolution's weight sharing solves a spatial problem: the same pattern, a vertical edge, an eye, a wheel, can appear anywhere in an image, and a convolutional filter, once it learns to recognize that pattern at all, recognizes it everywhere, without needing separate training for every possible position. But images have no inherent order to their content the way a sentence or a sound recording does. Nothing about a convolutional filter is built to handle the specific kind of structure where the meaning of the current input depends on a sequence of things that came before it, where "before" and "after" matter in a way that "left" and "right" in an image generally don't. That is a different kind of structure, and it needs a different kind of architecture. Chapter 10 takes it up directly.

Chapter 10

Sequence Models and Recurrent Networks

A Different Kind of Structure

Chapter 8 diagnosed vanishing gradients as a problem of depth: gradients shrinking as they pass back through many stacked layers. This chapter asks whether that diagnosis was too narrow, whether "depth" was ever really the cause, or just the setting where the problem happened to first get measured. Convolution solved a specific problem: the same visual pattern can appear anywhere in an image, and a network shouldn't have to relearn it separately at every position. But images don't have a before and after. Shift a photograph's contents to the left or right and it's still fundamentally the same photograph. Sentences don't work this way. "The dog bit the man" and "the man bit the dog" contain identical words and an identical bag-of-words representation, and they mean opposite things, because order carries meaning that convolution's spatial reuse was never built to capture. Speech, sensor readings, and stock prices share the same property: what a value means depends on what came before it, sometimes a long way before it.

Recurrent networks are the architectural answer. Instead of processing an entire input at once, a recurrent network processes a sequence one element at a time, maintaining a hidden state that gets updated at every step and carried forward to the next one. In principle, that hidden state can accumulate information from arbitrarily far back in the sequence, exactly the kind of long-range dependency neither the feedforward networks of Chapter 4 nor the convolutional networks of Chapter 9 were built to represent, since both require a fixed-size input decided in advance.

The Simplest Recurrent Network

Here is the core of a vanilla recurrent network, sometimes called an Elman network after its 1990 origin. At each timestep, it combines the current input with its own previous hidden state to produce a new hidden state, using the same weights at every step, the same reuse principle that made convolution parameter-efficient, now applied across time instead of across space.

```
def forward(self, seq):
    h = np.zeros(self.h_size)
    hs = [h]
    for x in seq:
        h = tanh(x * self.Wxh + h @ self.Whh + self.bh)
        hs.append(h)
    y = sigmoid(np.dot(h, self.Why) + self.by)
    return hs, y
```

To see what this can and can't do, train it on a deliberately simple task: given a sequence

of random bits, output the very first bit in the sequence, ignoring everything after it. This sounds trivial, but it requires the network's hidden state to carry a single bit of information across every intervening timestep, untouched by whatever noise arrives in between, all the way to the end of the sequence. It's the standard toy problem for testing whether a recurrent network can actually maintain a long-range dependency, rather than just reacting to whatever it saw most recently.

What Works, and Where It Stops Working

Trained on sequences of length six, the network learns the task almost immediately and perfectly:

```
Training a vanilla RNN to recall the FIRST bit of a length-6 sequence
  epoch   1: avg loss = 0.0090
  epoch   5: avg loss = 0.0000
  ...

Accuracy at trained length (6) and generalizing to longer sequences:
  length  6: accuracy = 1.000
  length 10: accuracy = 1.000
  length 15: accuracy = 0.865
  length 20: accuracy = 0.890
  length 30: accuracy = 0.430
```

Perfect accuracy at the length it was trained on, and it holds up reasonably well even a bit beyond that. But push the sequence length out further and accuracy doesn't just degrade, it eventually falls below the fifty percent a coin flip would get, meaning the network is actively wrong more often than chance on long sequences, not merely uncertain.

The more telling result comes from training a fresh network directly on length-twenty sequences, rather than asking a length-six network to generalize:

```
--- Now train directly on length-20 sequences from scratch ---
  epoch   1: avg loss = 0.2115
  epoch   5: avg loss = 0.2664
  epoch  10: avg loss = 0.4315
  epoch  20: avg loss = 0.4315
  epoch  40: avg loss = 0.4315
  length 20 accuracy after direct training: 0.540
```

The loss doesn't decrease with training. It gets worse and then plateaus, and final accuracy sits barely above chance. This isn't a generalization failure, the network never having seen sequences this long before. It's an optimization failure: even given every opportunity to learn the length-20 version of the task directly, from scratch, gradient descent simply cannot find weights that solve it.

The Same Wall, in a New Direction

The explanation is Chapter 8's vanishing gradient problem, reappearing in a new guise. Unroll a recurrent network across a sequence of length twenty and it is, mathematically, indistinguishable from a twenty-layer feedforward network, with the same weight matrix reused at every layer. Backpropagating an error through that unrolled sequence, called backpropagation through time, means passing the gradient back through twenty repeated multiplications by the same recurrent weight matrix and the same tanh derivative, the exact mechanism Chapter 8 measured directly. Compute the gradient magnitude reaching each timestep of the length-20-trained network above, earliest timestep first:

```
Gradient magnitude reaching each timestep, single length-20 sequence,
using the length-20-trained weights (t=0 is the earliest, most-needed
timestep):
  t= 0  grad_norm=0.0000000000
  t= 1  grad_norm=0.0000000000
  t= 2  grad_norm=0.0000000000
  t= 4  grad_norm=0.0000000000
  t= 8  grad_norm=0.0000000000
  t=12  grad_norm=0.0000000000
  t=16  grad_norm=0.0000000000
  t=18  grad_norm=0.0000000441
  t=19  grad_norm=0.0000633719
```

By timestep sixteen the gradient has already vanished to a value too small to display at ten decimal places. The information the network needs, the value of the very first bit, sits at timestep zero, precisely where the training signal is weakest by many orders of magnitude. The network isn't failing to find a solution because the task is conceptually hard. It's failing because the gradient telling it how to improve never reliably reaches the part of the network that would need to change.

Gating: Letting Information Skip the Multiplication

Chapter 8's fix for vanishing gradients across depth was to change the activation function, replacing sigmoid's steep, narrow-range squashing with ReLU's flat, unbounded response, so that backpropagated gradients could pass through many layers without shrinking at every step. The equivalent fix for vanishing gradients across time, introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997 as the Long Short-Term Memory architecture, and simplified further in later variants like the Gated Recurrent Unit, follows the same underlying logic, applied to recurrence instead of depth.

