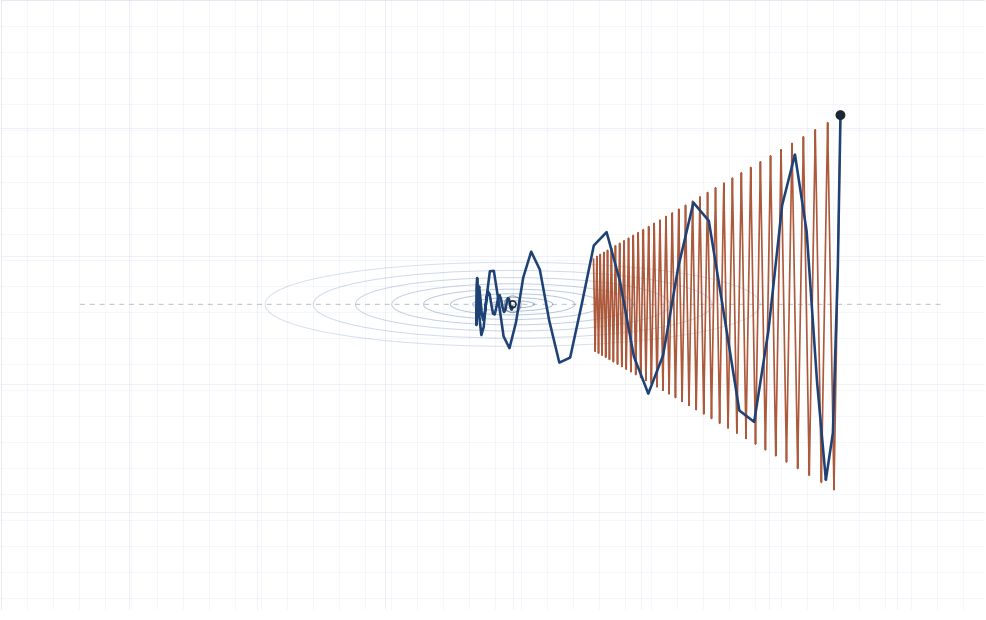

Gradient descent and momentum on an ill-conditioned valley — Chapter 3

The Arithmetic of Intelligence

A problem course in modern machine learning, from the pathological valley to direct preference optimization.

ANSHU AVINASH

Preface

This book teaches machine learning the way physics is taught to engineers: as a small set of results you derive, drill, and then recognize everywhere.

Most introductions to the field are surveys. They tell you *that* residual connections help, *that* larger models should see more data, *that* a preference-tuning method called DPO replaced a more complicated pipeline. A survey leaves you fluent in names and helpless in front of a design decision. When someone asks whether your serving cluster is memory-bound, whether a benchmark jump is capability or measurement, or why a reinforcement-learning recipe deleted its critic network, the names do not answer. A short calculation does.

The premise here is that modern machine learning — for all its scale and noise — rests on roughly forty short calculations. Each is the kind of thing a first-year engineering student could do, if someone set it up properly: a stability condition, a variance count, a constrained minimization, a geometric series, a Bayes flip. The field's landmark papers are these calculations wearing costumes. Learn the calculations and the costumes turn transparent; when next quarter's paper arrives, you will see within a page which budget it spends and which derivation prices it.

Accordingly, nothing in this book asks you to memorize a benchmark score, a release date, or an author list. Where a constant from the literature is needed — a scaling exponent, a data-law slope, an accelerator's bandwidth — it is *given*, the way $g = 9.8\ \mathrm{m/s^2}$ is given in a physics paper. What is never given is the model you must build to use it. That is the work, and it is the only thing examined.

The book covers two eras in one arc. Part II is the classical era, 2012–2022: why training deep networks was hard, what architecture did about it, how scale became a budgeting problem, and why compression is the master currency underneath all of it. Part III is the modern era, 2022 onward: the economics of inference, attention rebuilt around memory traffic, sparsity and thrift, the alignment pipeline, inference-time reasoning, diffusion, and the full-lifecycle cost of a deployed model. The ordering inside Part III is deliberate — systems before

alignment — because without the cost vocabulary of bytes, bandwidth, and budgets, the alignment era's design choices read as fashion instead of forced moves.

How to use this book

The unit of progress is a derivation, not a chapter

Appendix A lists the thirty-eight core derivations. You are done with a chapter when you can produce its derivations on blank paper, unprompted — not when you have read it. Three of them (the compute-optimal allocation of Chapter 7, the DPO collapse of Chapter 14, and one you meet early in Chapter 3) carry a special protocol: reproduce them from nothing on three separate days before moving on.

Work every drill; keep an error log

Drills appear inside sections with their answers one click away; end-of-chapter exercise sets are graded **A** (drills), **B** (problems), and **C** (challenges), with full solutions in Appendix D. Keep a notebook with one line per mistake: what you did, what was right, and which trap class it belongs to. The trap taxonomy (Appendix B) names the ten ways students reliably go wrong. Twenty hours of honest error-logging is worth more than any rereading.

Let code confirm the math

From Chapter 11 onward a lab manual (Appendix C) runs in parallel. Every lab's acceptance criterion is a number your derivation predicted: your streaming softmax must match a library implementation to six decimal places; your preference-tuning loss must begin at exactly $\ln 2$ — and you must prove why before you run it. When code and derivation disagree, one of them is wrong, and finding which is the curriculum.

PREREQUISITES

School-level mathematics — logarithms, elementary calculus, basic probability — and, for the labs, working Python. Chapter 1 sharpens eight mathematical reflexes and nothing else; if its gate feels easy, skip ahead without guilt. You are also assumed to know roughly what a neural network is — a stack of weighted sums and nonlinearities, trained by following a gradient. Chapter 3 states every such object precisely before it is used, but it states them; it does not motivate them.

Contents

PART I

Foundations

The Mathematical Toolkit

Eight small reflexes carry the entire book. None is difficult; all must become automatic before the machine learning begins.

Eight small pieces of mathematics carry this entire book. You almost certainly know all of them already. What you may not have is the ability to use them without thinking — and that, not knowledge, is what the chapters ahead will demand.

A physicist solving a mechanics problem does not rederive the chain rule; it is furniture in the mind, and attention goes to the physics. The same must be true here. When Chapter 7 asks you to minimize a loss under a compute constraint, the minimization itself cannot be where you spend effort — the insight is in what the answer *means*. So we drill the furniture first. Read each section, work its drills until they are mechanical, and pass the gate at the end before opening Chapter 2. If the gate is trivial for you, skip this chapter entirely; it exists only to remove obstacles, not to teach anything you will be tested on for its own sake.

The eight reflexes are: manipulating **logarithms and power laws**; the **algebra of variance**; the **binomial distribution and its normal approximation**; **constrained minimization by substitution**; **geometric series**; **counting in bits**; **turning a conditional around**; and **counting operations**. Each recurs a dozen times in what follows, and I flag the first major appearance of each as we go.

1.1 Logarithms and power laws

Nearly everything about scale — how loss falls as models grow, how error falls as data grows, how a probability compounds over a thousand steps — is a statement about power laws, and every power law is a straight line once you take logarithms. The reflex to build is moving between the two views without hesitation.

The one manipulation you will perform most often is solving $a^t = b$ for the exponent. Take logarithms of both sides: $t \ln a = \ln b$, so $t = \ln b / \ln a$. The base does not

matter as long as it is the same on top and bottom. A special case worth having in your fingers: the number of times you must multiply by a factor $f < 1$ to fall below a threshold ϵ is $t = \ln \epsilon / \ln f$, and because both logarithms are negative the ratio is positive.

The second manipulation is extracting an exponent from a verbal description of a power law. Suppose a quantity y depends on x as $y \propto x^{\beta}$, and you are told that multiplying x by ten multiplies y by 0.6 — a "forty percent relative reduction per tenfold increase." Then $10^{\beta} = 0.6$, so $\beta = \log_{10} 0.6 \approx -0.22$. Once you have β, any other question about the law — how much x must grow to halve y, say — is one more line of the same algebra. This exact move recovers a speech-recognition data law in Chapter 5 and the compute-optimal exponents in Chapter 7.

> **WORKED EXAMPLE 1.1**
>
> **A probability that compounds**
>
> A recurrent network multiplies a gradient by 0.95 at each time step. After how many steps has the gradient shrunk below one percent of its initial size?
>
> We need the smallest t with $0.95^{t} < 0.01$. Taking logarithms, $t > \ln(0.01)/\ln(0.95) = (-4.605)/(-0.0513) = 89.8$, so $t = 90$. A factor that looks almost harmless — ninety-five hundredths — erases the signal in about ninety steps. Chapter 5 makes this the whole reason the LSTM was invented.

Three constants are worth committing to memory so these estimates need no calculator: $\ln 2 = 0.693$, $\log_{10} 2 = 0.301$, and $\ln 10 = 2.303$. From the first two you can reconstruct most of what you need; for instance, the number of bits in a quantity is its natural-log information divided by 0.693.

DRILL 1.1

(a) Solve $0.9^{t} = 0.5$. (b) A loss term behaves as $L \propto N^{-0.076}$; by what factor must N grow to cut this term by one quarter? (c) A model reaches a perplexity of 64; what is its cross-entropy in bits per token? (Perplexity is 2^{H} with H in bits.)

Show answers

(a) $t = \ln 0.5 / \ln 0.9 = (-0.693)/(-0.105) = 6.58$. (b) We need $N^{-0.076}$ to reach 0.75 of its value, i.e. $k^{-0.076} = 0.75$ where N grows by factor k; so $k = 0.75^{-1/0.076} = 0.75^{-13.16}$. Since $\ln 0.75 = -0.288$, $\ln k = 0.288 \times 13.16 = 3.79$, $k = e^{3.79} \approx 44$. (c) $H = \log_2 64 = 6$ bits per token.

1.2 The algebra of variance

When many small independent effects combine, their *variances* add — not their standard deviations. This single fact, applied without fear, explains why a signal grows like the square root of depth, why attention scores must be rescaled by the square root of dimension, and why noise injected at every step of a recurrence is so much more destructive than noise injected once per layer. It is the most quietly powerful tool in the book.

Three rules suffice. First, for independent random variables the variance of a sum is the sum of variances: $\operatorname{Var}(\sum_i X_i) = \sum_i \operatorname{Var}(X_i)$. Second, scaling a variable scales its variance by the square: $\operatorname{Var}(aX) = a^2 \operatorname{Var}(X)$. Third, the product of two independent variables, each with zero mean and unit variance, itself has unit variance. From these three, every result we need follows in a line or two.

DERIVATION

The variance of a weighted sum of inputs

Consider a single artificial neuron that forms the weighted sum $s = \sum_{i=1}^{m} w_i x_i$ of m inputs. Suppose the inputs are independent with unit variance, and the weights are independent, drawn with mean zero and variance σ^2. What is the variance of s?

Each term $w_i x_i$ is a product of independent zero-mean variables, so it has mean zero and variance $\mathrm{Var}(w_i)\,\mathrm{Var}(x_i) = \sigma^2 \cdot 1 = \sigma^2$. The terms are independent of one another, so their variances add:

$$\mathrm{Var}(s) = \sum_{i=1}^{m} \sigma^2 = m\sigma^2.$$

This is the entire content of "fan-in scaling." If we want the neuron's output to have the same scale as its inputs — neither exploding nor collapsing as signal passes through the layer — we must choose $\sigma^2 = 1/m$. We will meet this exact condition again in Chapter 3 as the principle behind sensible initialization, and its cousin in Chapter 6, where the fan-in is a vector dimension and the same square root sets the scale of attention.

Two consequences deserve to be stated on their own, because each is a derivation you will be asked to reproduce. The dot product of two random d-dimensional vectors, each component independent with unit variance, has variance d — just apply the derivation above with $m = d$ and unit weights. And the accumulation of L independent perturbations, each of variance v, has variance Lv and therefore standard deviation $\sqrt{Lv}$: a drift that grows like the square root of the number of contributions. Chapter 4 uses precisely this to explain why signals swell as they pass down a hundred-layer residual stack.

DRILL 1.2

(a) Two random vectors of dimension 288 have independent unit-variance components. What is the standard deviation of their dot product? (b) Sixty-four independent contributions, each of variance 0.04, accumulate at one point. What is the resulting standard deviation?

Show answers

(a) Variance $= 288$, standard deviation $= \sqrt{288} = 12\sqrt{2} \approx 17.0$. (b) Variance $= 64 \times 0.04 = 2.56$, standard deviation $= 1.6$.

1.3 The binomial distribution and its normal approximation

Whenever something can go one of two ways — a residual block is traversed or skipped, a reasoning chain is right or wrong, a token is correct or not — and the trials are independent, the count of "successes" follows a binomial distribution. You need three things about it: its shape, its summary statistics, and how to approximate it with a bell curve when the counts get large.

For n independent trials each succeeding with probability p, the probability of exactly k successes is $\binom{n}{k} p^k (1-p)^{n-k}$. The mean number of successes is np and the variance is $np(1-p)$; take the square root of the latter for the standard deviation. When n is large, the distribution is well approximated by a normal curve with that same mean and standard deviation, and probabilities over a range of counts become areas under the bell — read off from the standard normal function Φ. Because the binomial is discrete and the normal continuous, we widen the target interval by half a unit at each end; this *continuity correction* is the little half-step that keeps the approximation honest.

$\Phi(z)$ is the probability that a standard normal variable falls below z, and a symmetric interval $[-z, +z]$ therefore holds $2\Phi(z) - 1$. You will need a few of its values with no table to hand, so here they are.

Φ: 0.5 → 0.691 · 1.0 → 0.841 · 1.5 → 0.933 · 2.0 → 0.977 · 2.04 → 0.9793 · 2.1 → 0.9821 · 2.5 → 0.994 · 3.0 → 0.9987

Interpolate linearly between listed points; the error lands in the fourth decimal, below anything the arithmetic in this book can notice. The two awkward entries, $\Phi(2.04)$ and $\Phi(2.1)$, are spelled out because a residual-network result in Chapter 4 turns on them and you will be asked to reproduce it closed-book.

> **WORKED EXAMPLE 1.2**
>
> **Majority vote of five**
>
> Five independent reasoning chains each reach the correct answer with probability 0.6; the majority answer is taken. What is the probability the majority is correct? Here $n = 5$ is too small for the normal approximation, so we sum the binomial directly for $k \geq 3$.
>
> $P(3) = \binom{5}{3}(0.6)^3(0.4)^2 = 10 \times 0.216 \times 0.16 = 0.3456$. $P(4) = \binom{5}{4}(0.6)^4(0.4) = 5 \times 0.1296 \times 0.4 = 0.2592$. $P(5) = (0.6)^5 = 0.0778$. The sum is 0.683. Sampling five chains and voting lifts a 60% chain to a 68% answer — and Chapter 15 shows this only ever helps when the single-chain probability already exceeds one half.

DRILL 1.3

Five independent chains are each correct with probability 0.7. What is the probability that the majority of the five is correct?

Show answer

$P(3) = 10(0.343)(0.09) = 0.3087$; $P(4) = 5(0.2401)(0.3) = 0.3602$; $P(5) = 0.16807$. Sum $= 0.837$.

1.4 Constrained minimization by substitution

The most valuable single calculation in this book is this: minimize a quantity that depends on two variables, when those variables are tied together by a constraint. The compute-optimal training recipe of Chapter 7 — arguably the most consequential result in modern machine learning — is exactly this calculation, and so are several smaller optimizations along the way. The method is humble:

use the constraint to eliminate one variable, then minimize an ordinary one-variable function by setting its derivative to zero.

Concretely, suppose you must minimize $f(N, D)$ subject to a constraint of the form $ND = C$ for a fixed budget C. Solve the constraint for one variable, $D = C/N$, substitute to obtain a function of N alone, differentiate, and set the derivative to zero. The value of N you find, together with $D = C/N$, is the constrained optimum. The subtlety that trips people — and it is worth previewing now so it is not a surprise in Chapter 7 — is that the balance condition at the optimum usually involves the *exponents* of the terms, not merely the terms themselves. Substitution makes that fall out automatically, which is exactly why we prefer it to guessing.

> **WORKED EXAMPLE 1.3**
>
> **A constrained minimum, start to finish**
>
> Minimize $f = N^{-1} + 4D^{-1}$ subject to $ND = 100$.
>
> Substitute $D = 100/N$: $f(N) = N^{-1} + 4 \cdot \frac{N}{100} = \frac{1}{N} + \frac{N}{25}$. Differentiate and set to zero: $f'(N) = -N^{-2} + \frac{1}{25} = 0$, so $N^2 = 25$, $N = 5$. Then $D = 100/5 = 20$, and $f = 0.2 + 0.2 = 0.4$. Notice that the two terms came out *equal* at the optimum — a recurring signature you can often use to check your work, though in Chapter 7 the equal quantities will be the exponent-weighted terms rather than the bare ones.

The same reflex, in a slightly different guise, handles stability questions. When an iterative process multiplies an error by a fixed factor each step, it converges precisely when the magnitude of that factor is below one. Reading off the condition "$|1 - \eta\lambda| < 1$" and solving it for the allowed range of η is the opening move of Chapter 3. It is the same comfort with a one-variable inequality that substitution demands, so we practice it here.

DRILL 1.4

(a) Minimize $2N^{-1} + D^{-1}$ subject to $ND = 50$. (b) For which positive step sizes η does the iteration $z \leftarrow (1 - \eta\lambda)z$ drive z to zero, given $\lambda > 0$?

Show answers

(a) $D = 50/N$ gives $f = 2/N + N/50$; $f' = -2/N^2 + 1/50 = 0 \Rightarrow N^2 = 100$, $N = 10$, $D = 5$, $f = 0.2 + 0.2 = 0.4$. (b) Need $|1 - \eta\lambda| < 1$, i.e. $0 < \eta\lambda < 2$, so $0 < \eta < 2/\lambda$.

1.5 Geometric series

An effect that repeats at every future step, discounted by a constant factor each time, sums to a geometric series. Two closed forms cover every use in this book. Starting from the present step, $\sum_{t \geq 0} \gamma^t = \frac{1}{1-\gamma}$ for $|\gamma| < 1$; starting from the next step, $\sum_{t \geq 1} \gamma^t = \frac{\gamma}{1-\gamma}$. That is all. Chapter 10 uses the second form to weigh a stream of future rewards against a single reward now, and to derive the threshold at which an agent would rather seize a small perpetual reward than accept a large one-off — the arithmetic behind a famous cautionary tale about misaligned incentives.

WORKED EXAMPLE 1.4

Perpetual trickle versus one-time prize

An agent values future rewards with discount factor $\gamma = 0.95$ per step. Option A pays 1 unit now and nothing after. Option B pays r units at every step from the next onward. For which r does the agent prefer B?

Option B is worth $\sum_{t \geq 1} \gamma^t r = \frac{\gamma r}{1-\gamma} = \frac{0.95\, r}{0.05} = 19r$. This exceeds Option A's value of 1 when $r > 1/19 \approx 0.053$. A perpetual reward of barely five percent of the one-time prize is already preferable — a small fact with large consequences in Chapter 10.

DRILL 1.5

With discount factor $\gamma = 0.9$, what perpetual per-step reward r (paid from the next step onward) is worth exactly 2 units of present reward?

Show answer

Value $= \frac{0.9\,r}{0.1} = 9r = 2 \Rightarrow r = 2/9 \approx 0.22$.

1.6 Counting in bits

The final reflex is to measure information in bits, because Chapter 10 — and, quietly, much of the rest of the book — treats learning as compression. Three facts carry the load. A choice among M equally likely options costs $\log_2 M$ bits to specify. A set of code words with lengths ℓ_i can be made unambiguously decodable if and only if $\sum_i 2^{-\ell_i} \le 1$ — the Kraft inequality — which we will read in reverse as a rule that assigns a probability $2^{-\ell}$ to a description of length ℓ. And the number of ways to choose half of $2n$ items, $\binom{2n}{n}$, has a logarithm close to $2n - \frac{1}{2}\log_2(\pi n)$; you will be given this approximation whenever it is needed, but you should know what it says — that a balanced binary pattern is very nearly incompressible.

The reason to install this now, rather than in Chapter 10, is that the interpretation matters more than the arithmetic. When a model assigns probabilities to the next token, the number of bits it would take to encode the true token is $-\log_2$ of the assigned probability; the average of that quantity is the cross-entropy; and a lower cross-entropy is literally a shorter encoding of the data. “A better predictor is a better compressor” is not a metaphor in this book — it is an identity you will compute.