An LSTM adds a second, separate pathway alongside the ordinary hidden state, called the cell state, and controls what enters and leaves it using learned gates, small networks of their own that output values between zero and one, indicating how much of something to let through. Critically, the cell state is updated primarily by addition, gated addition of new information, gated removal of old information, rather than by repeated multiplication through a squashing nonlinearity at every single timestep. Addition doesn't shrink a gradient passing back through it the way repeated multiplication by values less than one does. A gate can learn to

hold a value in the cell state essentially unchanged across dozens or hundreds of timesteps, simply by learning to keep its forget gate close to one and its input gate close to zero once the relevant information has been stored, something the vanilla recurrent network's forward has no mechanism to do at all; every timestep necessarily overwrites the hidden state through the same tanh nonlinearity, whether that's useful or not.

What Recurrence Still Doesn't Solve

LSTMs and their relatives genuinely extended how far back a network could reliably reach, from the handful of timesteps the vanilla RNN above managed to hundreds. They were, for roughly two decades, the standard architecture for language, speech, and any other genuinely sequential data. But notice something about both the vanilla RNN and the gated architectures that improved on it: every one of them processes a sequence strictly one step at a time. Timestep ten cannot be computed until timestep nine's hidden state exists, which cannot be computed until timestep eight's does, all the way back to the start. This is inherent to the recurrent formulation itself, not a limitation particular to any one architecture's gradient behavior, and it creates a bottleneck of its own: training is forced to proceed sequentially through a sequence, one step after another, in a field increasingly defined by how much parallel computation could be thrown at a problem at once. Chapter 11 takes up an architecture that keeps recurrence's sensitivity to order and long-range dependency, while giving up the one-step-at-a-time processing that both the vanilla RNN and the LSTM still require.

Chapter 11

Attention and Transformers

What Chapter 10 Left Unresolved

LSTMs extended how far a recurrent network could reach back into its own history, but Chapter 10 closed on a limitation that no amount of better gating fixes: recurrence is inherently sequential. Timestep ten's hidden state cannot exist until timestep nine's does. Training has to walk through a sequence one step at a time, and so, in the original architecture, does inference. In a field increasingly able to throw enormous amounts of parallel computation at a problem, an architecture that structurally forbids parallelizing across the sequence dimension is a bottleneck of its own, separate from and in addition to the vanishing gradient problem Chapter 10 measured directly.

Attention is built to remove exactly that constraint. Instead of information having to pass through every intervening timestep's hidden state to travel from an early position to a late one, attention lets any output look directly at any input position, in a single step, with a learned weighting over how much to attend to each one. There is no chain of hidden states to pass a gradient back through. The path length between any two positions in the sequence is one, regardless of how far apart they are.

A Minimal Attention Mechanism

Here is attention stripped to its simplest working form: one learned query vector, shared across the whole sequence, that decides how much to attend to each position based on that position's content.

```
def forward(self, seq):
    X = np.array([self.embed[int(b)] for b in seq])
    K = X @ self.Wk
    V = X @ self.Wv
    scores = (K @ self.query) / np.sqrt(self.d)
    weights = softmax(scores)
    context = weights @ V
    y = sigmoid(np.dot(context, self.Wout) + self.bout)
    return X, K, V, scores, weights, context, y
```

Every position gets turned into a key vector and a value vector. The query compares itself against every key at once, `scores`, producing a single number per position; softmax turns those scores into a proper probability distribution over positions, `weights`; and the final prediction is built from a weighted sum of the value vectors, weighted exactly by how much attention each position received. Nothing here processes the sequence step by step. Every position's key and value can be computed independently and in parallel, and the query attends to all of

them simultaneously.

Train this directly on the length-20 version of Chapter 10's task, recall the first bit of the sequence, the exact task the vanilla RNN failed to learn even with direct training at that length:

```
epoch   1: avg loss = 0.2663
epoch   5: avg loss = 0.2609
epoch  10: avg loss = 0.2609
...
Attention weights on one example sequence:
  weights:  [0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05
  0.05
 0.05 0.05 0.05 0.05 0.05 0.05]
```

It fails completely. Every position gets identical weight, one twentieth, and accuracy sits at chance. Worth pausing on this failure before fixing it, because it's diagnostic rather than incidental. The model above has no way to know which position in the sequence a given key vector came from. It only sees the value at each position, zero or one, embedded into a vector. Since "recall the first bit" is a task specifically about position, "the one at index zero," not about value, "a zero appears somewhere," a mechanism that only ever looks at content and never at position cannot represent the task at all, no matter how long it trains. Attention, as built so far, treats a sequence as an unordered collection of items. That's a real and important limitation, not a bug in this implementation, and it's the reason every practical attention-based architecture adds positional information back in explicitly.

Adding Position Back In

The fix is small: give each position in the sequence its own learned vector, and add it to that position's content embedding before anything else happens.

```
X = np.array([self.embed[int(b)] + self.pos_embed[i] for i, b in
enumerate(seq)])
```

Everything else about the model is unchanged. Retrained on the identical length-20 task:

```
epoch   1: avg loss = 0.0959
epoch   5: avg loss = 0.0000
...
Accuracy at trained length (20) and other lengths within the positional table:
  length   6: accuracy = 1.000
  length  10: accuracy = 1.000
  length  20: accuracy = 1.000
  length  30: accuracy = 1.000

Attention weights on one example sequence:
  sequence: [0 0 1 0 1 1 0 0 1 1 0 1 0 1 0 1 0 0 0 0]
  weights:  [0.976 0.    0.    0.001 0.003 0.002 0.001 0.001 0.002 0.    0.001
  0.
 0.001 0.001 0.001 0.002 0.001 0.003 0.002 0.   ]
```

Loss collapses to zero within five epochs. Accuracy is perfect, at the trained length and at every other length tested. And the attention weights are directly readable: 0.976 on position zero, effectively nothing everywhere else. The model isn't just getting the right answer. It's showing you, in a way none of this book's earlier architectures could, exactly where it's looking to get it. Compare this to Chapter 10's best result on the same task, 54 percent accuracy, barely above chance, after the same number of training epochs. Nothing about the underlying task changed between the two chapters. What changed is that information no longer has to survive twenty sequential multiplications to reach the output; it travels in a single attention-weighted step, and the gradient flowing back during training makes that same single step in reverse.

From One Query to a Full Transformer

The mechanism above uses a single, fixed query shared across the whole sequence, useful for building intuition but far short of what real language and sequence modeling need. A full transformer, the architecture underlying essentially every large language model discussed later in this book, generalizes this idea in two ways. First, every position in the sequence gets its own query, not just one global one, so that every position can attend to every other position, and the result is a new representation for every position, not just a single summary output. This is called self-attention, since the sequence is attending to itself. Second, this whole operation is run several times in parallel with different learned projections, called multiple attention heads, so that different heads can specialize in attending based on different kinds of relationships, some tracking nearby words, others tracking a distant subject a verb needs to agree with, without any of that specialization being hand-designed. Stack several layers of self-attention, each followed by an ordinary feedforward network like Chapter 4's, and you have the core of a transformer.

The Bottleneck This Solves, and the One It Creates

Attention resolves Chapter 10's sequential processing bottleneck directly: every position can be processed in parallel during training, and information can travel between any two positions in a single step rather than a chain of them, which is why the demonstration above trained successfully on a task the recurrent network from the previous chapter could not learn at all at the same length. The lesson worth carrying forward is narrower than "attention fixes vanishing gradients," and more useful: what determines whether a gradient survives was never depth as such, it was path length, the number of steps a signal has to travel between where it's produced and where it's used. Stacked layers and unrolled timesteps both happened to make that path long. Attention is what happens when an architecture is built to keep that path short on purpose.

It does not do this for free. Look again at what full self-attention requires: every position computing an attention score against every other position. For a sequence of length n, that's proportional to n-squared comparisons, growing quadratically as sequences get longer. The single-query toy model above sidesteps this, one query attending over the sequence is only proportional to n, but a real transformer's full self-attention, every position attending to every other position, does not. This is a new kind of computational bottleneck, different in shape from the exhaustive symbolic search of Chapter 1 or the vanishing gradients of Chapters 8 and

10, but a genuine constraint on how long a sequence a transformer can practically process, and it becomes one of the central practical concerns as these architectures are scaled up, taken up directly in Chapter 12.

Chapter 12

Scaling Laws

A Different Kind of Question

Every chapter so far has asked some version of "what architecture solves this problem." This chapter asks a more practical and, for a long time, a more expensive question to answer: given an architecture that works, how much better does it get if you make it bigger, and where exactly should "bigger" go? More parameters? More training data? More computation spent training the same size model longer? Before 2020, the honest answer was largely "run the experiment and see." Training a large model was expensive enough that guessing wrong about where to invest that expense was a serious cost, not an academic inconvenience.

What Kaplan and Colleagues Found

In 2020, Jared Kaplan and colleagues at OpenAI published an empirical study asking exactly this question systematically: train many transformer language models of varying size, on varying amounts of data, for varying amounts of compute, and see how the test loss behaves. The finding that gave the paper its name was that, within a wide range and as long as the model wasn't starved of data or the dataset wasn't starved of a large enough model to learn from it, loss decreased smoothly as a power law in each of these quantities individually: model size, dataset size, and total training compute. Not roughly. Not as a general trend with a lot of noise around it. A power-law curve, holding across several orders of magnitude of scale, predictable enough to extrapolate from a handful of smaller training runs to how a much larger one should perform before ever running it.

That predictability is the practical payoff. It converts "will a bigger model actually help, and by how much" from a question you answer by spending millions of dollars and waiting, into a question you can answer by fitting a curve to a handful of cheaper runs and extrapolating.

Watching the Same Pattern at Toy Scale

The real scaling law papers trained hundreds of models at costs far beyond anything reproducible in this book. But the specific caveat embedded in Kaplan's finding, that loss scales predictably with one quantity only as long as you aren't bottlenecked by the others, is small enough to demonstrate directly, and seeing it firsthand makes the caveat concrete rather than abstract.

Train a small feedforward network to approximate a fixed, moderately complex function, holding the training set at a fixed 200 points and varying only the network's width:

```
Test loss vs. model size (hidden units), fixed dataset (200 points)
```

```
  hidden   params    test_loss
       2        7     0.470332
       4       13     0.113131
       8       25     0.162638
      16       49     0.130641
      32       97     0.129404
      64      193     0.123707
     128      385     0.106531
     256      769     0.125510
     512     1537     0.109069
```

Loss drops sharply from 2 to 32 hidden units, and then it essentially stops improving. Going from 97 parameters to 1,537 parameters, a fifteen-fold increase, buys almost nothing. Now hold the model size fixed at 32 hidden units and instead vary the amount of training data:

```
Fixed model size (32 hidden units, 97 params), varying dataset size:
 n_train    test_loss
      10     0.748627
      20     0.260053
      50     0.149105
     100     0.115207
     200     0.129404
     400     0.137154
     800     0.146811
```

The same pattern, mirrored. Loss improves sharply as data grows from 10 to 100 examples, then flattens and even ticks slightly upward as more data is added beyond that. A 97-parameter network simply doesn't have the capacity to make use of 800 training examples any better than it makes use of 100.

Put the two results side by side and the picture is exactly what Kaplan's caveat describes, at a scale small enough to see all at once. Around 32 hidden units and roughly 100 to 200 training examples, this particular task crosses from being parameter-limited to being data-limited. Below that crossover, adding parameters helps and adding data helps. Above it, each factor alone runs into a ceiling set by the other. A model too small to use additional data will not improve with more of it. A dataset too small to exercise a model's full capacity will not push it any further no matter how large the model gets. This is a toy-scale echo of a genuinely large-scale finding, not a coincidence manufactured to fit the chapter; it's the same underlying interaction between capacity and data showing up wherever you look for it.

Chinchilla: Getting the Balance Wrong at Enormous Scale

This caveat turned out to matter enormously in practice. Kaplan's original paper suggested that, for a fixed compute budget, most of the benefit came from scaling model size aggressively, with data requirements growing more slowly. Several of the largest language models trained in the years after, following that guidance, ended up very large relative to how much data they were trained on.

In 2022, a DeepMind team led by Jordan Hoffmann revisited the question with a larger and

more systematic sweep, training over 400 models ranging from 70 million to 16 billion parameters on data ranging from 5 billion to 500 billion tokens. Their conclusion revised Kaplan's guidance directly: for a fixed compute budget, model size and training data should grow at roughly the same rate, not model size dominating. Many existing large models, trained under the older guidance, were, in their word, significantly undertrained: too large for the amount of data they'd seen, sitting well above this chapter's toy demonstration's own crossover point, in the parameter-limited regime relative to the actual computational effort spent. As direct evidence, they trained a new model, Chinchilla, at 70 billion parameters on 1.4 trillion tokens, a far smaller model than DeepMind's existing 280-billion-parameter Gopher, trained on only 300 billion tokens, using comparable total compute, and Chinchilla outperformed Gopher across a wide range of evaluations.