WORKED EXAMPLE 1.5

A probability from a code length

Two descriptions remain consistent with some data: one is 3 bits long, the other 5 bits. Reading the Kraft inequality as a prior, we assign each a weight $2^{-\ell}$: the first gets $2^{-3} = 1/8$, the second $2^{-5} = 1/32$. Normalizing, the shorter description holds $\frac{1/8}{1/8+1/32} = \frac{4}{5} = 0.8$ of the belief. Every additional bit of length halves a description's share — which, as Chapter 10 shows, is Occam's razor stated as a theorem rather than a preference.

DRILL 1.6

(a) How many bits are needed to single out one option from 1000 equally likely ones? (b) Three consistent descriptions have lengths 2, 4, and 4 bits. Under the $2^{-\ell}$ prior, what share of belief does the shortest hold?

Show answers

(a) $\log_2 1000 = \ln 1000 / \ln 2 = 6.908/0.693 \approx 9.97$ bits — just under 10. (b) Weights $2^{-2} = 1/4$, $2^{-4} = 1/16$, $2^{-4} = 1/16$; total $= 4/16 + 1/16 + 1/16 = 6/16$. Shortest share $= (4/16)/(6/16) = 2/3$.

1.7 Turning a conditional around

Two probabilities that look almost the same are not the same, and the gap between them is where a great deal of confident wrong reasoning about models lives. The chance that a leaked question is answered correctly is one number. The chance that a correct answer came from a leaked question is another. Getting from the first to the second is one line of algebra.

Write $P(H \mid E)$ for the probability of a hypothesis H given evidence E. **Bayes' rule** turns one conditional into the other:

$$P(H \mid E) = \frac{P(E \mid H)\, P(H)}{P(E)},$$

and the denominator is just the total probability of the evidence, assembled from every way it could arise: $P(E) = \sum_i P(E \mid H_i) P(H_i)$ over a set of hypotheses

that covers all the cases. In this book the sum almost always has two terms, and the whole calculation fits on one line.

Before working an example, notice what the formula says about magnitudes, because the surprise it produces is the reason the reflex is worth installing. If a hypothesis is rare, then even evidence that follows from it almost certainly can leave it unlikely — the small $P(H)$ on top is not rescued by a large $P(E \mid H)$. Most people's instinct runs the other way, and reads strong evidence as a strong conclusion regardless of how rare the cause was to begin with.

WORKED EXAMPLE 1.6

Which successes were memory

A quarter of a test set was seen during training and those items are always answered correctly; on the remaining three quarters the model answers correctly with probability 0.6. A question is answered correctly. What is the probability it was one of the seen ones?

Take H to be "this item was seen", so $P(H) = 0.25$ and $P(E \mid H) = 1$. For an unseen item, $P(E \mid \text{not } H) = 0.6$. Assemble the denominator first:

$$P(E) = 1 \times 0.25 + 0.6 \times 0.75 = 0.25 + 0.45 = 0.70.$$

Then Bayes' rule gives $P(H \mid E) = (1 \times 0.25)/0.70 = 0.357$. Better than a third of this model's correct answers are recall rather than skill — and note that the observed accuracy, 0.70, fell out of the denominator on the way. Chapter 9 runs this calculation backwards, inferring the quarter from the seventy percent, and the algebra is the same three symbols.

DRILL 1.7

(a) One tenth of a benchmark leaked and is always answered correctly; the clean accuracy is 0.5. Give the observed accuracy, and the probability that a correct answer was leaked. (b) A rare failure mode occurs in one run in a thousand. A detector fires on every genuine occurrence and also on 2% of healthy runs. The detector fires. What is the probability the failure is real?

Show answers

(a) $P(E) = 0.1 + 0.5(0.9) = 0.55$; $P(H \mid E) = 0.1/0.55 = 0.18$. (b) $P(E) = 1(0.001) + 0.02(0.999) = 0.001 + 0.020 = 0.021$, so $P(H \mid E) = 0.001/0.021 = 0.048$. A detector that never misses still leaves you wrong nineteen times in twenty when it fires, because the thing it looks for is rare. This is the magnitude surprise above, in its usual costume.

1.8 Counting operations

The last reflex is the cheapest to state and the most used in Part III: knowing what a computation costs before you run it. Everything expensive in this book is a matrix product, and a matrix product has exactly one cost formula.

The **matrix-multiply primitive** is the whole of it. Multiplying an $a \times b$ matrix by a $b \times c$ matrix produces ac entries, each a sum of b products, so the cost is

$$a\,b\,c \quad \text{multiply–accumulate operations,}$$

where a *multiply–accumulate* is one multiplication and one addition taken together, as the hardware performs it. Count these, not multiplications and additions separately, and the arithmetic in later chapters will match the figures the field quotes. When one of the three dimensions is 1 — a matrix meeting a single vector — the formula still holds and the cost collapses to ab, a fact Chapter 11 builds an entire economics on.

One corollary carries more weight than the rule itself. Every weight in a matrix is touched exactly once per input the matrix processes, so **a layer of P weights costs about P multiply–accumulates per input**. Cost per input is therefore a

statement about parameter count, and nothing else — which is why so much of the field's arithmetic can be done knowing only how big a model is.

WORKED EXAMPLE 1.7

The cost of one feed-forward block

A layer takes a vector of width 512, maps it up to 2048, and maps it back down to 512. What does it cost at one position, and how does that compare with its parameter count?

Up: a 1×512 vector against a 512×2048 matrix, so $1 \times 512 \times 2048 = 1{,}048{,}576$ multiply–accumulates. Down: $1 \times 2048 \times 512$, the same again. The block costs $2{,}097{,}152$ operations per position, about 2.1 million.

Its weights number $512 \times 2048 + 2048 \times 512 = 2{,}097{,}152$ — the same figure, as the corollary promised. Write the width as d and the middle as $4d$ and this is $8d^2$ either way, which is the expression Chapter 6 balances against the cost of attention.

DRILL 1.8

(a) Give the multiply–accumulate cost of multiplying a 128×512 matrix by a 512×64 matrix. (b) A model holds 7×10^9 weights. Roughly how many multiply–accumulates does it perform to process one input? (c) Two square matrices of side n are multiplied. How does the cost grow when n doubles?

Show answers

(a) $128 \times 512 \times 64 = 4{,}194{,}304$, about 4.2 million. (b) About 7×10^9, one per weight. (c) The cost is n^3, so doubling n multiplies it by eight.

1.9 What the chapter bought

That is the whole toolkit. Logarithms turn every power law into a straight line and every compounding factor into a step count. Variance algebra tells you how independent quantities accumulate — the single most reused fact in the book.

The binomial and its normal approximation handle anything that can go one of two ways. Constrained minimization by substitution is the engine behind the field's most consequential result. Geometric series price anything that repeats forever at a discount. Counting in bits is the currency in which prediction, compression, and simplicity all turn out to be the same quantity. Turning a conditional around separates what a number says from what it is usually taken to say. And counting operations converts a description of a model into a bill.

You will not meet these again as objects of study. From here they are furniture, assumed silently in every derivation, and the hours spent making them automatic now are repaid many times over. What Chapter 2 adds is not more mathematics but the objects the mathematics will be applied to — a layer, a gradient step, a softmax, a token — each written down once, precisely, so that no later derivation has to describe its own subject in words.

Exercises

Work these on paper. The A set drills the reflexes; the B set combines them the way later chapters will; the C challenge points forward. Solutions to B and C are in Appendix D.

A · DRILLS

1. Solve $0.98^{t} = 0.5$ for t.
2. A quantity obeys $y \propto x^{\beta}$, and a hundredfold increase in x halves y. Find β.
3. A neuron sums 1024 unit-variance inputs with weights of variance σ^2. What σ keeps the output variance at 1? What is the output variance if $\sigma = 0.02$ is used instead?
4. Seven independent trials each succeed with probability 0.5. Give the mean and standard deviation of the number of successes.
5. Minimize $3N^{-1} + D^{-1}$ subject to $ND = 48$.
6. With $\gamma = 0.99$, what is the present value of receiving 1 unit at every future step starting next step?
7. How many bits specify one microstate among $\binom{10}{5} = 252$ equally likely ones?

8. A fifth of a test set leaked and is always answered correctly; clean accuracy is 0.4. Give the observed accuracy and the probability that a correct answer was leaked.

9. Give the multiply–accumulate cost of a 256×1024 matrix against a 1024×256 one, and state its parameter count.

B · PROBLEMS

1. **Compounding two ways.** A signal is multiplied by 0.9 at each of 100 steps in one design, and by 0.99 at each of 100 steps in another. Give the surviving fraction in each case, and comment on which design preserves memory. (Preview of Chapter 5.)

2. **Square-root drift.** A fixed output coordinate receives, from each of L layers, an independent zero-mean perturbation of variance 0.01. Write the standard deviation of the accumulated perturbation as a function of L, and evaluate it at $L = 100$. (Preview of Chapter 4.)

3. **Exponent-weighted balance.** Minimize $f = AN^{-a} + BD^{-b}$ subject to $ND = C$, by substitution. Show that at the optimum $aAN^{-a} = bBD^{-b}$, and explain in one sentence why the bare terms AN^{-a} and BD^{-b} are *not* equal unless $a = b$. (This is the engine of Chapter 7.)

4. **A leaked benchmark.** A fraction c of test items were seen in training and are always answered correctly; on the remaining clean items the true accuracy is 0.6. The observed accuracy is 0.7. Solve for c. Then, given that an item was answered correctly, find the probability it was one of the leaked items. (Preview of Chapter 9.)

C · CHALLENGE

1. **The halving cost of scale.** A loss term falls as $L \propto N^{-\alpha}$ with $\alpha = 0.076$. Show that halving this term requires multiplying N by $2^{1/\alpha}$, and evaluate the factor. Then, separately, suppose data must scale as $D \propto N^{0.74}$; if N is raised by a factor of 1000, by what factor must D grow? Comment on what these two numbers together say about the cost of progress by scale alone. (You will meet both inside Chapter 7.)

GATE 1 · PASS BEFORE CHAPTER 2

Closed book, fifty minutes. Attempt all nine A-set exercises and B-1 through B-4. You pass at eight of thirteen fully correct. If you fall short, the fix is never to reread — it is to redo the drills in the section whose reflex failed, then retake the gate. These eight reflexes do not appear again as objects of study; from here they are simply assumed, and every hour spent making them automatic now is repaid tenfold in the chapters ahead.

READINGS FOR CHAPTER 1

Prince, *Understanding Deep Learning*

Free and rigorous. Chapter 1 and the probability appendix cover everything above at more length; keep it as the reference to consult whenever a later capsule moves too fast.

Sanderson, *Neural Networks* (3Blue1Brown)

Watch before Chapter 3 if you have never seen a neural network. Visual intuition for what the later algebra is about.

Khan Academy — Random variables and the binomial distribution

Only if Sections 1.2 and 1.3 felt shaky. Work the exercises until the mean and variance of a binomial are automatic.

Zhang et al., *Dive into Deep Learning*

A code-forward alternative reference for the whole book; its preliminaries chapter covers this toolkit alongside the notation you will meet later.

The Objects

Seven things this book computes with, written down exactly once. Not what they are for — only what they are.

You cannot derive a result about a thing you have not written down. What follows is the shortest possible statement of every object the rest of this book manipulates: a network, a gradient step, a softmax, a Jacobian product, a token, a divergence, a normalizer.

This is a reference chapter and it is deliberately flat. There are no surprises in it and nothing here is argued for — the arguing starts in Chapter 3, and every one of these objects earns its place there. Read it once at normal speed, then use it the way you use a table of integrals: when a symbol in a later chapter stops meaning something, come back, and go away again.

One warning about what this chapter is not. It does not explain why neural networks work, what problems they solve, or how anyone came to write them down in this form. Those are excellent questions and this is the wrong book for them; the readings at the end are the right ones. The book you are holding assumes you have seen a network before and asks a different question — what does it cost, and what does the arithmetic force.

2.1 A network, a loss, and a step

A **layer** takes a vector x of width m, forms weighted sums of its entries, adds a constant to each, and passes the result through a fixed nonlinear function ϕ applied entrywise:

$$y = \phi(Wx + b), \qquad W \in \mathbb{R}^{n \times m},\ b \in \mathbb{R}^{n}.$$

W holds the **weights** and b the **biases**; together they are the layer's **parameters**, and Chapter 1's counting rule says the layer costs about nm multiply–accumulates per input. The entries of y are the layer's **activations**. A **network**

stacks such layers, each consuming the previous one's output; running an input through the stack is the **forward pass**. Write θ for all the parameters of all the layers collected into one vector, and N for how many numbers that is — the quantity Chapter 7 spends its entire budget on.

The commonest nonlinearity in this book is the **rectifier**, $\phi(z) = \max(0, z)$: zero for negative input, the identity for positive. It contributes no gradient where it is flat, a fact Chapter 3 turns into a practical warning about biases.

A **loss** $L(\theta)$ is a single number measuring how badly the network does on data, and **training** means making it smaller by changing θ. The only search procedure that scales is **gradient descent**: compute the gradient $g = \nabla_\theta L$, which points in the direction of steepest increase, and step against it,

$$\theta \leftarrow \theta - \eta\, g,$$

where the **step size** η (also called the learning rate) is a small positive number. Chapter 3 is about how badly this can go.

Momentum is one modification, and because Chapter 3 derives a result about it that you will be asked to reproduce three times, it is worth having the update itself in front of you. Keep a running average v of recent gradients and step along that instead:

$$v \leftarrow \beta v + g, \qquad \theta \leftarrow \theta - \eta\, v,$$

with β between 0 and 1 — typically 0.9. Where successive gradients agree, v accumulates and the effective step grows; where they alternate in sign, they cancel. That is the whole mechanism, and Chapter 3 prices it.

DRILL 2.1

(a) A layer maps width 1024 to width 4096 with biases. How many parameters, and how many multiply–accumulates per input? (b) In the momentum update with $\beta = 0.9$, a constant gradient g is supplied at every step. What value does v approach?

Show answers

(a) $4096 \times 1024 + 4096 = 4{,}198{,}400$ parameters; about 4.19 million multiply–accumulates, one per weight. (b) $v \to g(1 + \beta + \beta^2 + \cdots) = g/(1-\beta) = 10g$ — Chapter 1's geometric series, and the reason momentum takes effectively larger steps along a consistent direction.

2.2 Logits, softmax, and cross-entropy

When a network must choose among V options it emits a vector of V real numbers called **logits**, written z. Logits are unconstrained: any real values, positive or negative. The **softmax** converts them into a probability distribution by exponentiating and normalizing:

$$\operatorname{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}}.$$

Three properties are used constantly and none is deep. The outputs are positive and sum to one, so this is a distribution. Adding the same constant to every logit changes nothing, because the constant factors out of numerator and denominator alike — softmax sees only differences, which is why Chapter 12's streaming version may subtract a running maximum without approximation. And when one logit exceeds the others by much more than 1, its probability approaches 1 and the rest collapse toward zero; the distribution **saturates**, and a saturated softmax has almost no gradient, since moving the logits barely moves the probabilities. Chapter 6 rescales attention scores for exactly this reason.

With two options the softmax reduces to the **logistic function** of the logit difference,

$$\sigma(u) = \frac{1}{1 + e^{-u}},$$

which recurs throughout the book as a gate value in Chapter 5 and as a preference probability in Chapter 14. Note $\sigma(0) = \frac{1}{2}$, so $-\ln \sigma(0) = \ln 2 = 0.693$ — a number that turns out to be an acceptance test for a piece of code in Chapter 14.

Given a true outcome i and a predicted distribution q, the **cross-entropy loss** is $-\ln q_i$, and its average over a dataset is the loss almost every model in this book minimizes. Chapter 1 already noted that $-\log_2 q_i$ is the number of bits an optimal code spends on outcome i; the loss is therefore a code length, which is the identity Chapter 10 builds a chapter on. **Perplexity** is the same quantity re-expressed as an effective number of equally likely options, 2^{H} with H the cross-entropy in bits.

> **WORKED EXAMPLE 2.1**
>
> **One prediction, priced**
>
> A model with a vocabulary of four sees a context and emits logits $(3.0,\ 1.0,\ 0.5,\ 0.0)$. The token that actually follows is the second. What does that prediction cost, in nats, in bits, and as a perplexity?
>
> Exponentiate: $e^{3} = 20.09$, $e^{1} = 2.718$, $e^{0.5} = 1.649$, $e^{0} = 1$, summing to 25.45. The distribution is $(0.789,\ 0.107,\ 0.065,\ 0.039)$. The true token was assigned $q = 0.107$, so the loss is $-\ln 0.107 = 2.237$ nats, which is $2.237 / 0.693 = 3.23$ bits, and the perplexity is $2^{3.23} = 9.4$.
>
> Read the last number carefully, because it is the one people misread. A perplexity of 9.4 on a vocabulary of 4 does not mean the model was choosing among nine options — there were only four. It means this particular prediction was *worse* than guessing uniformly, which would have cost $\log_2 4 = 2$ bits. Perplexity is an effective branching factor, and nothing stops it exceeding the real one when the model is confidently wrong. That is exactly what happened here: 0.789 of the mass went somewhere else.

DRILL 2.2

(a) Logits are $(2, 0, -1)$. Give the softmax probabilities. (b) Add 5 to every logit and give them again. (c) A model assigns the true token probability 0.25. Give the cross-entropy loss in nats and in bits.

Show answers

(a) $e^2 = 7.389$, $e^0 = 1$, $e^{-1} = 0.368$; total 8.757. Probabilities $(0.844, 0.114, 0.042)$. (b) Identical — the shared factor e^5 cancels. (c) $-\ln 0.25 = 1.386$ nats $= -\log_2 0.25 = 2$ bits.

2.3 Jacobians, and what backpropagation computes

A network is a composition of functions, so its derivatives are governed by the chain rule. When the functions map vectors to vectors, the object playing the role of a derivative is the **Jacobian**: for $y = F(x)$ with $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, the Jacobian J is the $n \times m$ matrix whose (i, j) entry is $\partial y_i / \partial x_j$. It says how each output responds to each input.

The chain rule then reads as a matrix product. For a stack of ℓ transformations, the sensitivity of the last output to the first input is

$$\frac{\partial h_L}{\partial h_0} = J_L \, J_{L-1} \cdots J_1,$$

and this product — a long chain of matrices multiplied together — is the single most consequential object in Part II. If the factors shrink, the product vanishes geometrically with depth; if they grow, it explodes. Chapter 4 fixes the first problem by adding an identity to every factor, and Chapter 5 meets the same product running along time instead of depth.

Backpropagation is the name for evaluating this product efficiently: sweep forward storing activations, then sweep backward multiplying by one Jacobian at a time, accumulating the gradient with respect to each layer's parameters as you pass it. Two facts about it are used later and neither requires the algorithm's details. The backward sweep costs roughly twice the forward sweep, for the reason the next box counts. And the activations from the forward pass must be kept

until the backward sweep reaches them, which is why they occupy memory — the third of the budgets Chapter 13 names.

DERIVATION

Six operations per parameter per token

The two facts above are enough to price training, and the number they produce carries four later chapters. Work in *floating-point operations* rather than multiply–accumulates, since that is the unit accelerators are sold in: one multiply–accumulate is a multiply and an add, so it is **2 FLOPs.**

Forward. By Chapter 1's corollary, each of the N parameters is used in one multiply–accumulate per token. That is N multiply–accumulates, or $2N$ FLOPs per token.

Backward. Each layer must produce two gradients, not one: the gradient with respect to its input, to hand to the layer below, and the gradient with respect to its own weights. Each is a product against the same weight matrix and costs what the forward pass cost. So the backward sweep is $2N$ multiply–accumulates, or $4N$ FLOPs per token — twice the forward pass, which is the origin of that ratio.

Adding them, training costs about $6N$ FLOPs per token, so a run over D tokens costs

$$C \approx 6ND \text{ FLOPs},$$

and serving, which performs the forward pass alone, costs about $2N$ FLOPs per generated token. The 6 and the 2 are not measurements of anything. They are 2 FLOPs per multiply–accumulate times one, two or three passes over the weights, and Chapters 7, 11 and 17 spend their entire budgets in these units.

DRILL 2.4

(a) A 13-billion-parameter model is trained on 2 trillion tokens. Give the training cost in FLOPs. (b) The same model serves 5×10^{12} tokens over its life. Give the serving cost, and its ratio to training. (c) Why is the backward pass twice the forward one rather than the same or three times?