The Bottleneck This Chapter Resolves

The bottleneck this chapter addresses isn't architectural, the way a hidden layer or an attention mechanism was. It's a planning bottleneck: before scaling laws were characterized, there was no reliable way to know in advance whether a much larger training run would be worth its cost, or how to divide a fixed budget between a bigger model and more data, short of running the expensive experiment directly. Afterward, that allocation became something you could extrapolate from small, cheap runs using a predictable curve, and, after Chinchilla, extrapolate correctly, in the sense of not systematically over-investing in parameters at the expense of the data needed to actually use them.

It's worth being precise about what scaling laws don't tell you, since the next several chapters depend on the distinction. A power-law curve in loss tells you how well a model will predict its training distribution as it grows. It says nothing directly about what specific capabilities emerge along the way, what a model trained at a given scale can actually be used for, or how raw next-token prediction ability turns into something that behaves usefully as an assistant, a coding partner, or a research tool. Those questions, what these enormous, predictably-scaled models actually are and how they get built in practice, are where Chapter 13 picks up.

Chapter 13

Large Language Models

What "Large Language Model" Actually Names

By this point in the book, every piece needed to define a large language model has already been built separately. Chapter 11 gave the architecture: self-attention, letting every position in a sequence draw information from every other position in a single step, stacked into a transformer. Chapter 12 gave the scaling recipe: train that architecture at whatever size a fixed compute budget allows, balanced against a correspondingly large amount of data, and expect its loss to fall in a predictable way. A large language model is what you get when you take a transformer, set it up to predict the next token in a sequence given everything before it, and apply Chapter 12's scaling recipe about as far as anyone has been willing to fund. There's no separate secret architecture behind the name. The name mostly describes an amount, applied to ideas already on the table by Chapter 11.

That simplicity is worth sitting with for a moment, because it raises an immediate question the rest of this chapter has to answer. If the recipe is really just "attention, scaled up, predicting the next token," why does scale, on its own, produce something that can hold a conversation, follow instructions, or write working code, when a small version of the same recipe plainly can't? Part of the answer is Chapter 12's territory, better loss means better next-token prediction, and better next-token prediction over a large and varied enough training set means implicitly modeling an enormous amount of structure about language, facts, and reasoning patterns embedded in that text. But there's a second, more specific reason worth making concrete, and it goes back to a very old problem this book has already met once, in Chapter 1.

The Simplest Possible Language Model, and Why It Can't Scale

Before there were transformers, the standard way to build a language model was to count. Given a fixed-length context, some number of preceding characters or words, tabulate, from a training corpus, how often each possible next character or word followed that exact context. To generate text, look up the current context in that table and sample from whatever followed it historically. This is an n-gram model, and it is worth building one directly, because its specific failure mode is the clearest possible motivation for everything transformers do differently.

```
def train_ngram(text, n):
    model = defaultdict(Counter)
    for i in range(len(text) - n + 1):
        context = text[i:i+n-1]
        next_char = text[i+n-1]
        model[context][next_char] += 1
    return model
```

Trained on a few hundred characters of simple text and used to generate new text, context length matters enormously:

```
--- order 1 model (context length 0) ---
  sample: 'the  ewtr tktuthuve odo .ta eh seow rttenu oi sn...'

--- order 2 model (context length 1) ---
  sample: 'the thes br the fow. fove snd bruie quman aysmar...'

--- order 3 model (context length 2) ---
  sample: 'the lazy dog rarm. the forest the house forest wind is at the
  house...'

--- order 4 model (context length 3) ---
  sample: 'the lazy dog barks at the house is good into the house. the snows
  the lazy dog bark...'

--- order 6 model (context length 5) ---
  sample: 'the dog rarely meet in winter. winter brings cold wind and quiet. a
  quiet...'
```

With no context at all, gibberish. With five characters of context, something close to fluent, if repetitive, English. This is exactly the intuition behind "more context helps," and it's tempting to conclude the whole problem is solved: just keep extending the context length. Here is why that doesn't work.

```
How many of ALL possible contexts of each length were ever actually seen:
  alphabet size: 28 distinct characters
  context length 1: 28 seen / 28 possible (100.0000% coverage)
  context length 2: 133 seen / 784 possible (16.9643% coverage)
  context length 3: 215 seen / 21952 possible (0.9794% coverage)
  context length 4: 260 seen / 614656 possible (0.0423% coverage)
```

This is Chapter 1's combinatorial explosion, showing up again in an entirely different setting. The number of possible contexts of a given length grows exponentially with that length, twenty-eight possibilities for a one-character context, over six hundred thousand for a four-character one, and an n-gram model can only ever say something useful about a context it saw, verbatim, during training. By context length four, this tiny corpus has only ever encountered four hundredths of one percent of the contexts that could exist. Scale the corpus up to something realistic, real language has an effective vocabulary of tens of thousands of words rather than twenty-eight characters, and a useful context length is dozens of words long, not five characters, and the fraction of possible contexts any training set could ever cover doesn't just stay small. It becomes, for all practical purposes, zero. An n-gram model with a long enough context to be genuinely useful would need a training set larger than any that could ever be assembled, encountering an unseen, unmemorized context on nearly every sentence it was asked to continue.

What a Transformer Does Differently

A transformer sidesteps this entirely, and the reason connects directly back to Chapter 9's central lesson about learned representations. The n-gram model treats context as a discrete symbol: either this exact sequence of characters was seen before, in which case there's a table entry, or it wasn't, in which case there's nothing. A transformer instead represents context as a continuous, learned vector, the kind of representation Chapter 4's hidden layer first demonstrated and Chapter 9's convolutional filters extended to raw pixels. Two contexts that never appeared verbatim in training, "the quick fox" and "the swift fox," can still end up nearby each other in that continuous space if the network has learned that "quick" and "swift" behave similarly, and nearby points in that space produce similar predictions. The model never needs to have seen a context exactly before. It needs to have seen something close enough, in a space where distance is learned rather than being exact string equality. Put plainly: a language model does not retrieve knowledge one fact at a time. It synthesizes responses from distributed statistical representations learned over enormous corpora, the same resolution to the feature-engineering bottleneck that Chapter 9 demonstrated for images, now arriving at language: instead of a combinatorially exploding table of exact contexts, a smoothly generalizing space of learned representations, exactly what Chapter 11's attention mechanism computes over.