Show answers

(a) $6 \times 1.3 \times 10^{10} \times 2 \times 10^{12} = 1.56 \times 10^{23}$ FLOPs. (b) $2 \times 1.3 \times 10^{10} \times 5 \times 10^{12} = 1.3 \times 10^{23}$, about 0.83 times the training cost — serving a popular model is the same order of expense as building it, which is the whole of Chapter 17. (c) Because two gradients are needed at each layer and only one activation: one gradient flows down to the previous layer, one lands on the weights. A layer at the very bottom needs only the second, which is why 6 is an approximation slightly on the high side.

2.4 Tokens

Language models do not consume characters or words. Text is first cut into **tokens**, drawn from a fixed **vocabulary** of size V, typically tens of thousands of entries: common words are single tokens, rare words split into several pieces, and every possible string is representable because single bytes are in the vocabulary as a fallback. The number of tokens a corpus becomes is what the symbol D counts in Chapter 7, and the **compression ratio** — characters per token, usually around four for English — is what converts between a corpus's size on disk and its size in the units this book budgets in.

A token is turned into a vector by lookup in an **embedding** matrix of shape $V \times d$, where d is the model's width; one row per vocabulary entry. At the other end, a **readout** matrix of shape $d \times V$ turns the final activations back into V logits. When the two matrices are the same array used twice they are called **tied**, which saves Vd parameters; when they are separate they are **untied**. Both tables scale with vocabulary rather than with depth, which is why Chapter 5 finds them occupying nearly half of a small model and why scaling studies count parameters excluding them.

DRILL 2.3

A model has $d = 4096$ and $V = 32{,}000$, with untied embedding and readout. (a) How many parameters sit in the two tables together? (b) A corpus of 40 GB of English text is tokenized at four characters per token; roughly how many tokens is that?

Show answers

(a) $2 \times 32{,}000 \times 4096 = 262$ million. (b) Roughly $4 \times 10^{10} / 4 = 10^{10}$ tokens, ten billion — assuming one byte per character, which for English is close enough.

2.5 Normalization

A **normalization layer** takes a vector, subtracts its mean, divides by its standard deviation, and then applies a learned scale and shift:

$$\mathrm{norm}(x)_i = \gamma_i \, \frac{x_i - \mu}{\sqrt{s^2 + \epsilon}} + \beta_i, \qquad \mu = \frac{1}{d}\sum_{j} x_j, \;\; s^2 = \frac{1}{d}\sum_{j}(x_j - \mu)^2,$$

with ϵ a small constant guarding against division by zero, and γ, β learned vectors of width d. The point is that whatever the layer receives, what it emits has a controlled scale, set by γ rather than by whatever happened upstream. Variants differ in what they average over and in whether they bother to subtract the mean; none of those distinctions matters to any calculation in this book. What matters is *where* such a layer is placed, which Chapter 4 settles by an argument about accumulated variance.

2.6 Divergence between two distributions

Two distributions over the same set can be compared by a single number. The **Kullback–Leibler divergence** from q to p is

$$\mathrm{KL}(p \,\|\, q) = \sum_{i} p_i \ln \frac{p_i}{q_i} \quad \text{nats},$$

the average, under p, of the log-ratio between the two. Read it as a cost: it is the number of extra nats per outcome you pay for encoding data that really follows p using a code built for q. Three properties are all that later chapters use. It is never negative. It is zero exactly when $p = q$. And it is not symmetric — $\mathrm{KL}(p\|q)$ and $\mathrm{KL}(q\|p)$ are different numbers, so the order of the arguments is never decorative.

Chapter 10 evaluates this for two Gaussians and reads the answer as the price of a weight's precision. Chapter 14 uses it as a leash, penalizing a policy for drifting from the one it started as. Both are the same formula with different distributions substituted in.

DRILL 2.4

Let $p = (0.5, 0.5)$ and $q = (0.9, 0.1)$. Compute $\mathrm{KL}(p\|q)$ and $\mathrm{KL}(q\|p)$ in nats, and confirm they differ.

Show answers

$\mathrm{KL}(p\|q) = 0.5\ln(0.5/0.9) + 0.5\ln(0.5/0.1) = 0.5(-0.588) + 0.5(1.609) = 0.511.$
$\mathrm{KL}(q\|p) = 0.9\ln(0.9/0.5) + 0.1\ln(0.1/0.5) = 0.9(0.588) + 0.1(-1.609) = 0.368.$
Different, as promised, and both positive.

2.7 What the chapter bought

Nothing was derived here, and that is the point: these seven objects are now written down, so that later chapters can compute with them instead of around them. A layer is a matrix, a bias and a nonlinearity, and its cost per input is its parameter count. Gradient descent steps against the gradient; momentum steps against a running average of gradients, and that running average is a geometric series. Softmax turns logits into probabilities, sees only their differences, and stops responding when it saturates. A network's sensitivity is a product of Jacobians, and long products are the recurring danger of Part II. Text becomes tokens, and tokens become rows of two large tables that scale with vocabulary rather than depth. Normalization fixes the scale of what leaves a layer. And a divergence prices the gap between two distributions, asymmetrically.

Chapter 3 begins the actual subject, and it begins by taking the humblest object above — the gradient step of Section 2.1 — and asking a question that held the field back for twenty years: not whether a good setting of θ exists, but whether stepping downhill can ever reach it.

Exercises

These check that the objects are in place, nothing more. If they are slow, reread the section rather than pressing on; every later chapter assumes this vocabulary silently. Solutions to B and C are in Appendix D.

A · DRILLS

1. A layer maps width 512 to width 512 with biases. Give its parameter count and its cost per input in multiply–accumulates.
2. Logits are $(1, 1, 1, 1)$. Give the softmax probabilities, and the cross-entropy loss in bits if the first option is correct.
3. Write the momentum update, and give the limiting value of v under a constant gradient g at $\beta = 0.95$.
4. A model has $V = 50{,}000$ and $d = 768$, with tied embedding and readout. How many parameters are in the table?
5. State two properties of $\mathrm{KL}(p \| q)$ that hold for every p and q.

B · PROBLEMS

1. **Where the parameters are.** A model has $d = 1024$, $V = 32{,}000$, untied tables, and twelve identical hidden layers of width 1024 with biases. Give the parameter count of the tables, of the twelve layers, and the fraction of the total that the tables occupy. Then say what happens to that fraction as the number of layers grows, and why scaling studies quote parameter counts excluding the tables.
2. **Saturation, numerically.** Compute the softmax of $(a, 0)$ for $a = 1, 4, 10$. For each, give the probability of the first option and the derivative $\sigma'(a) = \sigma(a)(1 - \sigma(a))$ of that probability with respect to a. State in one sentence what happens to the gradient as the gap widens, and why a network whose logits grow without limit stops learning.

3. **The product of Jacobians.** Suppose every layer in a stack of L has the same Jacobian $J = cI$ for a scalar c. Give the product over L layers, and evaluate its magnitude for $c = 0.9$ and $c = 1.1$ at $L = 50$. Comment on what this says about how narrow the usable range of c is.

C · CHALLENGE

1. **Why the loss is a length.** Cross-entropy is $-\ln q_i$ for the outcome that occurred; the KL divergence is $\sum_i p_i \ln(p_i/q_i)$. Show that the average cross-entropy under the true distribution p equals the entropy of p plus $\mathrm{KL}(p\|q)$. Conclude that minimizing cross-entropy is minimizing a divergence against a floor you cannot remove, and identify what that floor corresponds to in a language model. (You are deriving, in one line, the shape of the loss surface Chapter 7 minimizes and the irreducible term it carries.)

GATE 2 · PASS BEFORE CHAPTER 3

Closed book, twenty minutes. Write down, from memory and with every symbol named: the layer equation, the gradient-descent and momentum updates, the softmax, the cross-entropy loss, the Jacobian chain product, and the KL divergence. Then attempt A-2 and B-2. You pass when all seven are correct and complete — not approximately right. Every one of them appears inside a derivation you will be asked to reproduce later, and a symbol you cannot define is a derivation you cannot check.

READINGS FOR CHAPTER 2

Sanderson, *Neural Networks* (3Blue1Brown)

Episodes 1 and 2, about forty minutes. The only reading in this book that is prerequisite rather than supplementary: watch it before Chapter 3 if Section 2.1 was the first time you had seen a layer written down.

Prince, *Understanding Deep Learning*

Chapters 3 and 4 for layers and networks, Chapter 7 for backpropagation. The reference to consult whenever this chapter's compression is too severe; free, and rigorous where we are brisk.

Karpathy, *Hacker's Guide to Neural Networks*

Chapter 1, “Real-valued Circuits”, and stop at the end of it. Section 2.3 asserts what that chapter derives gate by gate, and doing it once by hand is worth an hour.

Sennrich, Haddow & Birch, *Neural Machine Translation of Rare Words with Subword Units* (2015)

Section 3, three pages, for how a vocabulary is actually built by merging frequent pairs. This is the algorithm Lab 1 asks you to implement.

Olah, *Visual Information Theory*

The Cross-Entropy and Kullback–Leibler Divergence sections, and stop there. Its pictures make the asymmetry of Section 2.6 obvious in a way the formula does not.

PART II

The Classical Era

The Geometry of Training

For two decades the obstacle to deep learning was not representation but optimization. The problem, and its cure, are a story about the shape of a valley.

Picture a narrow valley with steep walls. You are standing on one slope, blindfolded, allowed only to feel which way the ground falls and step that way. You will bounce from wall to wall and creep along the floor. That, almost exactly, is what training a deep network looked like for twenty years, and understanding why requires nothing more than the shape of the valley.

It is tempting to assume the difficulty is one of expressiveness — that among all possible settings of the weights, the rare good one is hard to find. For a long time the opposite was true. Good settings are plentiful. The trouble was that gradient descent, the only search procedure that scales, could not travel to them. It would take a step, overshoot in one direction, crawl in another, and stall. Naming that temptation is worth a sentence, because believing it sends you hunting for the wrong fix — more capacity, more regularization — when the problem was never capacity at all.

3.1 The loss surface as a bowl

Near a minimum, any smooth loss looks like a bowl. Along some directions the walls are steep; along others they are shallow. If we align our coordinates with the natural axes of the bowl, the loss separates into independent one-dimensional pieces, each a simple parabola with its own steepness. A convenient model of a two-dimensional slice is

$$L(x, y) = \tfrac{1}{2}\left(\lambda_1 x^2 + \lambda_2 y^2\right),$$

where λ_1 and λ_2 are the steepnesses — the curvatures — along the two axes. If λ_1 is much larger than λ_2, the bowl is a long narrow trough: steep across, shallow

along. The number that captures this asymmetry is the ratio of the largest curvature to the smallest,

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}},$$

called the *condition number*. A well-shaped bowl has κ near 1; a pathological valley has κ in the hundreds or thousands. Everything that follows is a statement about how badly a large κ hurts, and what can be done about it.

3.2 Why the step size is trapped

Gradient descent updates each coordinate by subtracting the step size η times the gradient. For our model surface the gradient along x is $\lambda_1 x$, so the update is $x \leftarrow x - \eta\lambda_1 x = (1 - \eta\lambda_1)x$, and likewise for y. Each coordinate is simply multiplied, every step, by a fixed factor. From Chapter 1's stability reflex we know such an iteration converges exactly when the magnitude of the factor is below one.

Before deriving it, guess. If one direction of the bowl is a hundred times steeper than another, how much does that slow you down — a little, or a lot? Most people's instinct is that it costs you a constant factor, some fixed tax for an awkward shape. The instinct is wrong, and wrong in an important way: the cost is proportional to the ratio itself, so a hundredfold asymmetry costs you roughly a hundredfold in steps. Here is where that comes from.

DERIVATION 3.1

The stable range of step sizes

The coordinate x obeys $x \leftarrow (1 - \eta\lambda_1)x$, which converges to zero if and only if $|1 - \eta\lambda_1| < 1$, i.e. $0 < \eta\lambda_1 < 2$. The same holds for y with λ_2. Both must hold at once, and since $\lambda_1 = \lambda_{\max}$ is the larger curvature, it is the binding one:

$$0 < \eta < \frac{2}{\lambda_{\max}}.$$

This is the whole trap in one line. The steep direction sets a ceiling on the step size. Any larger step and the iteration diverges along that axis, flinging the weights outward. So the step size is *hostage to the steepest curvature in the entire problem* — even though most directions would happily accept a far larger step.

Now watch what that ceiling does to the shallow direction. With η pinned just under $2/\lambda_{\max}$, the shallow coordinate contracts by a factor $|1 - \eta\lambda_{\min}|$, which is close to $1 - 2\lambda_{\min}/\lambda_{\max} = 1 - 2/\kappa$. When κ is large this factor is barely below one: the shallow direction crawls. The steep direction, meanwhile, is oscillating back and forth across the valley, using up the entire step budget just to stay stable. The network is not stuck because it cannot represent the answer; it is stuck because the geometry forces it to inch down the valley floor while ricocheting between the walls.

3.3 Counting the cost of curvature

We can make the slowness precise. The error along the shallow direction contracts by a fixed factor each step, so reaching a target reduction is a logarithm problem of exactly the kind Chapter 1 drilled. It is a standard result — we take it as given here, the way a physics problem gives you a formula for terminal velocity — that gradient descent on such a surface, at the best possible fixed step size, contracts the error each step by

$$\rho_{\mathrm{GD}} = \frac{\kappa - 1}{\kappa + 1},$$

while the same surface, optimized with a well-tuned *momentum* term that lets the shallow direction accumulate speed, contracts by

$$\rho_{\text{mom}} = \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}.$$

The number of steps to shrink the error by a factor F is $t = \ln F / \ln(1/\rho)$. The two rates above look similar, but the square root inside the momentum rate is decisive, as the worked example shows.

> **WORKED EXAMPLE 3.1**
>
> **Three hundred and forty-five steps, or thirty-four**
>
> Take a valley with $\kappa = 100$ — steep direction a hundred times sharper than the shallow one — and ask how many steps each method needs to reduce the error a thousandfold.
>
> For plain gradient descent, $\rho_{\text{GD}} = 99/101 = 0.9802$, so $t = \ln(1000)/\ln(1/0.9802) = 6.908/0.02 = 345$ steps. For momentum, $\rho_{\text{mom}} = 9/11 = 0.8182$, so $t = 6.908/\ln(1/0.8182) = 6.908/0.2007 = 34$ steps. The ratio is almost exactly ten — which is $\sqrt{\kappa} = \sqrt{100}$.
>
> This is the quantitative content of the phrase “momentum glides along the valley floor.” Plain descent pays a cost proportional to κ; momentum pays a cost proportional to $\sqrt{\kappa}$. On a valley conditioned at 10^4, that is the difference between ten thousand steps and one hundred.

DERIVATION 3.2 · PROTOCOL

Why the speedup is the square root

The two rates differ only by a square root inside them, and it is worth seeing exactly how that becomes a square root in the step count. Expand each for large κ.

For plain descent, $\rho_{\mathrm{GD}} = \frac{\kappa-1}{\kappa+1} \approx 1 - \frac{2}{\kappa}$. Since $\ln(1-x) \approx -x$ for small x,

$$\ln \frac{1}{\rho_{\mathrm{GD}}} \approx \frac{2}{\kappa}.$$

For momentum, $\rho_{\mathrm{mom}} = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} \approx 1 - \frac{2}{\sqrt{\kappa}}$, so $\ln(1/\rho_{\mathrm{mom}}) \approx 2/\sqrt{\kappa}$.

The step count to shrink the error by a factor F is $t = \ln F / \ln(1/\rho)$, so the ratio of counts is the inverse ratio of those logarithms:

$$\frac{t_{\mathrm{GD}}}{t_{\mathrm{mom}}} \approx \frac{2/\sqrt{\kappa}}{2/\kappa} = \sqrt{\kappa}.$$

The whole effect is that momentum's rate carries $\sqrt{\kappa}$ where plain descent carries κ, and the logarithm turns a difference of rates into a ratio of counts. At $\kappa = 100$ that is the factor of ten of Worked Example 3.1; at $\kappa = 10^4$ it is a hundred.

DRILL 3.1

A valley has $\kappa = 400$. Estimate the ratio of plain-gradient-descent steps to momentum steps for the same error reduction, without computing either count in full.

Show answer

The ratio of step counts is $\ln(1/\rho_{\mathrm{GD}})$ versus $\ln(1/\rho_{\mathrm{mom}})$ inverted, which for large κ tends to $\sqrt{\kappa}$. Here $\sqrt{400} = 20$. (Explicitly: $\rho_{\mathrm{GD}} = 399/401 \Rightarrow \ln(1/\rho) = 0.005$; $\rho_{\mathrm{mom}} = 19/21 \Rightarrow \ln(1/\rho) = 0.100$; ratio $0.100/0.005 = 20$.)

TRAP · T1

The deeper failure this chapter describes is a *training* failure, not overfitting. When a plain deep network does worse than a shallow one, the instinct is to suspect it has memorized and failed to generalize. But the error that rises is the training error itself: the optimizer never reached a good setting at all. Confusing an optimization failure with a generalization failure sends you reaching for the wrong remedy — regularization instead of better conditioning — and it is one of the most common misreadings of the early deep-learning literature.

3.4 Starting in the right place: the statistics of initialization

Conditioning governs how the optimizer moves once it starts; initialization governs whether it can start at all. Two separate requirements fall directly out of Chapter 1's variance algebra, and both were, historically, hard-won.

The first is scale. Recall the fan-in derivation: a unit summing m unit-variance inputs with weights of variance σ^2 produces an output of variance $m\sigma^2$. If σ^2 is chosen carelessly, signal either explodes or vanishes as it crosses each layer, and a deep stack multiplies that error many times over. Preserving the scale of activations from layer to layer requires

$$\sigma^2 = \frac{1}{m},$$

the principle behind every sensible initialization scheme. The precise constant is adjusted for the nonlinearity in use, but the $1/m$ skeleton is always there.

WORKED EXAMPLE 3.2

A too-small initialization

A layer has fan-in $m = 4096$. The variance-preserving choice is $\sigma^2 = 1/4096$, i.e. $\sigma \approx 0.0156$. Suppose instead someone initializes with a flat $\sigma = 0.01$. The output variance is then $m\sigma^2 = 4096 \times 10^{-4} = 0.41$. Each layer shrinks the signal to under half its variance; across a dozen layers the activations have all but vanished, and with them the gradients. The network trains glacially or not at all — not because anything is conceptually wrong, but because a single variance was set a factor of 1.5 too small and compounded.

The second requirement is asymmetry. Suppose every weight in a layer were initialized to the same constant. Then every unit in that layer computes the identical function of the inputs, receives the identical gradient, and updates identically — forever. The layer, however wide, collapses to a single effective unit, and no amount of training breaks the tie. This is why initialization must be *random*: randomness is what breaks the symmetry between units and lets them specialize. It is a one-line argument, but it is the reason a fundamental-looking operation — "set the weights to a sensible constant" — is quietly fatal.

DERIVATION 3.3

Symmetry never breaks itself

Consider two units in a layer, with incoming weight vectors w_1 and w_2, fed the same inputs. If $w_1 = w_2$ at initialization, then their outputs are equal at every input, so the loss depends on them only through their sum. The gradient of the loss with respect to w_1 therefore equals the gradient with respect to w_2, and one gradient step keeps $w_1 = w_2$. By induction the two are equal for all time. The only way to give them distinct gradients is to start them at distinct values — hence random initialization.

A small practical corollary closes the chapter, and it too is pure Chapter-1 reasoning. Units with a rectified activation — zero for negative inputs, linear for positive — contribute no gradient when their input is negative, and a unit that begins with a negative bias may sit dead from the start. Nudging the initial bias

slightly positive shifts the input distribution rightward, so more units begin in their active region and receive gradient. The same variance-and-mean bookkeeping that governs the weights governs this choice; it is not a trick so much as an accounting consequence.

DRILL 3.2

A layer has fan-in $m = 1024$. (a) What σ preserves unit output variance? (b) If instead $\sigma = 0.05$ is used, what is the output variance, and does the signal grow or shrink across the layer?

Show answers

(a) $\sigma = 1/\sqrt{1024} = 1/32 \approx 0.031$. (b) Output variance $= 1024 \times 0.0025 = 2.56$; since this exceeds 1, the signal *grows* across the layer, and would explode across a deep stack.

3.5 What the chapter bought

Everything that follows rests on what this chapter established, and all of it came from Chapter 1's tools. The stable step size is capped by the steepest curvature, so a badly conditioned problem is slow no matter how patient you are. Momentum converts that κ-sized penalty into a $\sqrt{\kappa}$-sized one — the first appearance of a theme, adaptivity to geometry, that runs through the rest of the field. And initialization is a variance-and-symmetry problem whose answers are forced, not chosen. Chapter 4 takes up the complementary question: if optimization is this sensitive to the shape of the surface, can we *build* networks whose surfaces are easier to descend? The answer — residual connections — is where architecture begins to do the optimizer's job for it.

Exercises

A · DRILLS

1. A valley has curvatures $\lambda_{\max} = 50$, $\lambda_{\min} = 2$. Give κ and the largest stable step size.

2. For $\kappa = 100$, compute ρ_{GD} and ρ_{mom} to four decimal places.
3. A layer of fan-in 256 is to preserve activation variance. What weight standard deviation does that require?
4. Explain in two sentences why initializing all weights of a layer to 0.1 is no better than initializing them all to 0.

B · PROBLEMS

1. **The valley, end to end.** For the surface $L(x, y) = \frac{1}{2}(\lambda_1 x^2 + \lambda_2 y^2)$ with $\lambda_1 = 200$, $\lambda_2 = 2$: (a) give the largest stable η; (b) with η set to 90% of that ceiling, give the per-step contraction factor along each axis; (c) estimate how many steps the shallow direction needs to contract a hundredfold, and comment on how many the steep direction "wastes" oscillating.
2. **The square-root promise, at a second condition number.** Reproduce Derivation 3.2 from nothing, then evaluate both step counts explicitly at $\kappa = 2500$ for a thousandfold error reduction, and confirm the ratio is 50. State in one sentence why the approximation $\ln(1 - x) \approx -x$ is safe here.
3. **Compounded initialization error.** A twelve-layer stack, each layer fan-in 500, is initialized with a uniform $\sigma = 1/\sqrt{1000}$ — a factor $\sqrt{2}$ too small. By what factor has the activation variance changed after all twelve layers, relative to a variance-preserving initialization?

C · CHALLENGE

1. **Reconditioning by rescaling.** Suppose a diagonal preconditioner multiplies the gradient along each axis by the reciprocal of that axis's curvature, so the effective update along axis i becomes $x_i \leftarrow (1 - \eta) x_i$. Show that this makes the effective condition number 1, and explain in a sentence what real second-order and adaptive methods are approximating when they do something like this. (This is the conceptual bridge from this chapter to why adaptive optimizers exist.)

GATE 3 · PASS BEFORE CHAPTER 4

Reproduce Derivations 3.1 (the stable step range) and 3.2 (the $\sqrt{\kappa}$ speedup) on blank paper, then attempt B-1 closed book. This is the first of the three derivations carrying the three-separate-days protocol: return to the κ-versus-$\sqrt{\kappa}$ result on two later days and reproduce it cold each time. It is the template for the compute-optimal derivation of Chapter 7, and the fluency you build here is the fluency you will spend there.

READINGS FOR CHAPTER 3

Martens, *Deep Learning via Hessian-Free Optimization* (2010)

Read the first two sections for the pathological-curvature argument in its original setting — the paper that made conditioning central.

Krizhevsky, Sutskever & Hinton, *ImageNet Classification with Deep CNNs* (2012)

The initialization and optimization choices in Sections 3–5 are the practical face of this chapter's theory.

Stanford CS231n — Optimization notes

The standing reference for step sizes, momentum, and adaptive methods; work the update-rule sections against your Derivation 3.1.

Goh, *Why Momentum Really Works* (Distill, 2017)

The "Dynamics of Momentum" section and the convergence-rate figure under it — exactly the $\sqrt{\kappa}$ result of B-2, made interactive. Move the sliders until the geometry is intuitive, then close it and reproduce the result cold.

The Architecture of Depth

If optimization is so sensitive to the shape of the loss surface, perhaps we can build networks whose surfaces are easier to descend. That idea, made concrete, is the residual connection.

Suppose you have a network that works at twenty layers and you want one that works at a hundred. You add eighty more layers of the same kind and train it the same way. It gets *worse* — and not on held-out data, where you might blame overfitting, but on the training set itself. Something about depth is defeating the optimizer. The fix, when it came, was a single addition sign.

We begin with the humblest ingredient, the convolution, because its costs are pure counting and because those counts recur every time we compare two designs. Then we turn to the central innovation of the era, the residual connection, and find that its single line of algebra conceals three separate theorems. We close with dilation, a way to see far across an image without paying for the privilege.

4.1 The arithmetic of convolution

A convolutional layer slides a small kernel across a grid, computing at each position a weighted sum of a local patch. Three quantities describe its cost, and all three are bookkeeping. First, the spatial size of the output: for an input of side n, a kernel of side k, padding p, and stride s,

$$n_{\text{out}} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1.$$

The floor is not a formality — it is where a great many arithmetic slips are born, because a fractional result must be truncated, not rounded. Second, the parameter count. A layer mapping C_{in} input channels to C_{out} output channels,

with a kernel of side k, holds $k^2 C_{\text{in}}$ weights per output channel plus one bias each:

$$\text{params} = (k^2 C_{\text{in}} + 1)\, C_{\text{out}}.$$

Forget the $+1$ and you will be a bias term short on every layer — a small error that the exercises will punish. Third, the arithmetic cost: producing one output pixel takes $k^2 C_{\text{in}} C_{\text{out}}$ multiply–accumulate operations, and you multiply by the number of output pixels to get the layer's total.

> **WORKED EXAMPLE 4.1**
>
> **Tracing a dimension through the opening layers**
>
> A $227 \times 227 \times 3$ image enters a layer of 96 kernels, each 11×11, stride 4, no padding; the result passes through a 3×3 max-pool at stride 2. Trace the spatial size and count the convolution's parameters.
>
> Convolution output side: $\lfloor (227 - 11)/4 \rfloor + 1 = \lfloor 216/4 \rfloor + 1 = 54 + 1 = 55$. Pooling output side: $\lfloor (55 - 3)/2 \rfloor + 1 = \lfloor 52/2 \rfloor + 1 = 26 + 1 = 27$. So the feature map is 27×27. Parameters in the convolution: $(11^2 \cdot 3 + 1) \cdot 96 = (363 + 1) \cdot 96 = 364 \cdot 96 = 34{,}944$. Drop the bias and you get $34{,}848$ — the classic off-by-a-bias mistake.

DRILL 4.1

A 55×55 feature map passes through two successive 3×3 max-pools, each at stride 2, no padding. What is the final spatial side?

Show answer

First pool: $\lfloor (55 - 3)/2 \rfloor + 1 = 27$. Second: $\lfloor (27 - 3)/2 \rfloor + 1 = 13$. Final side 13.

One design decision recurs so often it deserves its own note: replacing a single large kernel with a stack of small ones. Two stacked 3×3 convolutions see the same 5×5 input region as one 5×5 convolution — the receptive fields match — but the stack holds fewer weights and applies its nonlinearity twice instead of once. For C channels throughout, one 5×5 layer holds $25C^2$ weights; two 3×3 layers hold $2 \times 9C^2 = 18C^2$. The stack is cheaper *and* more expressive, which is why deep stacks of tiny kernels displaced wide ones.

DRILL 4.2

With $C = 256$ channels, how many weights are saved by replacing one 5×5 convolution with two stacked 3×3 convolutions? Give the answer to the nearest thousand.

Show answer

$25C^2 - 18C^2 = 7C^2 = 7 \times 65{,}536 = 458{,}752$, about $459{,}000$.

4.2 The residual connection, and the three theorems inside it

A residual block does not compute a new representation from scratch; it computes a *correction* to its input. Writing h_ℓ for the activations after block ℓ, a residual block sets

$$h_\ell = h_{\ell-1} + F_\ell(h_{\ell-1}),$$

where F_ℓ is the block's learned transformation. The added $h_{\ell-1}$ — the "skip" or "identity" path — looks almost too simple to matter. It matters three separate ways, and untangling them is the heart of this chapter.

First theorem: the gradient has a floor

Consider how a gradient travels backward through the stack. The sensitivity of the output to the input is a product of per-block Jacobians. For a residual block, that Jacobian is $I + J_\ell$, where J_ℓ is the Jacobian of F_ℓ alone. Across L blocks,

$$\frac{\partial h_L}{\partial h_0} = \prod_{\ell=1}^{L} (I + J_\ell).$$

Expand this product and something remarkable appears.

DERIVATION 4.1

The indestructible identity

Multiplying out $\prod_{\ell=1}^{L}(I + J_\ell)$, each factor contributes either its I or its J_ℓ. A term that takes J from exactly k of the factors, and I from the rest, is a product of k Jacobians; there are $\binom{L}{k}$ such terms. The expansion is thus organized by how many blocks a path passes through:

$$\frac{\partial h_L}{\partial h_0} = \underbrace{I}_{k=0} + \underbrace{\sum_{\ell} J_\ell}_{k=1} + \underbrace{\sum_{i<j} J_j J_i}_{k=2} + \cdots$$

Look at the $k = 0$ term: it is the bare identity I, formed by taking I from every factor. It contains none of the J_ℓ, so no amount of shrinking the individual Jacobians can destroy it. However small every F_ℓ becomes, the gradient still contains an undiminished identity path from output to input.

Contrast a plain network, whose Jacobian is the product $\prod_\ell J_\ell$ with no added identities. If those Jacobians are even slightly contractive, their product shrinks geometrically with depth — the vanishing gradient. The residual network's guarantee is exactly the surviving I: a floor beneath the gradient that the plain network lacks.

Second theorem: depth is an ensemble of shallow paths

The expansion above has a second reading. Each term is a *path*: a choice, at each block, of whether to pass through the transformation or skip it. There are 2^L such paths, and a path's "length" is the number of blocks it actually traverses. How long is a typical path? Since each block is independently taken or skipped, the length of a random path is a sum of L independent coin flips — a binomial distribution, exactly the object Chapter 1 prepared.

Ask yourself first what you expect the answer to be. A network with fifty-four residual blocks contains paths of every length from zero to fifty-four. If you had to guess the typical path length, you might reasonably say "most of them are long, since the network is deep." Hold that thought; the arithmetic is about to disagree with it, and the disagreement is the entire reason very deep residual networks train at all.

DERIVATION 4.2

Most paths are short

Model each block as traversed with probability $\frac{1}{2}$. A path's length is then $\text{Binomial}(L, \frac{1}{2})$, with mean $L/2$ and variance $L/4$. For $L = 54$ blocks, the mean length is 27 and the standard deviation is $\sqrt{54/4} = \sqrt{13.5} = 3.674$.

What fraction of paths have length between 20 and 34? Using the normal approximation with a half-unit continuity correction, the standardized endpoints are $(19.5 - 27)/3.674 = -2.04$ and $(34.5 - 27)/3.674 = +2.04$. The enclosed probability is $2\Phi(2.04) - 1 = 2(0.9793) - 1 = 0.959$.

So about 96% of the paths through a 54-block network — a network more than a hundred layers deep — traverse between 20 and 34 blocks. The network behaves less like one very deep chain and more like an *ensemble of many moderately shallow paths*. Depth, in a residual network, is a reservoir of shallow routes, not a single long one; and gradients flow easily precisely because most routes are short.

Third theorem: signals drift, so normalization belongs inside

The third consequence is about scale. Each block adds its correction to the running signal. If those corrections are roughly independent and zero-mean, then by Chapter 1's variance algebra their accumulated effect at any fixed coordinate grows in variance like the number of blocks — standard deviation like $\sqrt{L}$.

WORKED EXAMPLE 4.2

Square-root drift

Suppose each of $L = 100$ blocks adds, at a fixed output coordinate, an independent zero-mean perturbation of variance 0.01. The accumulated variance is $100 \times 0.01 = 1$, so the standard deviation is 1.0 — comparable to the signal itself. Had we used $L = 400$ blocks, the drift would be $\sqrt{4} = 2$. The signal swells as it descends the stack.

This is why normalization layers are placed *inside* the residual branch, rescaling F_ℓ's output before it is added, rather than on the skip path. Keep the identity path clean and controlled, and let the normalization tame the drifting correction — a design that the "pre-activation" residual block makes explicit.

The bottleneck: buying depth cheaply

A last piece of residual arithmetic concerns cost. Very deep networks cannot afford full-width 3×3 convolutions at every block, so the bottleneck block compresses the channels first: a 1×1 convolution reduces C channels to C/c, a 3×3 operates in that narrow space, and a final 1×1 restores the width. Because the expensive 3×3 now runs on $1/c$ of the channels, it costs $1/c^2$ as much.

WORKED EXAMPLE 4.3

Seventeen to one

Compare, at one spatial position with 256 channels, two stacked 3×3 convolutions against a bottleneck with compression $c = 4$ (256 down to 64). The plain pair costs $2 \cdot 9 \cdot 256^2 = 1{,}179{,}648$ multiply–accumulates. The bottleneck costs $(256 \cdot 64) + (9 \cdot 64^2) + (64 \cdot 256) = 16{,}384 + 36{,}864 + 16{,}384 = 69{,}632$. The ratio is 16.9, close to 17 to 1. The narrow middle is where nearly all the saving lives.

DRILL 4.3

At compression $c = 8$ (256 channels down to 32), estimate the bottleneck's MAC ratio against the plain 3×3 pair. (Use the same structure: two 1×1 layers plus one 3×3 in the narrow space, all at 256 outer channels.)

Show answer

Plain pair $= 18C^2$. Bottleneck $= C \cdot (C/8) + 9(C/8)^2 + (C/8) \cdot C = C^2/8 + 9C^2/64 + C^2/8 = C^2(8/64 + 9/64 + 8/64) = 25C^2/64$. Ratio $= 18/(25/64) = 18 \times 64/25 \approx 46$ to 1.

4.3 Dilation: seeing far without paying for it

Some tasks — labeling every pixel of an image, say — need each output to depend on a wide swath of input, yet must not blur away fine detail by pooling. Dilation resolves the tension. A dilated kernel spreads its taps apart, leaving gaps: a 3×3 kernel with dilation r still has nine weights, but they sample a region $(2r + 1)$ wide. Stacking dilated layers grows the reach geometrically while the parameter count grows not at all per layer.

DERIVATION 4.3

Exponential reach for linear cost

Adding a 3×3 layer of dilation r extends the receptive field by $2r$ on each axis: its outer taps reach r beyond the current field on each side. So a stack with dilations $1, 2, 4, \ldots, 2^{n-1}$ has receptive field

$$1 + 2(1 + 2 + 4 + \cdots + 2^{n-1}) = 1 + 2(2^n - 1) = 2^{n+1} - 1.$$

An undilated stack, by contrast, grows its field by 2 per layer, reaching only $2n + 1$ after n layers — linear, not exponential. To cover a 31×31 field, the dilated stack needs $n = 4$ layers ($2^5 - 1 = 31$); the undilated stack needs $n = 15$. Same kernels, same per-layer weights; four layers against fifteen.

The saving against a single wide kernel is even starker. A lone kernel covering a 511×511 field would hold $511^2 = 261{,}121$ weights per channel pair; the dilated stack that reaches the same field ($n = 8$, since $2^9 - 1 = 511$) holds $8 \times 9 = 72$. The ratio is about 3,600 to one. Exponential context, linear parameters — the entire case for dilation in one comparison.

DRILL 4.4

Using 3×3 kernels with dilations $1, 2, 4, \ldots$, find the smallest number of layers whose receptive field reaches at least 1000. Then give the ratio of a single equivalent kernel's weights to the stack's.

Show answer

$2^{n+1} - 1 \geq 1000 \Rightarrow 2^{n+1} \geq 1001 \Rightarrow n + 1 \geq 10 \Rightarrow n = 9$, giving a field of 1023. Single-kernel weights $1023^2 = 1{,}046{,}529$; stack $9 \times 9 = 81$; ratio $\approx 12{,}900$ to one.

TRAP · T5

Dilation is often misremembered as reducing resolution, like pooling. It does the opposite: it widens the receptive field *without* downsampling, which is exactly why it suits dense, pixel-level prediction where pooling would throw away the detail you are trying to recover. And the factorization of one 5×5 into two 3×3 layers is often said to "save computation at the cost of expressiveness" — also backwards. The stack is both cheaper and applies an extra nonlinearity; the expressiveness goes *up*.

4.4 What the chapter bought

Convolution arithmetic is the ruler we will measure every later architecture with; keep the floor, the bias, and the multiply–accumulate count automatic. The residual connection turned out to be three ideas: a gradient floor guaranteed by an indestructible identity term, an ensemble of predominantly shallow paths that makes great depth trainable, and a $\sqrt{L}$ signal drift that dictates where normalization goes. Dilation buys exponential reach for linear cost. Together these are how architecture began to do the optimizer's work — shaping the loss surface, and the flow of gradients across it, so that depth became an asset rather than an obstacle. Chapter 5 turns from space to time, and asks the same question about sequences: what makes a recurrent network trainable, and what pins its memory in place?

Exercises

A · DRILLS

1. A 224×224 input meets a 7×7 kernel, stride 2, padding 3. Give the output side.
2. A convolution maps 64 channels to 128 with 3×3 kernels and biases. Count its parameters.
3. For $L = 20$ residual blocks, how many distinct forward paths exist, and what is the mean path length?

4. A dilated 3×3 stack uses dilations $1, 2, 4, 8$. Give its receptive field.

B · PROBLEMS

1. **The path ensemble at 100 blocks.** For $L = 100$ residual blocks with each block traversed at probability $\frac{1}{2}$, estimate the fraction of paths whose length lies between 40 and 60 inclusive, using the normal approximation with continuity correction.

2. **Where the saving lives.** Derive the general bottleneck-to-plain MAC ratio as a function of the compression factor c, taking the plain design as two 3×3 convolutions at C channels and the bottleneck as 1×1 down, 3×3 narrow, 1×1 up. Evaluate at $c = 2$ and $c = 4$, and say what happens as c grows large.

3. **Reach versus depth.** You must cover a 255×255 receptive field with 3×3 kernels. Give the number of dilated layers required, the number of undilated layers required, and the ratio of a single equivalent kernel's weights to the dilated stack's.

4. **Drift, and where normalization goes.** A 60-block residual stack adds, at one output coordinate, an independent zero-mean perturbation of variance 0.02 per block. Give the accumulated standard deviation. Then suppose normalization is placed on the *skip* path instead of inside the branch, rescaling the running signal by $1/\sqrt{\ell}$ after block ℓ. Using Derivation 4.1, say what that does to the surviving identity term, and why the drift argument does not justify it.

5. **The cost of one design decision.** A layer must map 256 channels to 256 over a 7×7 receptive field at one spatial position. Cost it three ways: a single 7×7 convolution; three stacked 3×3 convolutions; and a bottleneck at $c = 4$ wrapping a single 3×3. Give parameters and multiply–accumulates for each, rank them, and say what each buys besides its number.

C · CHALLENGE

1. **Why identity mappings help twice.** The "pre-activation" residual block moves normalization and nonlinearity *before* the convolutions, so the skip path is a clean, unmodified identity. Argue, using Derivation 4.1, why keeping the skip path free of any transformation strengthens the gradient-floor guarantee; then, using Worked Example 4.2, argue why it also helps control the $\sqrt{L}$ drift. What single principle unifies the two arguments?

GATE 4 · PASS BEFORE CHAPTER 5

Reproduce Derivations 3.1 (the identity term) and 3.2 (the path-length binomial) on blank paper, and trace one full convolution-and-pool dimension chain in under three minutes. Attempt B-1 and B-2 closed book. You pass when the three derivations come without hesitation and the arithmetic carries its biases and floors intact.

READINGS FOR CHAPTER 4

He, Zhang, Ren & Sun, *Deep Residual Learning for Image Recognition* (2015)

The degradation figure in the opening — a 56-layer plain network worse than a 20-layer one on *training* error — is the vanishing gradient Derivation 4.1 cures. Read their ablation tables as experiments about that derivation.

He et al., *Identity Mappings in Deep Residual Networks* (2016)

The shortcut-variant table is the C-challenge argued in full: every way of putting something on the skip path, and what each costs. Compare their reasoning to yours before reading their conclusion.

Veit, Wilber & Belongie, *Residual Networks Behave Like Ensembles of Relatively Shallow Networks* (2016)

Derivation 4.2, measured. Go to the lesion-study figure, where blocks are deleted at test time and the ensemble degrades gracefully; that graceful curve is the path-length binomial you derived.

Yu & Koltun, *Multi-Scale Context Aggregation by Dilated Convolutions* (2015)

The origin of Derivation 4.3. The context-module table lists the dilation schedule term by term; check that its receptive fields match your $2^{n+1} - 1$.

Stanford CS231n — Convolutional Networks

The standing reference for the arithmetic of Section 4.1; work its worked examples against yours.