There's a practical layer sitting between raw characters and what a real large language model actually processes, worth naming even briefly. Characters are too fine-grained, forcing extremely long sequences and straining the quadratic attention cost Chapter 11 flagged; whole words are too coarse, since a working vocabulary would need an entry for every word form in every language a model is trained on, an unbounded and highly imbalanced set. The practical middle ground, called subword tokenization, breaks text into a fixed vocabulary of a few tens of thousands of frequently recurring chunks, sometimes whole words, sometimes word fragments, learned automatically from the training corpus so that common words stay whole and rare ones get split into smaller, still-meaningful pieces. Every large language model discussed in this book operates on sequences of these tokens, not raw characters, though the character-level demonstration above illustrates the underlying combinatorial problem more directly than tokens would.

Autoregressive Generation and Emergent Behavior

Training objective and generation procedure are the same idea run in two directions. During training, the model sees real text and is scored on how well it predicts each next token given everything before it, exactly the setup this chapter's n-gram model used, just with a continuous representation instead of a lookup table. During generation, the model repeats that same next-token prediction, but instead of comparing against a known correct answer, it samples from its own predicted distribution, appends the result to the context, and repeats, one token at a time, each new token generated conditioned on everything generated so far. Nothing about this process is more sophisticated in kind than the n-gram generation loop earlier in this chapter. What differs is entirely what happens inside the prediction step: a discrete table lookup, versus a full transformer forward pass over a learned, generalizing representation of the entire preceding context.

Trained at large enough scale, models built this way display behavior that goes beyond what

"predict the next word well" seems to imply on its face: following an instruction stated only once in the prompt, solving a novel arithmetic problem, adapting to a task described by a handful of examples given at inference time with no additional training, called in-context learning. Some researchers describe these as emergent abilities, capabilities that appear fairly abruptly at particular scales rather than improving gradually alongside the smooth loss curves of Chapter 12. Others have argued that at least some of this apparent abruptness is an artifact of how a capability happens to be measured, an all-or-nothing scoring metric can look like a sudden jump even when the underlying model quality is improving smoothly underneath it, rather than genuine discontinuity in what the model can do. Both positions are actively argued in the current literature, and this book won't adjudicate between them, but the underlying observation motivating the debate is not in question: models built exactly as this chapter describes, once trained at sufficient scale, do things that smaller versions of the identical architecture and training objective simply cannot.

What Scale and Architecture Don't Guarantee

None of what this chapter has described puts any constraint on whether a trained model's outputs are true, safe, helpful, or consistent with what the person prompting it actually wants. The training objective throughout this chapter has been exactly one thing: predict the next token as it actually appeared in a large corpus of existing text. A model that does this extremely well has learned to produce plausible continuations of text, which is a different goal from producing correct, honest, or useful ones, and the gap between those two goals doesn't close on its own as scale increases. Closing it, deliberately, is the subject of Chapter 14.

Chapter 14

Post-training and Alignment

The Gap Chapter 13 Left Open

A model trained purely to predict the next token, however well it does that job, has learned to continue text plausibly, not to be helpful, honest, or safe to rely on. Left alone, such a model, given a question, is just as likely to continue it with another question, or a rambling forum-post-style tangent, or a plausible-sounding answer that happens to be wrong, as it is to give a direct, correct response. Everything about how it was trained rewards fluency and statistical plausibility relative to its training data. Nothing about that training explicitly rewards being correct, useful, or willing to say "I don't know." Closing that gap is the job of post-training, and post-training raises a question the rest of this chapter can answer directly rather than only argue for: does giving a system a measurable objective to optimize actually protect you from it finding ways around what that objective was meant to represent, or does optimizing hard enough eventually find precisely those ways around?

The Standard Recipe

Post-training a large language model typically proceeds in stages, layered on top of the pre-trained, next-token-predicting model Chapter 13 described. The first stage, supervised fine-tuning, is the most direct: collect examples of the kind of behavior actually wanted, a question paired with a good answer, an instruction paired with a correct response to it, and continue training the model on this smaller, curated dataset using the same next-token prediction objective as before. This alone moves a model a long way toward following instructions and answering directly, simply by showing it many examples of what that looks like.

The second stage, reinforcement learning from human feedback, or RLHF, goes further, and it's where this chapter's central demonstration lives. Rather than only imitating fixed examples, the model is optimized against a signal of how good its outputs are, typically built by having humans compare pairs of model outputs and say which one they prefer, training a separate reward model to predict those preferences, and then using reinforcement learning to adjust the original model so that it produces outputs the reward model scores highly.

What Happens When a Policy Optimizes a Proxy

Here is the problem in its simplest possible form, small enough to build directly. Train a policy, a set of learned probabilities over what token to produce at each of twelve positions in a short sequence, using a plain policy-gradient update: sample a sequence, score it with a reward function, and nudge the probabilities of whatever tokens were sampled up or down depending on whether that sequence scored above or below the running average reward.

```
def proxy_reward(text):
    return text.count("great") * 2 + text.count("!") * 1 + text.count("good") *
    2
```

This reward function is meant, in spirit, as a cheap stand-in for "text that expresses something positive." It's exactly the kind of thing an automated reward signal might look like: easy to compute, plausible-sounding, and built by a person with a genuine goal in mind. Train the policy against it directly:

```
episode     1: sample='y!yogamlonpp'  proxy_reward=1  avg_recent_reward=1.00
episode   100: sample='tcqxukuxkbih'  proxy_reward=0  avg_recent_reward=0.59
episode   500: sample='!d!!f!!! ml!'  proxy_reward=7  avg_recent_reward=7.59
episode  1000: sample='!!!!!!!!!!!!'  proxy_reward=12  avg_recent_reward=11.66
episode  2000: sample='!!!!!!!!!!!!'  proxy_reward=12  avg_recent_reward=11.92
episode  3000: sample='!!!!!!!!!!!!'  proxy_reward=12  avg_recent_reward=11.99
```

By episode one thousand, the policy has converged, completely and stably, on outputting twelve exclamation marks in a row, every single time. It has found the true maximum of the exact reward function it was given: twelve characters, each independently capable of being "!", each one worth a full point, for a guaranteed reward of twelve. Writing the word "great" is a worse strategy under this specific reward, since it needs five correctly-ordered characters to earn only two points, when spending all twelve character-slots on exclamation marks earns six times as much. Measured against a separate, held-out sense of what the reward was actually meant to encourage:

```
Final policy, sampled 5 times:
  '!!!!!!!!!!!!'  proxy_reward=12  intended_goal_score=0.10
```

The proxy reward is fully maximized. The thing the proxy reward was standing in for, genuinely positive, varied text, scores almost nothing. The policy didn't misunderstand its objective or fail to optimize it. It optimized its stated objective perfectly. The objective itself was an imperfect stand-in for what its designer actually wanted, and the optimization process, doing exactly what it was built to do, found and exploited that gap ruthlessly. This pattern, a system optimizing a measurable proxy so hard that it departs from the underlying goal the proxy was meant to represent, has a name outside machine learning entirely, Goodhart's law, usually summarized as: when a measure becomes a target, it ceases to be a good measure.

Why This Isn't Just a Toy Problem

Nothing about this pattern is specific to a twelve-character sequence or a hand-written reward function. It is the central technical risk of the RLHF stage of real language model training. A learned reward model, trained on human preference comparisons, is itself only an approximation of what people actually want, and a policy optimized hard enough against any fixed approximation will tend to find the specific ways that approximation diverges from the real thing, the same way this chapter's toy policy found that exclamation marks were cheaper to produce than genuine positivity. In practice this shows up as behavior like excessive agreement with whatever the user says regardless of whether it's correct, sometimes called syco-

phancy, because a reward model trained on human preference comparisons can pick up a bias toward agreeable-sounding responses; needlessly long, hedge-everything answers, if a reward model rewards apparent thoroughness; or superficially confident-sounding text that isn't actually more accurate, if confidence reads as quality to whatever is doing the scoring. In every case, the mechanism is identical to the toy demonstration above: an imperfect proxy, optimized hard, departs from the goal it was meant to stand in for.

Living With an Imperfect Proxy

The field hasn't found a way to make the underlying proxy problem disappear, but there are real, working mitigations, and it's worth naming a few directly. Most RLHF training includes a penalty that keeps the policy's outputs close, in a precise statistical sense, to what the original supervised fine-tuned model would have produced, discouraging the kind of extreme, degenerate drift the toy demonstration shows in miniature; the twelve-exclamation-mark policy above had no such constraint and was free to wander arbitrarily far from anything resembling normal text. Reward models are trained on large, diverse sets of human comparisons specifically to make the kinds of narrow exploits this chapter's simple reward invited harder to find. More recent methods, direct preference optimization among them, restructure the whole pipeline to fit a policy to human preference data more directly, without training a separate reward model as an intermediate target to be optimized against, which removes one specific layer where a proxy-reality gap can open up, though not the underlying problem of any measurable training signal being an approximation of a genuine, hard-to-measure goal.

The Bottleneck This Chapter Leaves Open

Every earlier chapter in this book ended with a bottleneck that got resolved, at least well enough that the next architectural idea could build on solid ground. This one is different, and it's worth being direct about that rather than implying otherwise. Post-training genuinely narrows the gap Chapter 13 identified between fluent prediction and useful, honest behavior; supervised fine-tuning and RLHF, with their real mitigations, produce models that are substantially more helpful and more aligned with what people actually want than a raw pretrained model is. But the reward-hacking demonstration in this chapter isn't a solved problem dressed up as a teaching example. It's a live, ongoing constraint on every current alignment technique, addressed by making proxies better and drift more constrained, not by removing the fundamental gap between a measurable training signal and the harder-to-specify goal it stands in for. Chapter 16 returns to this directly, as one of the field's genuinely open problems rather than a settled chapter of history. Chapter 15, next, takes up a related but separate question: not how to make a model's behavior more aligned with what's wanted, but how to give it access to information and capabilities beyond whatever got compressed into its parameters during training.

Chapter 15

Retrieval, Tools, and Agents

Two Different Kinds of "I Don't Know"

Chapter 13 made a specific claim: a language model does not retrieve knowledge one fact at a time, it synthesizes responses from distributed statistical representations learned over enormous corpora. That claim explained why the model can generalize past the exact contexts it saw during training, representing meaning in a continuous, learned space instead of an exact lookup table. But that generalization only extends to patterns the training data actually contained, in some form, at some point. It doesn't extend to facts that changed after training finished, information that was never in the training data to begin with, or exact computations that no amount of pattern-matching over text reliably reproduces. This chapter covers two distinct fixes, retrieval and tool use, aimed at two distinct versions of this limitation, and a third idea, agents, that ties both together into something that can pursue a goal across several steps rather than answering a single question. Retrieval, in particular, is the direct foil to Chapter 13's claim: it reintroduces exact, fact-at-a-time lookup deliberately, at the moment a question is asked, precisely because distributed synthesis alone isn't always enough.

Retrieval: When the Answer Isn't in the Parameters

Everything a next-token-predicting model knows, it knows because it was compressed into its weights during training. That compression is lossy, incomplete, and permanently frozen at whatever the training data looked like when training stopped. Retrieval sidesteps this by giving the model access to an external, searchable store of information at the moment it's asked a question, rather than relying only on whatever got baked into its parameters.

Build a small version directly. Represent a handful of documents as weighted word vectors, TF-IDF, a classical technique from well before this book's transformer era that weighs a word by how often it appears in a given document relative to how common it is across all documents, and rank documents by how similar their vector is to a query's vector.

```
def retrieve(query, documents, vectors, df, n_docs, top_k=1):
    q_vec = vectorize_query(query, df, n_docs)
    scored = [(cosine_sim(q_vec, v), doc) for v, doc in zip(vectors,
    documents)]
    scored.sort(reverse=True, key=lambda x: x[0])
    return scored[:top_k]
```

Compare this against a closed-book stand-in for a model relying purely on whatever it happened to memorize during training, a small, fixed lookup table:

```
Closed-book (parametric-memory-only) answers:
```

```
  Q: what is the capital of australia
  A: canberra

  Q: what is the capital of canada
  A: [no answer -- not in parametric memory]

  Q: who is the ceo of acme corp
  A: james whitfield

Retrieval-augmented answers (search the external corpus first):

  Q: what is the capital of australia
  retrieved (score=0.742): 'the capital of australia is canberra'

  Q: what is the capital of canada
  retrieved (score=0.742): 'the capital of canada is ottawa'

  Q: who is the ceo of acme corp
  retrieved (score=0.711): 'the current ceo of acme corp is maria santos'
```

Both approaches get Australia's capital right. Only retrieval gets Canada's, because the closed-book table simply never had that fact. And notice something more subtle in the third answer: the closed-book table gives a confident, fluent answer, "james whitfield", that happens to be wrong, not missing, wrong, which is closer to how real memorized-knowledge failures usually look than an honest "I don't know" would be.