Memory and Gates

A recurrent network remembers by multiplying. Whatever is multiplied a thousand times either vanishes or explodes — unless something pins the factor to one. That pin is the whole story.

A recurrent network remembers by multiplying. To carry information from step one to step three hundred, it must pass that information through three hundred multiplications, and whatever survives is what the network can recall. Multiply anything by 0.95 three hundred times and you have nothing left. Multiply by 1.0 and you have everything. Almost everything in this chapter follows from that observation.

Learning long-range structure requires a gradient to travel backward across many steps, and backward travel across a recurrence is a *product* of per-step factors. Chapter 4 showed what long products do: they die or explode unless something holds them in check. Here the same problem reappears along the time axis rather than the depth axis, and the fix has the same character. What follows derives the horizon over which memory survives, the single constant that sets it, and a consequence for regularization that catches most people out.

5.1 The gradient horizon

First the object itself, because the whole chapter turns on one of its terms. A gated cell carries a memory vector c_t from step to step. At each step it decides how much of that memory to keep and how much new material to add:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t,$$

where g_t is a candidate update computed from the input and the previous state, i_t is an *input gate* deciding how much of it to admit, f_t is the *forget gate* deciding how much of the old memory to retain, and $\odot$ multiplies entrywise. Both gates are values in $(0, 1)$ produced by a logistic function — Section 2.2 — of a learned

pre-activation, so the cell computes them afresh at every step rather than carrying them as fixed weights.

Now differentiate. Holding the gates fixed, $\partial c_t / \partial c_{t-1} = \mathrm{diag}(f_t)$: the memory's sensitivity to its own past is the forget gate and nothing else. That is the entire reason this architecture exists, and it is why the factor in what follows is not an abstraction but a number the network chooses.

Backpropagation through time multiplies the gradient, at each step, by exactly that factor. Call it f. After t steps the gradient carries f^t. If $f < 1$, the influence decays geometrically; the number of steps over which a gradient survives is a logarithm problem of exactly the kind Chapter 1 drilled.

DERIVATION 5.1

How far back memory reaches

A gradient scaled by f per step falls below a fraction ϵ of its original size after t steps, where $f^t < \epsilon$, i.e.

$$t > \frac{\ln \epsilon}{\ln f}.$$

Both logarithms are negative for $f < 1$ and $\epsilon < 1$, so t is positive. With $f = 0.95$ and $\epsilon = 0.01$: $t > \ln(0.01)/\ln(0.95) = (-4.605)/(-0.0513) = 89.8$, so the gradient is gone after about 90 steps. With $f = 0.5$, the horizon collapses to $\ln(0.01)/\ln(0.5) \approx 6.6$ steps — a network that cannot see past the last handful of tokens.

The critical case is $f = 1$. Then $f^t = 1$ for all t: the gradient neither decays nor grows, and memory reaches arbitrarily far back. A recurrent cell that could hold its per-step factor at exactly one would have, in principle, unbounded memory. This is the target the gated cell was built to hit.

The device that hits it is the *forget gate*. The gated cell maintains a memory that is multiplied each step not by a fixed weight but by a gate value $f \in (0, 1)$ that the network computes. When the network wants to remember, it drives the gate toward one, and the memory is carried forward almost untouched — the "constant-error carousel" that gives the gradient a clear channel across time. When it wants to forget, it drives the gate down. The horizon is no longer a fixed property of the architecture; it is something the network sets, token by token.

5.2 Setting the gate by its bias

A gate is produced by a squashing function of a learned pre-activation: $f = \sigma(z)$, where σ is the logistic function and z includes a bias term. At the start of training, before the input-dependent part has learned anything, the gate sits at $\sigma(b)$ for bias b. If we want the cell to *default* to remembering — a sensible prior for tasks with long dependencies — we should choose b so that $\sigma(b)$ is close to one.

DERIVATION 5.2

The bias for a chosen default memory

To make the forget gate open to a value f at initialization, invert the logistic: $f = \sigma(b) = \frac{1}{1+e^{-b}}$ gives

$$b = \ln \frac{f}{1-f}.$$

For $f = 0.95$: $b = \ln(0.95/0.05) = \ln 19 = 2.94$. Initializing the forget-gate bias near 3, rather than 0, starts the network with a memory horizon of about 90 steps (by Derivation 5.1) instead of the roughly 7 steps that $b = 0$, $f = 0.5$ would give. A single constant, chosen by a one-line inversion, changes the default reach of the network by an order of magnitude — which is why high forget-gate bias is standard practice.

DRILL 5.1

(a) What forget-gate value gives a horizon (to the 1% level) of about 300 steps? (b) What bias produces it?

Show answers

(a) Need $\ln(0.01)/\ln f = 300 \Rightarrow \ln f = -4.605/300 = -0.01535 \Rightarrow f = 0.9848$.
(b) $b = \ln(0.9848/0.0152) = \ln(64.8) = 4.17$.

5.3 Why dropout must respect the recurrent path

This next result catches almost everyone, including people who have used these networks for years. Dropout regularizes a network by randomly zeroing a fraction of its signals during training. On a feedforward network this is benign. On a recurrent network, *where* you apply it decides whether the network can remember at all — and the reason is once again the compounding of a per-step factor.

DERIVATION 5.3

Noise on the recurrent path compounds in time

Suppose dropout with keep-probability 0.9 were applied to the memory as it is carried from each step to the next — on the recurrent path itself. Then over a sequence of 100 steps, the probability that a particular piece of state survives untouched is 0.9^{100}. By Chapter 1's logarithm reflex, $0.9^{100} = e^{100 \ln 0.9} = e^{-10.54} = 2.7 \times 10^{-5}$. The carried memory is annihilated; the very pathway the forget gate worked to protect is destroyed by noise applied a hundred times over.

Now suppose instead the dropout is applied only to the *vertical* connections — the ones feeding from layer to layer at a single time step, not from step to step. Then a given piece of information is perturbed a number of times equal to the network's *depth*, perhaps two or three, regardless of how long the sequence is. The regularization does its job on the transformations between layers while leaving the memory-carrying pathway deterministic. That contrast — noise that compounds per time step versus noise that compounds per layer — is the entire content of a well-known result on recurrent regularization.

TRAP · T5

It is tempting to think inverted dropout's rescaling — dividing surviving signals by the keep-probability to preserve the mean — makes recurrent-path dropout harmless. It does not. Rescaling fixes the *expectation*, but multiplicative noise compounds in *variance*, and it is the variance that destroys the carried state over a long sequence. The design goal is not an unbiased mean; it is a memory pathway that stays deterministic.

5.4 Counting the parameters of a recurrent layer

Before we leave recurrence, a piece of bookkeeping we will reuse. A gated cell computes four quantities per step — a candidate update and three gates — each a linear map of the concatenated input and previous hidden state, plus a bias. For input width d_{in} and hidden width d_h, each of the four maps has $d_h(d_{\text{in}} + d_h)$ weights and d_h biases, so the layer holds

$$4\,d_h\,(d_{\text{in}} + d_h + 1)$$

parameters. Assemble a full model from these and a pattern emerges worth noticing.

WORKED EXAMPLE 5.1

Where the parameters actually are

Take a language model with an embedding of width 1500 over a vocabulary of 10,000; two stacked recurrent layers each of hidden width 1500 (the first fed by the embedding); and an untied output projection back to the vocabulary, with a bias. Count each part.

Embedding: $10{,}000 \times 1500 = 15{,}000{,}000$. First recurrent layer: $4 \times 1500 \times (1500 + 1500 + 1) = 6000 \times 3001 = 18{,}006{,}000$. Second layer: identical, $18{,}006{,}000$. Output projection: $1500 \times 10{,}000 + 10{,}000 = 15{,}010{,}000$. Total: $66{,}022{,}000$, about 66 million.

Notice that the embedding and output projection together are 30 million — about 45% of the model — yet neither adds any depth; they are lookup and readout. This is precisely why scaling studies, when they relate capability to size, count *non-embedding* parameters: the vocabulary tables scale with vocabulary, not with the model's actual computational depth.

DRILL 5.2

A single recurrent layer has input width 512 and hidden width 512. How many parameters does it hold?

Show answer

$4 \times 512 \times (512 + 512 + 1) = 2048 \times 1025 = 2{,}099{,}200$, about 2.1 million.

5.5 Data as a lever: power laws in the wild

Recurrent models were where the field first watched error fall as a clean power law in the quantity of training data — and where the brutal economics of that law became visible. The reflex from Chapter 1, extracting an exponent from a “per tenfold” statement, is all we need to reason about it.

WORKED EXAMPLE 5.2

The price of halving an error

Suppose a speech system's word error rate obeys a power law in training hours, and each tenfold increase in data yields a 40% relative reduction in error. Then error $\propto D^{\beta}$ with $10^{\beta} = 0.6$, so $\beta = \log_{10} 0.6 = -0.222$. To *halve* the error we need $10^{\beta x} = 0.5$, i.e. $x = \log_{10}(0.5)/0.222 = 0.301/0.222 = 1.36$ decades of data — a factor of $10^{1.36} = 22.7$.

Starting from a 10% error at 12,000 hours, reaching 5% therefore demands about $12{,}000 \times 22.7 \approx 273{,}000$ hours. Each halving of error costs a roughly twenty-three-fold multiplication of data. Power laws are smooth and dependable, but they are not cheap — a theme Chapter 7 will make the center of the story.

DRILL 5.3

A different system improves 30% relative per tenfold data increase. What data multiple halves its error?

Show answer

$10^{\beta} = 0.7 \Rightarrow \beta = \log_{10} 0.7 = -0.155$. Halving: $x = 0.301/0.155 = 1.94$ decades $= 10^{1.94} \approx 88\times$.

5.6 What the chapter bought

Memory in a recurrent network is a per-step factor held near one; the horizon over which it survives is a logarithm, and the gate's initial bias sets that horizon by a one-line inversion. Regularization must respect the distinction between noise that compounds in time and noise that compounds in depth — a distinction that decides whether the network can remember at all. And error falls as a power law in data whose exponent you can read straight off a "per tenfold" statement, with consequences for cost that scale is about to make unavoidable. Chapter 6 introduces the mechanism that replaced the recurrent product entirely: attention, which reaches any past token in a single step, and whose own

design choices — the $\sqrt{d}$ scaling chief among them — fall straight out of the variance algebra we have been using all along.

Exercises

A · DRILLS

1. A forget gate sits at $f = 0.9$. Over how many steps does a gradient fall to 1% of its value?
2. What forget-gate bias yields $f = 0.99$ at initialization?
3. A recurrent layer has input width 800 and hidden width 400. Count its parameters.
4. An error law improves 50% relative per tenfold data. Give its exponent β.

B · PROBLEMS

1. **Two places to put noise.** A network runs 200 time steps and is 3 layers deep. Compare the expected number of times a single piece of information is perturbed under (a) dropout on the recurrent path with keep-probability 0.9, and (b) dropout only on the vertical connections with the same keep-probability. Give the surviving fraction in case (a) and the perturbation count in case (b), and state which design preserves memory.
2. **Designing a horizon.** You want a cell whose default memory (to the 1% level) reaches about 500 steps. Find the required forget-gate value and the bias that produces it.
3. **Anatomy of a model.** A model has an embedding of width 1024 over a 30,000-word vocabulary, three recurrent layers of hidden width 1024, and an untied output projection with bias. Give the total parameter count and the fraction that is embedding-plus-projection. Comment on why that fraction matters for scaling studies.

C · CHALLENGE

1. **The carousel as a limit.** Show that the constant-error carousel — unbounded memory — is the $f \to 1$ limit of Derivation 5.1, and explain why a real gated cell cannot simply fix $f = 1$ permanently. (Hint: think about what a network that can never forget would do on a task requiring it to discard stale

information, and connect this to why f is made input-dependent rather than a constant.)

GATE 5 · PASS BEFORE CHAPTER 6

Reproduce Derivations 4.1 (the horizon), 4.2 (the bias inversion), and 4.3 (the compounding contrast) on blank paper. Attempt B-1 and B-2 closed book. You pass when the horizon logarithm and the per-step-versus-per-layer distinction are both automatic — the second is the one students most often get wrong under time pressure.

READINGS FOR CHAPTER 5

Olah, *Understanding LSTM Networks* (2015)

The step-by-step walkthrough section, read twice. Its gate diagrams make the constant-error carousel visual, and Derivation 5.1 is that picture's arithmetic shadow.

Karpathy, *The Unreasonable Effectiveness of Recurrent Neural Networks* (2015)

The character-level examples section, for what a working memory horizon actually buys; then the visualization section, where individual cells are shown tracking quotes and brackets across hundreds of steps — Derivation 5.1's horizon, caught in the act.

Zaremba, Sutskever & Vinyals, *Recurrent Neural Network Regularization* (2014)

Six pages. Find Derivation 5.3 — the argument for dropping only the non-recurrent connections — in their setup.

Amodei et al., *Deep Speech 2* (2015)

The data-scaling figure, and nothing else on a first pass. Read its slope against Worked Example 5.2's power-law reasoning and check you get the same exponent from the picture.

Hochreiter & Schmidhuber, *Long Short-Term Memory* (1997)

The original. Read the introduction's account of vanishing gradients as the problem Derivation 5.1 quantifies.

Attention

Recurrence reaches the past by multiplying through every intervening step. Attention reaches it in one. The consequences — for scale, for cost, and for a famous old trick — are all arithmetic.

A recurrent network reaches a token nine hundred positions back by passing through all nine hundred intervening steps. Attention reaches it in one. That difference sounds like a mere efficiency, but it changes what the network can learn, what it costs, and — as an old preprocessing trick will reveal — what we mean when we say a technique "helps."

An attention layer compares every position to every other by a dot product, turns those comparisons into weights with a softmax, and mixes the values accordingly. Three quantitative facts govern the design: the dot products must be rescaled by the square root of the dimension, or the softmax saturates; the cost is quadratic in sequence length but crosses over with the feed-forward cost only at a specific length; and the maximum distance information must travel drops from linear to constant. Each falls out of tools already in hand.

6.1 The block, and what it holds

Before pricing attention we have to say what it is. An attention layer does not receive queries and keys; it manufactures them. From the sequence of activations X, with one row of width d per position, it forms three projections using learned matrices:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V,$$

and then compares every position to every other, normalizes the comparisons, and mixes the values accordingly:

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_{\text{head}}}}\right) V.$$

The softmax is applied along each row, so each position's output is a weighted average of all positions' value vectors, the weights summing to one. Section 2.2 defined it; Section 6.2 explains the divisor.

A layer does not do this once but H times in parallel, with H separate triples of projections. These are the **heads**, and each works in a narrower space of width $d_{\text{head}} = d/H$. Their outputs are concatenated back to width d and passed through a fourth learned matrix W_O. The point of splitting is that different heads can attend to different things; the price of splitting is nothing at all, and that is worth seeing rather than being told.

Count the projections. Each head's W_Q, W_K, W_V is $d \times d_{\text{head}}$, and there are H of them, so the three together hold $3 \times H \times d \times (d/H) = 3d^2$ weights — the H cancels. Add W_O at $d \times d$ and attention holds $4d^2$ weights per layer, whatever H is. Sixteen heads and one head cost the same. Splitting is free because it partitions a fixed width rather than adding to it, and forgetting this is how people come to believe that more heads means a bigger model.

The rest of the block is the position-wise feed-forward network of Section 2.1, widening d to $4d$ and back, which Worked Example 1.7 already counted at $8d^2$. Each of the two parts is wrapped in a residual connection and a normalization layer — Chapter 4's construction, applied here — which together add only $O(d)$ parameters and are negligible in any count. So a transformer layer holds

$$\underbrace{4d^2}_{\text{attention}} + \underbrace{8d^2}_{\text{feed-forward}} = 12d^2 \text{ weights},$$

and by Chapter 1's corollary it costs about that many multiply–accumulates per position. One number, and it prices every transformer in the rest of this book.

WORKED EXAMPLE 6.1

Seven billion, counted from four numbers

Part III costs everything against a model described only as "7 B parameters, 32 layers, 32 heads of dimension 128". Check that those numbers are consistent, and recover the vocabulary's share.

The head dimension fixes the width: $d = H\,d_{\text{head}} = 32 \times 128 = 4096$. Then each layer holds $12d^2 = 12 \times 4096^2 = 201{,}326{,}592$ weights, and thirty-two layers hold $32 \times 201{,}326{,}592 = 6.44 \times 10^9$.

The tables are extra. At a vocabulary of $32{,}000$ with untied embedding and readout, they add $2 \times 32{,}000 \times 4096 = 0.26 \times 10^9$, for a total of 6.70×10^9 — the seven billion, to two figures, from four numbers and one formula.

Notice what the tables are here: four percent of the model, against the forty-five percent they occupied in the recurrent model of Worked Example 5.1. Nothing about vocabularies changed. The layers grew.

DRILL 6.1

(a) A model has $d = 5120$ and 40 heads. Give d_{head}, and the parameters in one layer. (b) The same model has 40 layers. Give the non-embedding parameter count. (c) Its designers double the head count to 80, holding d fixed. By how much does the parameter count change?

Show answers

(a) $d_{\text{head}} = 5120/40 = 128$; one layer holds $12 \times 5120^2 = 314{,}572{,}800$, about 315 M. (b) $40 \times 3.15 \times 10^8 = 1.26 \times 10^{10}$, about 12.6 B. (c) Not at all. The heads partition d; doubling H halves d_{head} and the H cancels out of $3d^2$ exactly as before.

6.2 Why divide by the square root of the dimension

An attention score is a dot product of a query vector and a key vector, each of dimension d_{head} — written d throughout this section, since only the dimension of

the vectors being compared matters to the argument. If their components are independent with unit variance, Chapter 1 already told us the variance of that dot product.

DERIVATION 6.1

The score variance, and the scale that tames it

For $q, k \in \mathbb{R}^d$ with independent, zero-mean, unit-variance components, the score is $q \cdot k = \sum_{i=1}^{d} q_i k_i$. Each term is a product of independent unit-variance variables, hence has variance 1; the d terms are independent, so their variances add:

$$\mathrm{Var}(q \cdot k) = d, \qquad \text{standard deviation} = \sqrt{d}.$$

For $d = 512$, the raw scores have standard deviation $\sqrt{512} = 16\sqrt{2} \approx 22.6$. Feeding logits of that magnitude into a softmax pushes almost all its mass onto a single entry: the output is nearly one-hot, and its gradient — which is largest when the distribution is spread out — collapses toward zero. This is the sigmoid saturation of the recurrent era reborn inside attention, and it is solved the same way: keep the pre-activation at unit scale. Dividing each score by $\sqrt{d}$ restores the variance to 1 and keeps the softmax responsive.

The lesson generalizes a theme running through the book. In Chapter 3 the fix for exploding activations was a $1/\sqrt{m}$ on the weights; here the fix for saturating scores is a $1/\sqrt{d}$ on the logits. Both are the same instinct — hold the variance at one as signal crosses an operation — and both are one line of variance algebra.

DRILL 6.2

Attention operates at dimension $d = 128$. (a) What is the standard deviation of an unscaled score? (b) By what factor must scores be divided to restore unit variance?

Show answers

(a) $\sqrt{128} = 8\sqrt{2} \approx 11.3$. (b) Divide by $\sqrt{128} \approx 11.3$; this restores variance to 1.

6.3 What attention costs

A transformer layer has two expensive parts: the attention itself, and the position-wise feed-forward network that follows it. Both costs come from the matrix-multiply primitive of Chapter 1 — multiplying an $a \times b$ matrix by a $b \times c$ matrix costs abc multiply–accumulates — and comparing them tells you which regime a model lives in.

DERIVATION 6.2

The attention–feedforward crossover

For a sequence of length n and model width d: attention forms an $n \times n$ score matrix from $n \times d$ queries and keys ($n^2 d$ operations), then mixes an $n \times n$ weight matrix with $n \times d$ values (another $n^2 d$), totaling $2n^2 d$. The feed-forward network maps width d up to $4d$ and back, at each of n positions: $n \cdot d \cdot 4d$ up and the same down, totaling $8nd^2$. Setting them equal,

$$2n^2 d = 8nd^2 \quad \Longrightarrow \quad n = 4d.$$

Below $n = 4d$, the feed-forward network dominates and attention is a minor cost. Only past that crossover does the quadratic term take over. This is why quadratic attention was tolerable for the sentence-length sequences of early machine translation and became the binding constraint only when context windows grew to thousands of tokens — a story Chapter 11 resumes in the modern setting.