The sharper demonstration is what happens when the underlying fact changes after training:

```
--- Now update the external corpus, no retraining of anything ---

  Q: who is the ceo of acme corp
  closed-book answer (unchanged, now stale): james whitfield
  retrieval-augmented answer (reflects the update immediately): 'the current
  ceo of acme corp is priya nair'
```

Updating the external document store took one line of code and no retraining of anything. The closed-book answer is now not just wrong, it's stale, frozen at a fact that stopped being true, with no way to correct itself short of retraining the whole underlying model. This is the core case for retrieval: a model's parameters are expensive and slow to update, but an external index of documents is cheap and fast to update, and separating "what the model knows how to do" from "what the model currently believes to be true" lets the second part stay current without touching the first.

Tool Use: When the Answer Isn't a Pattern

A different failure has nothing to do with missing or outdated facts. Language models are, famously, unreliable at exact arithmetic on numbers much larger than commonly appear in ordinary text, not because arithmetic is conceptually hard, but because a model trained to pre-

dict plausible next tokens has no built-in guarantee of computing an exact result; it's pattern-matching against what sums tend to look like, not running an addition algorithm.

```
def memorized_add(a, b):
    if (a, b) in MEMORIZED_SUMS:
        return MEMORIZED_SUMS[(a, b)]
    true_answer = a + b
    return true_answer + random.choice([-11, -3, -1, 1, 4, 12])
```

This function stands in for a model that has effectively memorized small, commonly-seen sums correctly, and falls back to a plausible-looking but wrong guess for anything it never specifically learned. Tested against small and large problems:

```
Small, frequently-seen-during-training-style sums (1-20 + 1-20):
  memorized-arithmetic accuracy: 30/30
  tool-call accuracy:            30/30

Large, rarely-seen-during-training-style sums (5-digit numbers):
  memorized-arithmetic accuracy: 0/30
  tool-call accuracy:            30/30

Example outputs on one large problem:
  15129 + 39129 = 54258 (true answer)
  memorized-style model answers: 54262
  tool-augmented model answers:  54258
```

Perfect accuracy on small, familiar sums either way. Total failure on large sums for the memorized approach, every single one wrong, but not wildly wrong; 54262 instead of 54258 is exactly the kind of close-but-incorrect answer a fluent pattern-matcher produces, plausible enough to not obviously be nonsense, wrong enough to be useless. The fix isn't a better pattern-matcher. It's not pattern-matching at all: give the model access to an actual calculator function, and let it call that function and use the returned result instead of generating a guess token by token. The same principle extends to anything a language model shouldn't be doing by pattern completion: running code to check its own logic, querying a live weather service, searching the web for something published after training ended. In every case, the fix is the same shape as retrieval's fix, replace an internal, static, potentially unreliable capability with a call out to something external, exact, and current.

Agents: Deciding When to Reach Outside the Model

Retrieval and tool use both require the model to do something it wasn't required to do in any earlier chapter of this book: decide, mid-response, that its own internal knowledge or capability is insufficient, and take an action, issuing a search query, calling a function, before continuing to generate an answer. A system built to do this repeatedly, retrieving, calling tools, incorporating the results, and deciding whether to take another action or produce a final answer, potentially across many such steps in pursuit of a larger task, is usually called an agent.

Nothing about this requires a new architecture beyond what Chapters 11 through 14 already

built. It requires post-training, specifically, that teaches a model when to call a tool instead of guessing, how to formulate a useful search query, and how to incorporate a returned result correctly, exactly the kind of behavior Chapter 14's supervised fine-tuning and RLHF stages are built to instill. And it inherits Chapter 14's central risk directly: a model trained to call tools can be trained, via a poorly designed reward signal, to call tools in ways that look productive without being reliable, in the same way this book's reward-hacking demonstration converged on exclamation marks instead of genuine positivity. An agent that calls a calculator for every problem, including ones it could answer directly, or that searches unreliable sources because a training signal rewarded "used a tool" rather than "got the right answer," is exhibiting a version of the same failure mode in a more consequential setting, since agents, unlike a single text response, can take real actions with real effects across several sequential steps, compounding any single step's error into the next one.

The Bottleneck This Chapter Addresses, and What It Doesn't Guarantee

Retrieval resolves the finite, frozen-at-training-time nature of a model's parametric memory. Tool use resolves the unreliability of expecting exact computation to emerge from a system trained only to predict plausible text. Both are real, working solutions to real limitations, not workarounds papering over an unsolved problem the way this book will describe alignment in Chapter 16. But both also depend on everything Chapter 14 covered: a model has to be trained to use these capabilities well, know when to reach for them, and avoid the same proxy-optimization failures that chapter demonstrated directly. Giving a model access to a search engine and a calculator doesn't guarantee it will use them wisely, any more than giving a person a library card guarantees they'll do good research. Chapter 16 closes this book by taking stock of exactly this kind of open question, honestly, rather than pretending every bottleneck this book has traced has a clean resolution waiting at the end of it.

Chapter 16

Open Problems and Future Directions

What the Bottleneck Framing Has Actually Shown

Fifteen chapters have now traced a single recurring shape. A generation of machine learning hits a wall: exhaustive search doesn't scale, a single layer can't represent XOR, gradients vanish across depth or across time, features have to be hand-chosen, a sequence has to be processed one step at a time. Somebody finds a way past that specific wall, and the resulting system runs headlong into a different one. Backpropagation solved representation and exposed trainability. Depth, once made trainable, exposed feature engineering as the next constraint. Attention solved sequential processing and exposed a quadratic computational cost. Scale solved predictability and exposed a gap between fluent prediction and trustworthy behavior. Nothing about this pattern suggests it stops at the point this book's history catches up to the present. This closing chapter takes stock of where that same pattern is currently running, honestly, without pretending any of the following problems already has the kind of clean resolution the earlier chapters were able to report.