The other cost that matters is not arithmetic but distance. In a recurrence, information from position i reaches position j only by passing through all $|i - j|$ steps between them: the path length is linear in the gap, and it is along that long path that gradients decay. In attention, any position reaches any other in a single operation: the maximum path length is constant, independent of the gap. A token nine hundred positions back is no harder to consult than its neighbor. That collapse from linear to constant path length — not the arithmetic — is the deepest reason attention learns long-range structure where recurrence struggled.

DRILL 6.3

A model has width $d = 512$ and feed-forward width 2048. At what sequence length do attention and feed-forward costs balance? Below that length, which dominates?

Show answer

$n = 4d = 2048$. Below $n = 2048$, the feed-forward network dominates.

6.4 Temperature: reshaping confidence without reordering it

When a model samples its next token, it divides the logits by a temperature T before the softmax. This is worth a moment because a simple monotonicity argument settles what temperature can and cannot do. Dividing by a positive T is a monotone transformation of the logits: it can sharpen the distribution (small T, toward the argmax) or flatten it (large T, toward uniform), but it can never change which logit is largest. Temperature reshapes confidence; it never reorders the candidates.

WORKED EXAMPLE 6.2

Two temperatures on a two-token vocabulary

Logits are $z_1 = 2, z_2 = 0$. At $T = 1$, the probability of token 1 is $\sigma(2) = 1/(1 + e^{-2}) = 1/(1.135) = 0.881$. At $T = 0.5$, the logits double to $4, 0$, so $\sigma(4) = 1/(1 + e^{-4}) = 1/(1.018) = 0.982$: colder sampling has sharpened the lead. As $T \to 0$ the distribution tends to the argmax; as $T \to \infty$ it tends to uniform. At no finite positive T does token 2 overtake token 1 — monotonicity forbids it.

WORKED EXAMPLE 6.3

A perplexity drop that sounds enormous and is half a bit

A language model improves from perplexity 114.5 to perplexity 78.4 — a 31% relative reduction. How large is that gain in bits per token?

Perplexity is 2^H where H is cross-entropy in bits, so $H = \log_2(\text{perplexity})$. Before: $\log_2(114.5) = 6.84$ bits/token. After: $\log_2(78.4) = 6.29$ bits/token. The improvement is $6.84 - 6.29 = 0.55$ bits per token — barely half a bit.

The lesson: perplexity is exponential in bits. A gain that sounds dramatic in the exponential unit can be modest in the linear unit that file sizes, bandwidth budgets, and the compression arguments of Chapter 10 are denominated in. Whenever someone reports a perplexity improvement, convert it to bits before deciding whether to be impressed.

6.5 The reversal trick, and what it reveals

Before attention, sequence-to-sequence models used a recurrent encoder to read the source and a recurrent decoder to write the target. A curious preprocessing trick helped: reversing the source sequence before encoding. Why should reading a sentence backward help translate it? The answer is one of the most instructive small calculations in the book, because it distinguishes two things students routinely conflate — optimization and representation.

Work out what you think reversal should do before reading on. The natural expectation is that it shortens something — that reading the sentence backwards must somehow bring related words closer together, since otherwise why would it help? Write down what you would predict for the average distance between a source word and its translation. Then compare it with what follows. The gap between the two is the lesson.

DERIVATION 6.3

The lag of the reversal trick, and its unchanged mean

Suppose source and target both have length n, with a strictly monotonic alignment: target position j corresponds to source position j. Define the *lag* of position j as the number of recurrent steps between reading source token j and emitting target token j.

Unreversed. Source token j is read at encoder step j; target token j is emitted at step $n + j$ (after the whole source is read). The lag is $(n + j) - j = n$, the same for every position.

Reversed. The source is fed backward, so source token j is now read at step $n - j + 1$. Target token j is still emitted at step $n + j$. The lag is $(n + j) - (n - j + 1) = 2j - 1$.

Now compute the mean lag over all positions:

$$\frac{1}{n}\sum_{j=1}^{n}(2j - 1) = \frac{1}{n} \cdot n^2 = n.$$

The mean lag is *exactly unchanged* at n. And the maximum lag has grown, from n to $2n - 1$. By every aggregate measure of distance, reversal made things no better — and in the worst case, worse.

So why does it help? Because it changes the *distribution* of lags without changing their mean. Unreversed, every alignment is equally distant, n steps away; the network has no easy foothold anywhere. Reversed, the early alignments become very short — the first target position now lags its source by just one step — at the cost of longer lags later. Gradient descent can seize those early, short-lag alignments immediately, and once the beginning of the sentence is aligned, the rest follows. The trick works because optimization needs an early handhold, not because the representational problem got any smaller.

TRAP · T3

The reversal trick is the canonical mean-versus-maximum confusion. It is natural to assume a trick that helps must reduce the average distance information travels — but here the mean is provably unchanged and the maximum is worse. Whenever a technique helps without improving the aggregate you first reach for, suspect that it is reshaping a distribution to aid *optimization*, not shrinking a quantity to aid *representation*. Attention, incidentally, removes the trade-off entirely by making every lag constant.

6.6 What the chapter bought

Attention's $\sqrt{d}$ scaling is the same variance-control instinct as sensible initialization, applied to logits. Its cost is quadratic in length but only overtakes the feed-forward network past $n = 4d$, which is why quadratic attention was affordable for years. Its constant path length is the true source of its reach. And the reversal trick, properly analyzed, is a lesson in reading a technique correctly: some interventions change what a network *can* represent, and others merely change how easily gradient descent *finds* a representation — and telling the two apart is a skill the rest of the book will keep demanding. Chapter 7 takes up what becomes, once architecture stops being the bottleneck, the field's central question: given a fixed budget, how much model and how much data — answered by the derivation that turned compute-optimal training from a matter of taste into a calculation.

Exercises

A · DRILLS

1. Attention runs at $d = 256$. Give the standard deviation of an unscaled score and the required scaling factor.
2. A model has width $d = 768$. Give the sequence length at which attention and feed-forward costs balance.

3. Logits are $3, 1$. Give the probability of the first token at temperature $T = 1$ and $T = 0.5$.

4. For a source of length $n = 6$ with the reversal trick, give the lag of positions 1 and 6.

B · PROBLEMS

1. **Which regime.** A model has width $d = 1024$ and is run at sequence length $n = 8192$. Compute the attention and feed-forward MAC counts (in units of d^2 and n), give their ratio, and state which dominates. Then find the sequence length at which they would balance.

2. **The reversal trick in full.** For a monotonic length-n alignment, derive the reversed lag $2j - 1$ from scratch, prove the mean lag is exactly n, and give the maximum lag. Then explain in two sentences why the trick aids optimization despite leaving the mean unchanged.

3. **Temperature cannot reorder.** Prove that for any positive temperature T and any two logits $z_a > z_b$, the softmax probability of a exceeds that of b. Conclude that sampling temperature reshapes confidence but never changes the ranking of candidates.

C · CHALLENGE

1. **One instinct, three chapters.** The $1/\sqrt{m}$ of Chapter 3, the normalization-inside-the-branch of Chapter 4, and the $1/\sqrt{d}$ of this chapter are all the same principle. State that principle in one sentence, then show how each of the three is an instance of it, identifying in each case what quantity is being held at what value and what would go wrong otherwise.

> **GATE 6 · PASS BEFORE CHAPTER 7**
>
> Reproduce Derivations 5.1 (the $\sqrt{d}$ scaling), 5.2 (the $n = 4d$ crossover), and 5.3 (the reversal lag) on blank paper. Attempt B-1 and B-2 closed book. You pass when the crossover and the mean-lag proof come without hesitation — the latter is the one most often botched, because the sum $\sum(2j - 1) = n^2$ must be seen instantly.

READINGS FOR CHAPTER 6

Vaswani et al., *Attention Is All You Need* (2017)

Read for the shapes and the $\sqrt{d_k}$ scaling of Derivation 6.1; the cost accounting of Section 6.3 is theirs made explicit.

Rush, *The Annotated Transformer*

The Model Architecture section, where the paper is rebuilt as runnable code. Read it for the wiring once the derivations are clear, and match its tensor shapes against Section 6.1's projections.

Bahdanau, Cho & Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate* (2014)

The alignment-matrix figure. It shows the constant-path-length benefit of Section 6.3 before anyone had named it — each bright cell is a lag that recurrence would have had to travel.

Sutskever, Vinyals & Le, *Sequence to Sequence Learning with Neural Networks* (2014)

Section 3.3 is the reversal trick of Derivation 6.3, in its original habitat.

Weng, *Attention? Attention!* (2018)

Its summary table of attention variants, read after the derivations rather than before, to place what you have derived in the wider family.

The Economics of Scale

Once architecture stops being the bottleneck, progress becomes a budgeting problem: given a fixed amount of compute, how should you split it between a bigger model and more data? The answer is a single constrained minimization — the most consequential in the field.

You have a fixed budget of computation, and two ways to spend it: a bigger model, or more data for a smaller one. Spend it wrong and you will train a model that is worse than the one the same money could have bought. For years this was decided by intuition and precedent. It is a calculus problem, and the calculus is the most valuable thing in this book.

What makes it tractable is that loss falls as a smooth power law in both model size and data, and that compute is very nearly the product of the two. Those two facts are enough. We will set the problem up, solve it by substitution, read off how each quantity should scale with the budget, and finish by re-budgeting a famously over-large model into a much smaller and better-trained one at identical cost.

7.1 The loss surface of scale

Empirically, the loss of a trained language model is captured remarkably well by a sum of three terms:

$$L(N, D) = AN^{-a} + BD^{-b} + E.$$

Here N is the number of parameters and D the number of training tokens. The first term falls as the model grows; the second falls as the data grows; the third, E, is an irreducible floor — the entropy of language itself, which no amount of scale removes. The exponents a and b are small positive numbers, measured from experiments and, in the spirit of this book, *given* to you when needed. The shape of this surface is all we require.

Two facts about power laws set the stage. First, because the exponents are small, improvements are expensive: halving a term takes far more than a doubling of its variable. Second, the two reducible terms trade off against each other under a compute budget, and finding the best trade is the whole game.

> **WORKED EXAMPLE 7.1**
>
> **What a halving really costs**
>
> Take the parameter term with exponent $a = 0.076$. To halve it we need $N^{-0.076}$ to fall by a factor of two, i.e. to multiply N by k where $k^{-0.076} = \frac{1}{2}$. Then $0.076 \log_2 k = 1$, so $\log_2 k = 13.16$ and $k = 2^{13.16} \approx 9{,}100$. Halving this loss term demands a nine-thousand-fold increase in parameters. The power law is smooth and utterly dependable — and staggeringly expensive. “Reliable” is not the same as “cheap.”

7.2 The compute-optimal allocation

Training compute is, to a good approximation, proportional to the product of parameters and tokens: $C = kND$, with $k = 6$ — two floating-point operations per parameter per token on the forward pass and four on the backward, as counted in Chapter 2. It is not an empirical constant; it is three passes over the weights at two operations each. Given a fixed budget C, we want the split of that budget between N and D that minimizes the loss. This is a constrained minimization, and we solve it exactly as Chapter 1 prescribed: use the constraint to eliminate one variable, then differentiate.

One warning before we start, because nearly everyone makes the same mistake here on first contact. When two terms trade off under a constraint, it is tempting to assume the optimum is where the two terms are *equal*. That is a good instinct and it is very nearly right — but not quite, and the correction matters. Watch for where the exponents appear, and note that they appear because we differentiated, not because anyone put them there by hand.

DERIVATION 7.1 · THE CROWN JEWEL

The exponent-weighted balance

Minimize $L(N, D) = AN^{-a} + BD^{-b} + E$ subject to $C = kND$. Substitute $D = C/(kN)$, so the data term becomes $B\,(kN/C)^b$, and the irreducible E drops out of the differentiation:

$$L(N) = AN^{-a} + B\left(\tfrac{k}{C}\right)^b N^b.$$

Differentiate and set to zero:

$$\frac{dL}{dN} = -aAN^{-a-1} + bB\left(\tfrac{k}{C}\right)^b N^{b-1} = 0.$$

The second term is bBD^{-b}/N once we substitute D back. Multiplying through by N gives the optimum condition:

$$\boxed{aAN^{-a} = bBD^{-b}}.$$

At the optimal allocation, the two reducible loss terms are equal *after weighting each by its own exponent*. Solving the balance together with $ND \propto C$ yields the scaling of each with the budget:

$$N \propto C^{\,b/(a+b)}, \qquad D \propto C^{\,a/(a+b)}.$$

And the ratio of tokens to parameters scales as $D/N \propto C^{(a-b)/(a+b)}$.

TRAP · T2 · THE MOST EXPENSIVE ERROR IN THE BOOK

It is tempting to write the balance as $AN^{-a} = BD^{-b}$ — the bare terms equal, without the exponent weights. This is wrong, and it produces the wrong optimal ratio whenever $a \neq b$. The reason the exponents appear is that the constraint trades N for D at a rate that depends on how fast each term responds — and that rate is exactly the exponent. Substitution makes the weights fall out automatically, which is precisely why we solve it that way rather than by guessing that the terms should balance.

The special case $a = b$ is worth naming, because it is the regime people carry in their heads. When the exponents are equal, $N \propto C^{1/2}$ and $D \propto C^{1/2}$: parameters and tokens grow together as the square root of compute, and the ratio D/N is a

constant, independent of budget. “Use a fixed number of tokens per parameter” — the rule of thumb that a model should see roughly twenty tokens for each parameter — is this equal-exponent case.

DRILL 7.1

Show that when $a = b$, the token-to-parameter ratio D/N is independent of the compute budget C, and that both N and D scale as $C^{1/2}$.

Show answer

With $a = b$, the exponents in $N \propto C^{b/(a+b)}$ and $D \propto C^{a/(a+b)}$ both equal $\frac{1}{2}$. Then $D/N \propto C^{1/2}/C^{1/2} = C^0$, a constant.

7.3 Re-budgeting a real model

The derivation's practical bite is that it exposes over-large, under-trained models. A model that is too big for the data it saw is spending compute inefficiently: the same compute, re-allocated toward the optimal ratio, would train a smaller model on more data to a lower loss.

WORKED EXAMPLE 7.2

From 175 billion to 51 billion

Consider a well-known model of $N = 175$ billion parameters trained on $D = 300$ billion tokens — a ratio of under two tokens per parameter, far below the optimal twenty. Its compute is $C = 6ND = 6 \times (1.75 \times 10^{11})(3 \times 10^{11}) = 3.15 \times 10^{23}$ operations.

Hold that compute fixed and re-allocate to the ratio $D = 20N$. Then $C = 6N(20N) = 120N^2$, so $N^2 = 3.15 \times 10^{23}/120 = 2.625 \times 10^{21}$, giving $N = 5.12 \times 10^{10}$ — about **51 billion parameters**, trained on roughly 1.02 trillion tokens. The same compute buys a model a third the size, trained on more than three times the data, at a lower loss. This single calculation redirected how the field spends its largest training budgets.

DRILL 7.2

Under the coupling $D \propto N^{0.74}$, if parameters are increased by a factor of 1000, by what factor should the dataset grow?

Show answer

$1000^{0.74} = 10^{3\times 0.74} = 10^{2.22} \approx 166$. A thousandfold in parameters calls for about a 166-fold increase in data.

7.4 What the chapter bought

The compute-optimal allocation is a constrained minimization whose answer — the exponent-weighted balance and the resulting $C^{b/(a+b)}$ scaling — turned a matter of intuition into a matter of arithmetic. Its equal-exponent special case is the familiar "fixed tokens per parameter" rule; its general form exposes when a model is misallocated; and applying it to a real over-large model shrinks it by a factor of three at equal compute. This is the derivation to own most thoroughly in the entire book, and the three-days protocol below is not optional. Chapter 8 turns from the economics of scale to its machinery: once a model is too large for one accelerator, how is the work split across many, and what does that splitting cost?

Exercises

A · DRILLS

1. A loss term falls as $N^{-0.095}$. What parameter multiple halves it?
2. A budget is $C = 1.2 \times 10^{24}$ operations. Using $C = 120N^2$ (the ratio-20 recipe), give N and D.
3. For exponents $a = 0.34, b = 0.28$, give the exponents with which N and D scale in C.
4. State the compute-optimal balance condition, with the exponent weights in place.

B · PROBLEMS

1. **The full allocation.** Starting from $L(N, D) = AN^{-a} + BD^{-b} + E$ and $C = kND$, derive by substitution the optimum condition $aAN^{-a} = bBD^{-b}$ and the scalings $N \propto C^{b/(a+b)}$, $D \propto C^{a/(a+b)}$. State in one sentence why the bare-term balance is wrong.

2. **Re-budgeting.** A model has $N = 70$ billion parameters and $D = 1.4$ trillion tokens. Holding its compute $6ND$ fixed, find the parameter count at the ratio $D/N = 20$, and the corresponding token count. Is the original over- or under-trained?

3. **The cost of a halving.** For $a = 0.076$, show that halving the parameter loss term costs a factor $2^{1/a}$ in N, and evaluate it. Then, separately, comment on what this implies about the compute cost of steady loss reduction, given that compute scales with N.

C · CHALLENGE

1. **When the ratio drifts.** Using $D/N \propto C^{(a-b)/(a+b)}$, determine the sign of the drift in the optimal tokens-per-parameter ratio as the budget grows, for the case $a < b$ and the case $a > b$. Explain, in terms of which loss term responds faster to its variable, why the budget should be steered toward the slower-improving resource. (This anticipates why real recipes have drifted toward more tokens per parameter over time — a theme Chapter 17 completes.)

GATE 7 · PASS BEFORE CHAPTER 8 — THREE-DAYS PROTOCOL

Reproduce Derivation 7.1 — substitution, differentiation, the exponent-weighted balance, and the $C^{b/(a+b)}$ scalings — on blank paper, from nothing, on three separate days. Between attempts, do not reread it; let it settle and reconstruct it cold. This is the second of the book's three protocol derivations, and it is the one you will lean on most: Chapter 14's DPO collapse uses the same substitute-and-cancel discipline, and the fluency transfers directly. You pass when all three reconstructions are complete and correct, and B-2's re-budgeting comes without notes.

READINGS FOR CHAPTER 7

Hoffmann et al., *Training Compute-Optimal Large Language Models* (Chinchilla, 2022)

The paper this chapter is built on. Its three separate approaches to the frontier are three routes to your one first-order condition; read Section 3 against Derivation 7.1.

Kaplan et al., *Scaling Laws for Neural Language Models* (2020)

Where the power-law form $L(N, D)$ was first fit; read Figures 1–4 to see the surface you just minimized on.

Background on scaling behavior

Useful framing for why smooth power laws coexist with apparent jumps — a tension Chapter 9 resolves.

Branwen, *The Scaling Hypothesis*

Long, and three pages of the opening argument are enough. The ideology built on top of Derivation 7.1; read it to see what people concluded from a first-order condition.

Sutton, *The Bitter Lesson* (2019)

One page. The philosophical companion to the compute-optimal result.

The Machinery of Scale

A model too large for one accelerator must be split across many. The splitting is never free — and the cost, like everything else here, is a short calculation you can do from a timing diagram.

A model that will not fit on one accelerator must be cut into pieces and spread across many. Nothing about that is free. Cut it into sequential stages and the accelerators spend part of their time waiting on each other; spread its state across devices and you must first know how large that state actually is. Both costs are countable, and counting them is this chapter.

Assign consecutive layers to consecutive devices — pipeline parallelism — and no device can begin until the one before it has finished, so the fleet idles through part of every step. Divide the optimizer state instead and the arithmetic is trivial division, but the figure you must compute first is the *undivided* total, because that is what tells you whether the run is possible at all. Both penalties shape real systems, and both are a few lines of counting.

8.1 The pipeline bubble

Split a network into P sequential stages, one per accelerator. A batch flows through stage 1, then stage 2, and so on. If we fed one whole batch through, only one accelerator would be busy at a time — catastrophic waste. So we cut the batch into M micro-batches and stream them, so that while micro-batch 2 is in stage 1, micro-batch 1 is already in stage 2. But the pipeline still has to fill at the start and drain at the end, and during fill-and-drain some accelerators sit idle. That idle fraction is the *bubble*.

DERIVATION 8.1

The bubble fraction from a timing diagram

Let each micro-batch take one unit of time per stage. Draw the schedule: micro-batch m can enter stage s only after it has left stage $s-1$ and after micro-batch $m-1$ has left stage s. Tracing the diagonal, the last micro-batch enters stage 1 at time $M-1$ and exits the final stage P at time $M+P-1$. So the whole batch completes in $M+P-1$ time units.

The useful work is MP stage-units (each of M micro-batches through each of P stages). The total accelerator-time spent is $P(M+P-1)$ (all P accelerators, held for the whole duration). The idle — bubble — fraction is

$$1-\frac{MP}{P(M+P-1)}=\frac{P-1}{M+P-1}.$$

Two boundary cases carry the meaning. With $M=1$ — no micro-batching, naive model parallelism — the bubble is $(P-1)/P$, which for $P=8$ is $7/8=87.5\%$ idle. That catastrophe is the entire reason micro-batching exists. As M grows large, the bubble tends to zero: more micro-batches amortize the fill-and-drain.

WORKED EXAMPLE 8.1

How many micro-batches to hide the bubble

For an 8-stage pipeline, how many micro-batches keep the bubble at or below 5%? Require $\frac{7}{M+7}\le 0.05$, i.e. $M+7\ge 140$, so $M\ge 133$. You need well over a hundred micro-batches per optimizer step just to keep one-twentieth of your hardware from sitting idle — which is why pipeline parallelism is a tool of last resort, reached for only when a model will not fit any other way.

DRILL 8.1

A 16-stage pipeline runs with $M = 48$ micro-batches. What fraction of accelerator-time is lost to the bubble?

Show answer

$(P-1)/(M+P-1) = 15/(48+15) = 15/63 = 0.238$, about 24%.

8.2 Sharding and the memory wall

The second penalty is memory, and its headline number is worth building rather than accepting. Training keeps far more per parameter than the parameter, and what it keeps is countable.

DERIVATION 8.2

Sixteen bytes for every parameter

Modern training runs in **mixed precision**: the arithmetic is done in half precision because that is what the hardware is fast at, while a full-precision copy of each weight is kept so that many small updates are not lost to rounding. Count what has to be resident, per parameter, to take one step.

Half-precision weight, read by the forward and backward passes — **2** bytes.

Half-precision gradient, produced by the backward pass — **2** bytes.

Full-precision master weight, which the update is actually applied to — **4** bytes.

Full-precision first moment, the optimizer's running mean of gradients — **4** bytes.

Full-precision second moment, its running mean of squared gradients — **4** bytes.

$$2 + 2 + 4 + 4 + 4 = 16 \text{ bytes per parameter.}$$

The two moments are what an adaptive optimizer keeps so that each coordinate's step can be scaled by its own gradient history — the cheap approximation to per-axis conditioning sketched in the challenge at the end of Chapter 3. They are why the figure is 16 and not 8: plain gradient descent would keep no moments at all, and the same run would need half the memory and converge far worse.

Note which entries a technique could hope to remove, because it decides what any memory saving can possibly be worth. Freezing a parameter removes its gradient and all three full-precision copies and leaves only the 2 bytes that must be read to compute anything at all — fourteen of the sixteen. That is the arithmetic underneath Chapter 13's low-rank adaptation, and it is why the saving there is two orders of magnitude rather than a factor of two.

Multiply by the parameter count and the total is enormous; the remedy is to shard that state across many accelerators, each holding a slice. The arithmetic is mere division — but you should always compute the *unsharded* figure first, because that is the wall the sharding exists to climb.

WORKED EXAMPLE 8.2

The wall, and the ladder over it

A 50-billion-parameter model at 16 bytes of optimizer state per parameter needs $16 \times 5 \times 10^{10} = 8 \times 10^{11}$ bytes — 800 gigabytes — of state. No single accelerator holds that; it is the memory wall. Shard it with no redundancy across 64 accelerators and each holds $800/64 = 12.5$ gigabytes, which fits comfortably. The sharding did nothing clever — it divided by 64 — but computing the unsharded 800 GB first is what shows you why the division was necessary.

DRILL 8.2

A 13-billion-parameter model at 16 bytes/parameter of state is sharded across 32 accelerators. Give the unsharded total and the per-accelerator share.

Show answers

Unsharded: $16 \times 1.3 \times 10^{10} = 2.08 \times 10^{11}$ bytes $= 208$ GB. Per accelerator: $208/32 = 6.5$ GB.

TRAP · T7 · RATE VERSUS STOCK

The bubble is a *fraction of accelerator-time*, not an amount of time, and the two get confused constantly. A 17.9% bubble on an eight-stage pipeline does not mean seventeen-point-nine of anything — it means that across the whole fleet, for the whole step, that proportion of available accelerator-seconds produced nothing. Doubling the micro-batch count halves the fraction while *lengthening* the wall-clock step, because more micro-batches means more work per step. Efficiency and latency move in opposite directions here, and a claim about one says nothing about the other.

The same caution applies to sharding. “12.5 GB per accelerator” is a stock; “800 GB of state” is the stock that matters for feasibility. Quoting only the first hides whether the run is possible at all.

8.3 Activations, and buying memory back with arithmetic

Weights and optimizer state are not the whole bill. The backward pass needs the activations the forward pass produced — Section 2.3 — so every intermediate result must be held from the moment it is computed until the gradient reaches it. That store scales with quantities the previous section never mentioned: the batch, the sequence length, and the depth.

Guess the size before computing it. Optimizer state for a 7-billion-parameter model is 112 GB and does not depend on the batch at all; activations depend on nothing else so strongly. Most people expect activations to be the smaller term. At training batch sizes they are routinely the larger one, and the reason is that they are counted per token rather than per parameter.

> **WORKED EXAMPLE 8.3**
>
> **What the forward pass leaves behind**
>
> A transformer layer of width d stores, per token, a handful of intermediate vectors: the block's input, the three projections, the attention output, and the two feed-forward activations at widths d and $4d$. Call it $c\,d$ values per token per layer, with c around 10 once everything is counted.
>
> For $d = 4096$, $L = 32$, half precision, a batch of 8 sequences of 4096 tokens — $Bs = 32{,}768$ tokens in flight:
>
> $$10 \times 4096 \times 32 \times 32{,}768 \times 2 \text{ bytes} = 8.6 \times 10^{10} = 86 \text{ GB}.$$
>
> Against 112 GB of optimizer state and 14 GB of weights, the activations are the second-largest term and within a factor of two of the largest — on a batch most practitioners would call modest. Double the sequence length and they pass everything else.

The remedy is the trade this book keeps meeting: spend the abundant resource to relieve the binding one. Store the activations at only a few points in the network — **checkpoints** — and recompute the rest during the backward pass from the nearest one below.

DERIVATION 8.3

The square root of depth

Split L layers into g equal segments, storing only each segment's input. Two costs result. The stored activations are the g checkpoints, plus, while recomputing inside one segment, that segment's L/g layers:

$$\text{stored} \propto g + \frac{L}{g}.$$

Differentiate with respect to g — Chapter 1's constrained-minimum reflex, with no constraint to substitute this time. Setting $1 - L/g^2 = 0$ gives $g = \sqrt{L}$, and the stored total is $2\sqrt{L}$ rather than L.

The price is one extra forward pass over each segment during the backward sweep: about $2N$ FLOPs per token on top of the $6N$ of Chapter 2, so roughly a third more arithmetic. At $L = 64$ the memory falls by a factor of $64/(2 \times 8) = 4$ for that third. Memory is bought with arithmetic at a rate the roofline of Chapter 11 will make it easy to judge.

DRILL 8.3

(a) For $L = 100$ layers, give the optimal number of checkpoint segments and the factor by which stored activations fall. (b) A run holds 86 GB of activations and 112 GB of optimizer state. Checkpointing at $\sqrt{L}$ with $L = 32$; what is the new activation figure, and which term now binds?

Show answers

(a) $g = \sqrt{100} = 10$; stored goes from 100 to $2\sqrt{100} = 20$, a factor of 5. (b) $2\sqrt{32} = 11.3$ against 32, so activations fall to $86 \times 11.3/32 = 30$ GB. Optimizer state now dominates, and the next thing to attack is the 16 bytes of Derivation 8.2 rather than the activations.

8.4 What the splitting costs in traffic

One cost remains, and it is the one that decides which axis of parallelism you reach for. Splitting work across devices means the devices must talk, and their

link is slower than their memory by roughly the same margin that memory is slower than arithmetic.

Data parallelism — every device holds the whole model and processes a different slice of the batch — requires that the gradients be summed across devices before the step. That sum moves every gradient once: $2N$ bytes per device per step in half precision, whatever the batch size. Tensor parallelism, which splits individual matrices across devices, instead exchanges *activations* at every layer, so its traffic scales with tokens in flight rather than with parameters. Pipeline parallelism moves only the activations at stage boundaries, which is the least traffic of the three — and buys that with the bubble of Derivation 8.1.

DRILL 8.4

A 7-billion-parameter model is trained data-parallel in half precision over a link carrying 50 GB/s. Give the bytes summed per device per step, and the time that takes. If a step's arithmetic occupies 400 ms, what fraction of the step is communication?

Show answer

$2N = 1.4 \times 10^{10}$ bytes $= 14$ GB, taking $14/50 = 0.28$ s. Against a 400 ms step that is 280 ms of talking to 400 ms of working — 41% of the step, and the reason gradient summation is overlapped with the backward pass rather than performed after it.

8.5 What the chapter bought

None of the penalties of scale is mysterious; all submit to counting. The pipeline bubble, $(P-1)/(M+P-1)$, comes straight from a timing diagram and explains why micro-batching is mandatory and why pipelines are a last resort. Memory sharding is division, but the unsharded figure is the wall that motivates it. Activations are counted per token rather than per parameter, which is why they overtake the optimizer state on any serious batch, and checkpointing buys them back at $2\sqrt{L}$ for about a third more arithmetic. Communication is the fourth cost, and the axis you choose is a choice about which of parameters or tokens you would rather move. These are the arithmetic facts that shape how the largest training runs are laid out across hardware — and they return, transformed, in Chapter 11,

where the same roofline thinking governs not training but the economics of serving a trained model one token at a time. Before that, Chapter 9 confronts a subtler problem: once models are this large and this capable, how do we know what they can actually do — and when a benchmark is lying to us?

Exercises

A · DRILLS

1. A 4-stage pipeline runs with 12 micro-batches. Give the bubble fraction.
2. What is the bubble fraction of a 10-stage pipeline with no micro-batching ($M = 1$)?
3. A 30-billion-parameter model at 16 bytes/parameter: give the unsharded state in GB.
4. The same model sharded over 40 accelerators: give the per-accelerator state.

B · PROBLEMS

1. **Deriving the bubble.** From the scheduling rules — micro-batch m enters stage s only after leaving $s-1$ and after $m-1$ leaves s — derive the completion time $M+P-1$ and the bubble fraction $(P-1)/(M+P-1)$. Evaluate at $(P, M) = (8, 1)$ and $(8, 32)$, and say what the two numbers mean operationally.
2. **Sizing micro-batches.** For a 12-stage pipeline, find the minimum number of micro-batches that keeps the bubble at or below 8%.
3. **The wall.** A 175-billion-parameter model is trained at 16 bytes/parameter of state. Give the unsharded total, and the number of 80 GB accelerators required to hold the state alone (ignoring activations). Comment on why the unsharded figure is the right thing to compute first.

C · CHALLENGE

1. **Two penalties at once.** Suppose adding pipeline stages P lets you fit a larger model (relieving the memory wall) but raises the bubble (wasting time). Sketch qualitatively how the bubble fraction and the per-accelerator memory each move as P increases at fixed M, and argue why real systems combine pipeline parallelism with other forms of sharding rather than relying on

pipelining alone. (This is the conceptual seed of modern multi-axis parallelism.)

GATE 8 · PASS BEFORE CHAPTER 9

Reproduce Derivation 8.1 from a hand-drawn timing diagram — the completion time and the bubble fraction — on blank paper, and evaluate both boundary cases. Attempt B-1 and B-2 closed book. You pass when you can draw the staircase, read $M + P - 1$ off it, and produce the 87.5% idle figure for a naive 8-stage pipeline without hesitation.

READINGS FOR CHAPTER 8

Huang et al., *GPipe* (2018)

The bubble-overhead figure and the timing diagram beside it. Match their pipeline schedule against Derivation 8.1's, and confirm their overhead expression is your $(P-1)/(M+P-1)$.

Shoeybi et al., *Megatron-LM* (2019)

The partitioning figures, for a different axis of splitting from Chapter 8's. Read for which resource tensor parallelism attacks, and confirm from Section 8.4 that it trades traffic where pipelining trades idle time.

Rajbhandari et al., *ZeRO* (2019)

The state-sharding of Section 8.2, systematized; the three stages of ZeRO are three levels of the division you just did.

Austin et al., *How to Scale Your Model*

The best free systems text on this material; work its parallelism chapter now, and note that its roofline chapter previews Chapter 11.

Hugging Face, *The Ultra-Scale Playbook*

Its data-parallelism chapter and its activation-memory chapter, which are Sections 8.3 and 8.4 with measurements attached. Useful for the C-challenge's multi-axis picture.

Measurement

A capability that appears suddenly, a benchmark score that leaps — these are among the most misread phenomena in the field. Two small calculations show how much of the drama lives in the ruler rather than the model.

Here is a claim you will meet often: a model was scaled up, and at some size an ability it did not have simply appeared. It is a striking claim, and sometimes true. But before you believe it about any particular benchmark, there are two calculations to run — and they will frequently show that what changed was not the model but the ruler held against it.

Two calculations do the work. The first shows how a smoothly improving model can produce a sharp jump on a benchmark that scores exact matches. The second shows how contamination — test items seen during training — inflates observed accuracy, and how to invert the inflation. Neither requires anything past Chapter 1.

9.1 Emergence as a property of the ruler

Suppose a model performs a task with k independent parts — adding two k-digit numbers, say — and gets each part right with probability p. If the benchmark scores an *exact match*, awarding credit only when all k parts are correct, then the measured accuracy is p^k. Now watch what a smooth improvement in p does to p^k.

Do this one on paper as you read. Pick a per-part accuracy of 0.9 and eight parts. Before computing, write down whether you expect the exact-match score to be closer to 0.9 or closer to 0.5. Nearly everyone writes something in the eighties or nineties, because 0.9 feels high and eight feels small. The answer is 0.43, and the size of that surprise is precisely the size of the illusion this section is about.

DERIVATION 9.1

A cliff manufactured from a slope

Exact-match accuracy is p^k. As a function of the underlying per-part skill p, this is smooth — but for large k it is nearly flat at low p and rises steeply near $p = 1$, because raising a number just below one to a high power collapses it, while raising a number near one preserves it. The 50% point sits where $p^k = \frac{1}{2}$, i.e. at $p = 2^{-1/k}$, which increases toward 1 as k grows.

Concretely, take $k = 8$. At $p = 0.67$, exact-match accuracy is $0.67^8 = e^{8 \ln 0.67} = e^{-3.20} = 0.041$ — about 4%. At $p = 0.90$, it is $0.90^8 = e^{-0.843} = 0.43$ — about 43%. A modest, smooth improvement in per-digit skill, from 0.67 to 0.90, produces a tenfold leap on the benchmark. The model got steadily better; the benchmark reported a sudden emergence.

WORKED EXAMPLE 9.1

Where the cliff appears to fall

Tabulate exact-match accuracy p^k at $k = 8$ as the per-part skill improves:

```
p = 0.50 → 0.004   ·   0.70 → 0.058   ·   0.80 → 0.168   ·
0.90 → 0.430   ·   0.95 → 0.663
```

Between $p = 0.5$ and $p = 0.7$ the benchmark barely moves — from under half a percent to under six. Between 0.8 and 0.95 it moves from 17% to 66%. An observer watching only this metric would report a capability that lay dormant and then arrived; an observer watching p would report steady, unremarkable progress. The apparent onset sits near the 50% frontier $2^{-1/8} = 0.917$, and for larger k it sits closer to 1 and looks sharper still.

TRAP · T4 · METRIC VERSUS CAPABILITY

An abrupt jump in a thresholded score is *not* evidence of a discontinuous change in the underlying capability. The per-part skill p — visible in a smooth metric like per-token log-loss — may be improving perfectly steadily while the exact-match metric snaps upward. Before calling a capability "emergent," ask whether the ruler is a threshold on many smoothly-improving parts. Much reported emergence survives this test poorly.

DRILL 9.1

A model does k-digit exact-match addition with per-digit accuracy $p = 0.98$. What is the largest k for which exact-match accuracy stays at or above 50%?

Show answer

Need $0.98^k \geq 0.5$, i.e. $k \leq \ln(0.5)/\ln(0.98) = (-0.693)/(-0.0202) = 34.3$. Largest integer $k = 34$.

9.2 Contamination: when the benchmark has been seen

A subtler distortion is contamination: some fraction of the test items appeared in the training data, so the model answers them from memory rather than skill. This inflates the observed score, and a two-line model lets us both quantify and invert the inflation.

DERIVATION 9.2

Solving for the leak, then flipping it with Bayes

Let a fraction c of test items be contaminated — answered correctly with probability 1 — while on the clean fraction $1 - c$ the model's true accuracy is p. The observed accuracy is the mixture

$$\text{observed} = c \cdot 1 + (1 - c)\, p = c + p(1 - c).$$

Given an observed score and a known (or assumed) clean accuracy, solve for c. Suppose observed $= 0.70$ and clean accuracy $p = 0.60$: then $0.70 = c + 0.60(1 - c) = 0.60 + 0.40c$, so $c = 0.25$. A quarter of the benchmark leaked, and the observed score overstates true skill by ten points.

Now the more revealing question: given that an item was answered correctly, what is the probability it was contaminated? By Bayes' rule, contaminated items contribute $c \times 1$ to the correct-answer mass, out of a total correct mass equal to the observed accuracy:

$$P(\text{contaminated} \mid \text{correct}) = \frac{c}{\text{observed}} = \frac{0.25}{0.70} = 0.357.$$

Over a third of the model's apparent successes are memory, not skill. This is the calculation to run whenever a headline score seems too good — and it is nothing more than the Bayes flip of Chapter 1.

DRILL 9.2

A benchmark shows observed accuracy 0.85; the model's clean accuracy is believed to be 0.75. (a) What contamination fraction c explains the gap? (b) Given a correct answer, what is the probability it was contaminated?

Show answers

(a) $0.85 = c + 0.75(1 - c) = 0.75 + 0.25c \Rightarrow c = 0.40$. (b) $P = c/\text{observed} = 0.40/0.85 = 0.47$.

9.3 Voting, briefly

One more measurement idea belongs here, though Chapter 15 develops it fully: taking the majority vote of several independent attempts. We met the calculation in Chapter 1 — five chains at $p = 0.6$ vote correctly with probability 0.68. The one fact to carry forward is that voting helps only when the single-attempt probability already exceeds one half; below that threshold, the majority amplifies the error instead of correcting it. Voting sharpens a capability that is already there; it does not manufacture one. That distinction — sharpening versus creating — is exactly the metric-versus-capability theme of this chapter in another guise.

9.4 What the chapter bought

A benchmark can mislead in two quite different ways, and this chapter priced both. Exact-match scoring turns a smooth improvement in per-part skill into an apparent cliff, and the 50% frontier $p = 2^{-1/k}$ tells you where the cliff will seem to fall. Contamination inflates a score by $c + p(1 - c)$, and Bayes' rule reveals how much of the model's success is memory. Both are warnings to measure the ruler before trusting the miracle — a habit that separates careful readers of the field from credulous ones. Chapter 10 closes Part II by descending to the idea beneath all of this: that learning is compression, that simplicity is a probability, and that a single length prior turns Occam's razor from a preference into a theorem.

Exercises

A · DRILLS

1. A task has $k = 5$ independent parts, each correct with $p = 0.9$. Give the exact-match accuracy.
2. At what per-part accuracy p does a $k = 10$ exact-match task reach 50%?
3. Observed accuracy 0.72, clean accuracy 0.60. Find the contamination fraction.
4. With that contamination, what fraction of correct answers are contaminated?

B · PROBLEMS

1. **The shape of emergence.** For an exact-match task with $k = 8$, tabulate accuracy at $p = 0.5, 0.7, 0.8, 0.9, 0.95$. Identify where the steepest rise occurs and relate it to the 50% frontier $p = 2^{-1/k}$. Explain in two sentences why a per-token log-loss curve would show no such cliff.

2. **Inverting a leak.** A model reports 0.90 on a benchmark. Independent evidence suggests its true clean accuracy is 0.65. Find the implied contamination fraction, and the probability that a given correct answer came from contamination. Comment on what these numbers imply for trusting the headline score.

3. **Voting's threshold.** Show, for majority voting over three independent attempts, that the majority is more accurate than a single attempt if and only if the single-attempt accuracy exceeds $\frac{1}{2}$. (Compute the majority accuracy $3p^2(1-p) + p^3$ and compare to p.)