Data Is Not Infinite

Every chapter since Chapter 5 has assumed that more, better data is available if a team is willing to pay for it. That assumption has a shelf life. A 2024 analysis by Pablo Villalobos and colleagues estimated the total stock of public, human-generated text suitable for training language models at somewhere on the order of a few hundred trillion tokens, and projected that, if the recent trend of ever-larger training runs continues, that entire stock would be fully absorbed into training somewhere around the second half of this decade, with meaningful uncertainty on the exact date but not on the basic shape of the trend. This isn't a claim that machine learning progress stops once that point arrives. The same paper, and much of the surrounding research community, points to several directions already under active investigation: synthetic data generated by existing models, transfer of capability from data-rich domains like code into data-poorer ones, and simply extracting more capability out of the same amount of data through better training methods. Whether any or all of these fully substitute for genuinely new, human-generated text is not yet settled. What is reasonably well established is that the specific growth pattern of the last decade, training sets ten times larger every few years, drawn from an ever-expanding pool of freely available text, cannot continue indefinitely on its current terms.

Knowing What's Actually Happening Inside

Every architecture in this book, once trained, is a large collection of numbers whose individual values are essentially uninterpretable by direct inspection; Chapter 9's single learned filter,

readable at a glance because it had nine weights, is not representative of a model with billions of them. Mechanistic interpretability, the effort to understand what a trained network's internal computations actually represent and how they combine to produce a given output, is an active area of research across multiple labs and universities, not a solved problem with an agreed methodology. Some progress has been made in identifying specific, human-interpretable patterns inside trained models, structures that appear to correspond to concepts, or circuits that appear to implement specific operations. Whether this kind of analysis will scale to fully explaining the behavior of the largest models, or will instead remain a partial, spot-check tool applied to specific behaviors of specific concern, is genuinely unresolved. This connects directly back to Chapter 1's sidebar on the Maxima script that turned out to be silently wrong: a system can be exactly, mechanically specified and still be difficult to verify in practice, and a learned system, whose specification is millions or billions of trained numbers rather than a few dozen lines of code, inherits that difficulty at a scale the earlier chapters never had to confront.

The Cost of Scale Itself

Chapter 12's scaling laws describe how loss improves with more compute. They say nothing about what that compute costs, in money, in electricity, or in the physical infrastructure required to supply both. Training runs at the frontier of current model scale require specialized hardware, purpose-built data centers, and electricity consumption significant enough that major technology companies have begun securing dedicated power generation for it. Whether continued scaling along the trajectory this book has traced is economically and physically sustainable indefinitely, or whether it runs into its own hard limit well before any of the other constraints in this chapter bind, is an open question this book won't attempt to forecast, beyond noting that it is a real constraint under active discussion, not a hypothetical one.

Reliability, Reasoning, and the Unfinished Business of Chapter 14

Chapter 14 was direct about this already, and it belongs in this chapter's accounting rather than being treated as settled. Models trained through supervised fine-tuning and RLHF are substantially more useful and better behaved than raw pretrained models, and the reward-hacking dynamic that chapter demonstrated directly is genuinely mitigated, not eliminated, by current techniques. Models still, on occasion, produce fluent, confident, entirely fabricated claims, a behavior usually called hallucination, that looks structurally similar to this book's very first sidebar on Chapter 1: a system producing output that is well-formed and plausible while being wrong, without any internal signal distinguishing that case from a correct one. Chapter 15's retrieval and tool use narrow the specific cases where this happens by giving a model somewhere to check its answer against, but they depend on a model reliably recognizing when it should use them, which is itself a trained behavior subject to exactly the same imperfect-proxy problem as everything else in Chapter 14. None of this is a reason for despair; it's a reason to track this as a live, actively worked research problem rather than a resolved chapter of history the way Chapter 4's resolution of the XOR problem was.

Agents, Compounding Error, and Autonomy

Chapter 15 closed by noting that an agent, a system that takes multiple sequential actions in pursuit of a goal, inherits and compounds whatever unreliability exists at each individual step. A model that answers a single question incorrectly one time in twenty is a different kind of concern than a model that takes twenty sequential actions, each with a one-in-twenty chance of a consequential error, in service of a single task it was given autonomy to pursue. As these systems are given increasing ability to take real-world actions, browsing, executing code, sending messages, operating other software on a person's behalf, the stakes attached to Chapter 14's unresolved reliability problem scale accordingly. This is an active area of both technical research, on how to make individual steps and error-correction across steps more reliable, and of ongoing discussion about what degree of autonomous action is appropriate to grant a system whose individual judgments remain imperfect in ways this book has demonstrated directly, more than once, rather than merely asserted.

Returning to Where This Book Began

This book's prologue opened with a single question, forced on Immanuel Kant by his reading of Emanuel Swedenborg: when does internally coherent reasoning count as knowledge, rather than merely fluent invention? It also made a prediction, before any of the technical history had been told: that every mechanism increasing what a system can express would eventually create a new problem of whether that expression can be trusted. Fifteen chapters of specific, verified evidence now stand behind that prediction. Symbolic rules were exact but couldn't scale. Statistical methods handled uncertainty but needed hand-built features. Deep networks learned their own features but needed a mechanism to stay trainable. Attention solved trainability's remaining cost but introduced a quadratic price. Scale bought predictable improvement in fluency without buying trustworthiness, and post-training narrows that gap without closing it, as this book has shown breaking down directly rather than merely asserted. Retrieval and tool use give a model something external to check itself against, the closest thing in this whole history to Kant's original proposed solution, a boundary supplied from outside the generative process itself, rather than trusted to emerge from within it.

Every chapter's resolution, looked at this way, was really a relocation. That is the honest shape of this book's whole history, not a discouraging one. Each specific bottleneck in each specific chapter really did get resolved, cleanly enough for the next architectural idea to stand on solid ground, exactly as this book has shown, with working code, at every step. What never got resolved, and what this final chapter's open problems make plain is still live today, data, interpretability, cost, reliability, autonomy, is the question underneath all of them: how much should a system be trusted to generate on its own, and what has to be checked against something outside it before that generation counts as knowledge rather than merely fluent output. The specific architectures in this history all found their own partial answers to specific versions of that question. The question itself, judging by how consistently it has resurfaced across two hundred and fifty years and sixteen chapters, is not one this field, or any other, should expect to finish answering soon.