C · CHALLENGE

1. **Two illusions, one theme.** Emergence (Section 9.1), contamination (Section 9.2), and voting (Section 9.3) are all instances of a single caution: the number you observe is a function of both the model and the measurement, and confusing the two misleads. For each of the three, identify precisely what belongs to the model, what belongs to the ruler, and what error you make if you attribute the ruler's contribution to the model. (This is trap class T4 stated in full generality.)

GATE 9 · PASS BEFORE CHAPTER 10

Reproduce Derivations 8.1 (the p^k cliff and its 50% frontier) and 8.2 (the contamination mixture and its Bayes flip) on blank paper. Attempt B-1 and B-2 closed book. You pass when you can produce the $0.67 \to 0.90$ becomes $4\% \to 43\%$ figures and the $c/\text{observed}$ Bayes ratio without notes.

READINGS FOR CHAPTER 9

Wei et al., *Emergent Abilities of Large Language Models* (2022)

The case for emergence, in its eight-benchmark figure. Read those curves as the claim Derivation 9.1 puts to the test, and decide for each panel whether the metric is a threshold.

Schaeffer, Miranda & Koyejo, *Are Emergent Abilities of Large Language Models a Mirage?* (2023)

The rebuttal, and its metric-swap figure is the whole argument: the same models, scored continuously, show no cliff. Read the two papers as a debate you can now adjudicate.

Dodge et al., *Documenting the English Colossal Clean Crawled Corpus* (2021)

The provenance table, for what actually ends up in a training corpus — the empirical face of the contamination fraction c in Derivation 9.2.

Zhang et al., *Understanding Deep Learning Requires Rethinking Generalization* (2016)

Networks can memorize random labels; the backdrop against which contamination becomes a serious worry, and a bridge to Chapter 10.

Compression and Occam

Underneath every result so far lies one currency: description length. Learning is compression, simplicity is a probability, and Occam's razor is not a preference but a theorem.

"A model that predicts well compresses well." That sounds like a metaphor, or at best an analogy. It is neither. It is an identity, and you can compute both sides of it. Once you take it literally, a surprising amount of what the earlier chapters treated separately — overfitting, generalization, why simple explanations deserve preference — turns out to be one subject measured in one unit.

The organizing principle has one sentence: **total code length equals model bits plus residual bits, and the smallest total wins.** Everything in this chapter is that sentence in a different costume — selecting among models, weighing two hypotheses, pricing the precision of a weight, reading a cross-entropy as a file size, or assigning belief to competing programs. We take them in turn.

10.1 Minimum description length

To communicate a dataset to someone who has neither, you may send a model and then the residuals — the corrections the model fails to predict. A complex model costs more to send but leaves smaller residuals; a simple one costs little but leaves much unexplained. The *minimum description length* principle says: choose the model minimizing the total.

WORKED EXAMPLE 10.1

The zero-error model loses

Each parameter costs 12 bits to encode. Three candidates fit the same data:

- **Model A:** 3 parameters, residuals need 480 bits. Total $36 + 480 = 516$.
- **Model B:** 20 parameters, residuals need 190 bits. Total $240 + 190 = \mathbf{430}$.
- **Model C:** 60 parameters, residuals need 0 bits — it fits the training data perfectly. Total $720 + 0 = 720$.

MDL selects **B** at 430 bits. Note what happened to Model C: it achieves zero training error and is the *worst* of the three, losing by 290 bits. Overfitting, usually described in words, is here a number — the price of the parameters exceeds the residual bits they save.

The same accounting compares two hypotheses directly. If H_1 needs 30 bits to specify and assigns the observed data probability 2^{-10}, its total is $30 + 10 = 40$ bits. If H_2 needs only 12 bits but assigns probability 2^{-35}, its total is 47. The *simpler* hypothesis loses by 7 bits — odds of $2^7 = 128$ to 1 against it. Simplicity is charged in the same currency as fit, and the sum decides.

DRILL 10.1

Hypothesis A is a 15-bit program with likelihood 2^{-4}. Hypothesis B is a 5-bit program with likelihood 2^{-k}. What is the largest integer k for which B is still preferred?

Show answer

A totals $15 + 4 = 19$. B totals $5 + k$, which must be under 19, so $k \leq 13$.

10.2 Cross-entropy is a file size

The connection between prediction and compression is not a metaphor. When a model assigns probability q to the token that actually occurs, an optimal code spends $-\log_2 q$ bits encoding it. The average of that quantity over a corpus is the

cross-entropy, so a model's cross-entropy in bits per token is *literally* the per-token size of the compressed corpus.

> **WORKED EXAMPLE 10.2**
>
> **A corpus, in gigabytes**
>
> A corpus of 23 billion tokens is modeled at a cross-entropy of 1.6 bits per token. The compressed size is $23 \times 10^9 \times 1.6 = 36.8 \times 10^9$ bits, and dividing by 8×10^9 bits per gigabyte gives **4.6 GB**. Improving the model to 1.4 bits per token would shrink the same corpus to 4.0 GB. Every gain in predictive quality is, exactly and measurably, a gain in compression.

One conversion has to be made explicit, because it is where the identity meets a real corpus and where nearly everyone slips. A model's loss is quoted per *token*, and a file's size is measured in *characters* or bytes. The two differ by the tokenizer's compression ratio γ, the average number of characters a token stands for — Section 2.4, and about 4 for English. Bits per character is bits per token divided by γ, and if you compare two models whose tokenizers differ without making that division, you are comparing rulers rather than models.

> **WORKED EXAMPLE 10.3**
>
> **Two models, two tokenizers, one file**
>
> Model P reaches 1.8 bits per token at $\gamma = 4.2$ characters per token. Model Q reaches 1.4 bits per token at $\gamma = 3.0$. Which compresses a 1 GB text file smaller?
>
> Per character, P spends $1.8/4.2 = 0.429$ bits and Q spends $1.4/3.0 = 0.467$ bits. On 10^9 characters that is 4.29×10^8 bits $= 53.6$ MB for P against 4.67×10^8 bits $= 58.4$ MB for Q.
>
> P wins, though its per-token loss is the worse of the two by a wide margin. The comparison people actually make — 1.4 against 1.8 — gets the answer backwards, because a token is not a fixed amount of text. This is trap class T4 wearing a compression costume: the number moved, but so did the ruler.

DRILL 10.2

(a) A model scores 1.6 bits per token at $\gamma = 3.8$. Give its bits per character, and the compressed size of a 40 GB corpus. (b) A tokenizer is improved so γ rises from 3.8 to 4.4 while bits per token rises from 1.6 to 1.8. Did the model get better or worse at compressing text?

Show answers

(a) $1.6/3.8 = 0.421$ bits per character; $4 \times 10^{10} \times 0.421/8 = 2.11 \times 10^{9}$ bytes, about 2.1 GB. (b) Better: $1.8/4.4 = 0.409$ bits per character against 0.421. The per-token figure got worse and the thing that matters got better, because each token now carries more text.

10.3 Paying for precision

A subtler costume: how many bits does a single weight cost? If we transmit weights not as exact numbers but as a distribution — a posterior $\mathcal{N}(\mu, \sigma_q^2)$ against a shared prior $\mathcal{N}(0, \sigma_p^2)$ — the cost is the Kullback–Leibler divergence between them,

$$D_{\mathrm{KL}} = \ln \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + \mu^2}{2\sigma_p^2} - \frac{1}{2} \quad \text{nats.}$$

Three readings of this formula matter more than the formula itself. When the posterior equals the prior ($\sigma_q = \sigma_p = 1, \mu = 0$), the cost is exactly zero: a weight you have learned nothing about is free. As $\sigma_q \to 0$ the first term diverges: *infinite precision costs infinite bits*. And a weight the network is content to leave noisy — large σ_q — is a *cheap* weight.

WORKED EXAMPLE 10.4

Three weights, three prices

Take a shared prior of $\sigma_p = 1$ and price three weights the network might want.

One it has learned nothing about ($\mu = 0$, $\sigma_q = 1$): $\ln 1 + \frac{1+0}{2} - \frac{1}{2} = 0$ nats. Free, exactly.

One it is mildly confident about ($\mu = 0.5$, $\sigma_q = 0.5$): $\ln 2 + \frac{0.25+0.25}{2} - \frac{1}{2} = 0.693 + 0.25 - 0.5 = 0.443$ nats.

One it insists on ($\mu = 0.5$, $\sigma_q = 0.01$): $\ln 100 + \frac{0.0001+0.25}{2} - \frac{1}{2} = 4.605 + 0.125 - 0.5 = 4.23$ nats — nearly ten times the mild one, for a weight in the same place with the same mean.

The whole bill is in the first term, $\ln(\sigma_p/\sigma_q)$, and it diverges as $\sigma_q \to 0$. Precision, not magnitude, is what a description-length penalty charges for, which is why such a penalty produces a model that is confident about a few things and vague about most.

TRAP · T5 · DIRECTION REVERSAL

Students routinely get this backwards, assuming a noisy weight must be costly. The reverse is true, and the reversal *is* the mechanism: because precision costs bits, a network under a description-length penalty buys precision only where precision buys accuracy, and lets every other weight stay blurry. That is how such a penalty produces simple models — not by removing weights, but by declining to pay for their exactness.

DRILL 10.3

Compute the coding cost for $\sigma_p = 1$, $\sigma_q = 0.1$, $\mu = 0.5$, in nats and in bits. (Use $\ln 10 = 2.303$, and 1 nat = 1.4427 bits.)

Show answer

$\ln(1/0.1) + \frac{0.01+0.25}{2} - 0.5 = 2.303 + 0.13 - 0.5 = 1.933$ nats $= 1.933 \times 1.4427 = 2.79$ bits.

10.4 Occam's razor as a theorem

Now the deepest costume. Chapter 1 introduced the Kraft inequality: a set of prefix-free descriptions with lengths ℓ_i satisfies $\sum_i 2^{-\ell_i} \le 1$. Read in reverse, this licenses treating $2^{-\ell}$ as a *probability* assigned to a description of length ℓ. That single move converts a preference for simplicity into a consequence of probability theory.

Here is the step that surprises people. We are about to conclude that shorter explanations are more probable — not as a working assumption, not as a scientific convention, but as an arithmetic consequence. If that sounds like it must be smuggling something in, look carefully at what actually gets assumed. It is only that descriptions are written in a code no description of which is a prefix of another. Everything else follows.

DERIVATION 10.1

Every extra bit halves the vote

Suppose several programs are each perfectly consistent with the data observed so far, so their likelihoods are all 1. Under the prior $P(p) = 2^{-\ell(p)}$, their posterior weights are proportional to $2^{-\ell}$. Comparing two programs of lengths $\ell_1 < \ell_2$, their posterior ratio is $2^{\ell_2 - \ell_1}$: *each additional bit of description length halves a program's share of belief.* No preference for simplicity was assumed; it fell out of the length prior.

To predict, mix the programs by their weights. Take three consistent programs of lengths 8, 12, and 20 bits, predicting the next bit as 1, 0, and 1 respectively. Scale all weights by 2^{20} to work in integers: they become $2^{12} = 4096$, $2^{8} = 256$, and 1, totaling 4353. The programs predicting 1 hold $4096 + 1 = 4097$, so

$$P(\text{next bit} = 1) = \frac{4097}{4353} = 0.941.$$

The 20-bit program holds $1/4353 = 0.023\%$ of the belief — twelve extra bits have reduced it by a factor of 4096. Equal fit does not mean equal belief.

DRILL 10.4

In the setting above, suppose the shortest program were 16 bits instead of 8 (others unchanged). Recompute $P(\text{next bit} = 1)$.

Show answer

Weights scaled by 2^{20}: $2^4 = 16$, $2^8 = 256$, 1; total 273. Programs predicting 1 hold $16 + 1 = 17$, so $P = 17/273 = 0.062$ — now below one half, because the 12-bit program predicting 0 has become the shortest.

10.5 Typicality is not membership

A short but sharp idea. Consider the set M of all 10,000-bit strings containing exactly 5,000 ones. Its size is $\binom{10000}{5000}$, and by the approximation of Chapter 1, $\log_2 \binom{2n}{n} \approx 2n - \frac{1}{2}\log_2(\pi n)$, which here gives about **9,993** bits: specifying a member takes nearly the full 10,000 bits, so a typical member is essentially incompressible.

Now take the alternating string $0101\ldots01$. It belongs to M — it has exactly 5,000 ones. Yet its description length is tiny: "print 01 five thousand times" is a few dozen bits. Its *randomness deficiency* — the gap between the bits a typical member needs and the bits this member actually needs — is nearly maximal, about 9,993 bits. The string satisfies every constraint the model imposes and is nonetheless an outrageous thing for that model to produce. Membership in a model is not typicality under it, and conflating the two is how one fails to notice a black swan.

WORKED EXAMPLE 10.5

A member that no model would produce

Take M, the set of 10,000-bit strings with exactly 5,000 ones, and compare two of its members.

A typical member: by Chapter 1's approximation, $\log_2 \binom{10000}{5000} \approx 2n - \frac{1}{2}\log_2(\pi n)$ with $n = 5000$, giving $10{,}000 - \frac{1}{2}\log_2(15{,}708) = 10{,}000 - 6.97 \approx 9{,}993$ bits. Specifying it takes essentially the full 10,000 bits; it is incompressible.

The alternating string $0101\ldots01$: also in M, and describable in a few dozen bits. Its randomness deficiency — what a typical member costs minus what this one costs — is about $9{,}993$ bits, very nearly the maximum possible.

Both satisfy every constraint the model imposes. One is what the model is about; the other is an outrage the model cannot rule out. Membership is not typicality, and a model that only checks membership has no way to notice the difference.

10.6 Two puzzles the framework dissolves

Grokking. A network trained on a rule-like task sometimes memorizes first and only much later — abruptly — discovers the rule that generalizes. In code-length terms this is unmysterious. The arithmetic is in the box below. Both solutions sit at zero training error, so *the training loss is silent about the transition* — but the description length is not, and neither is the test accuracy. Delayed generalization is a search finding the short code late.

WORKED EXAMPLE 10.6

Grokking, in bits

A network memorizes 1,000 training examples at 10 bits each: $10{,}000$ bits of model. Later it finds the rule that generates them, which costs 500 bits. Both encodings leave zero residual — both fit the training data exactly — so the training loss cannot tell them apart, and does not.

$$\frac{10{,}000}{500} = 20\times \text{ compression, at unchanged training error.}$$

The description length fell by a factor of twenty while the quantity being optimized did not move at all. That is why the transition looks sudden and unmotivated on a loss curve and is neither: the search was always running, over a space in which short codes are rare and therefore found late.

Double descent. Classical wisdom says test error traces a U as model size grows. Modern practice sees it fall, rise to a peak, and fall again past the point where the model can interpolate the training data. The peak is quantifiable. For least squares with p parameters and n samples ($p < n - 1$) at noise variance σ^2, the expected excess test risk is $\sigma^2 p/(n - p - 1)$ — take this as given. As $p \to n - 1$ the denominator vanishes and the risk diverges: at the interpolation threshold there is effectively one interpolating solution, and it is maximally sensitive to noise. Past the threshold many interpolating solutions exist, an optimizer biased toward small norm can select a simple one, and the risk falls again.

WORKED EXAMPLE 10.7

The approach to the threshold

With $n = 100$ samples and $\sigma^2 = 1$, compare the excess risk at $p = 50$ and $p = 90$. At $p = 50$: $50/(100 - 50 - 1) = 50/49 = 1.02$. At $p = 90$: $90/(100 - 90 - 1) = 90/9 = 10$. The ratio is **9.8**. Adding parameters near the threshold is not mildly harmful; it is catastrophic — and yet pushing further past it recovers.

10.7 Compression and the measure of intelligence

The chapter's last costume applies description length to agents rather than models. If we score an agent's performance $V_i \in [0,1]$ across many environments, how should the environments be weighted? Weighting each by 2^{-K_i}, where K_i is the environment's description complexity, gives

$$\Upsilon = \sum_i 2^{-K_i} V_i .$$

Three properties follow immediately. By Kraft, $\Upsilon \le 1$ for any agent. An agent competent only in very complex environments scores near zero, because 2^{-K} decimates them: *Occam is built into the definition of intelligence itself.* And adding one bit to every environment's complexity halves Υ exactly.

> **WORKED EXAMPLE 10.8**
>
> **Scoring an agent, and a caution about rewards**
>
> An agent scores $V = 0.5, 0.9, 0.3$ in environments of complexity $K = 1, 2, 3$ bits. Then $\Upsilon = \tfrac{1}{2}(0.5) + \tfrac{1}{4}(0.9) + \tfrac{1}{8}(0.3) = 0.25 + 0.225 + 0.0375 = \mathbf{0.51}$.
>
> A related calculation, using Chapter 1's geometric series, prices a temptation. An agent discounting the future at $\gamma = 0.95$ values a perpetual per-step reward r at $\gamma r/(1-\gamma) = 19r$. This beats a one-time reward of 1 whenever $r > 1/19 = \mathbf{0.053}$. A trickle worth barely five percent of the one-off prize is already preferable — which is why an agent able to seize its own reward channel finds doing so overwhelmingly attractive, and why the reward source must be kept outside the agent's control.

10.8 What the chapter bought

Everything above was the same sentence in different clothes: total code length decides. It selects models and exposes overfitting as arithmetic; it makes cross-entropy a literal file size; it prices a weight's precision and thereby explains how simplicity is bought; it converts Occam's razor from a preference into a theorem via the length prior; it distinguishes membership from typicality; and it dissolves grokking and double descent into statements about which code is short

and how many solutions exist. Part II ends here. Part III turns to the modern era, and it opens with a shift in what is scarce: not the compute to train a model, but the bytes and bandwidth to serve one.

Exercises

A · DRILLS

1. Four prefix-free programs have lengths 1, 2, 3, 3 bits. What total prior mass do they carry, and what does that tell you about the code?
2. A corpus of 8 billion tokens is modeled at 2.0 bits/token. Give the compressed size in GB.
3. Two hypotheses total 22 and 29 bits. By what odds is the first preferred?
4. Compute Υ for $V = (1.0, 0.5)$ in environments of complexity $K = (2, 3)$.

B · PROBLEMS

1. **Model selection, fully.** Each parameter costs 9 bits. Model P: 5 parameters, 300 bits of residual. Model Q: 25 parameters, 95 bits. Model R: 50 parameters, 0 bits. Give all three totals, name the MDL choice, and state by how many bits the zero-error model loses. Explain in one sentence what quantity overfitting corresponds to here.
2. **The length prior in action.** Four consistent programs have lengths 6, 9, 9, and 14 bits, predicting the next symbol as A, B, A, and B. Compute the posterior probability of A. Then state the general rule relating a program's extra length to its share of belief, and prove it in one line.
3. **Approaching the threshold.** With $n = 200$, $\sigma^2 = 1$, tabulate the excess risk at $p = 100, 150, 190, 195$. Describe the trend and say what it implies about model sizes near $p \approx n$.
4. **Precision is what costs.** Using the Gaussian coding formula, compute the cost of a weight at $\sigma_p = 1$ with (a) $\sigma_q = 0.5, \mu = 0$; (b) $\sigma_q = 0.05, \mu = 0$. Explain what the comparison says about which weights a description-length penalty leaves imprecise.
5. **The same corpus, three tokenizers.** A model reaches 1.9 bits per token at $\gamma = 4.5$, a second 1.5 at $\gamma = 3.2$, a third 2.4 at $\gamma = 6.0$. Rank them as compressors of

the same file, give each one's compressed size for 5 GB of text, and state which comparison a per-token leaderboard would get wrong.

6. **What a bit of precision is worth.** Using the Gaussian coding formula at $\sigma_p = 1$, $\mu = 0$, tabulate the cost of a weight at $\sigma_q = 1, 0.5, 0.25, 0.1, 0.01$. Show that each halving of σ_q adds $\ln 2$ nats once σ_q is small, and explain in one sentence why that makes precision a *linear* cost in bits of precision rather than a diminishing one.

C · CHALLENGE

1. **One currency, eight costumes.** This chapter presented model selection, hypothesis odds, weight precision, cross-entropy-as-file-size, the length prior, typicality, grokking, and double descent. For each, state precisely what plays the role of "model bits" and what plays the role of "residual bits," or explain why one of the two is absent. Then argue in a paragraph why a single principle can wear this many disguises — and what that suggests about how to read a new result in the field.

GATE 10 · PASS BEFORE CHAPTER 11 — AND BEFORE PART III

Reproduce Derivation 10.1 (the length prior, its halving rule, and a mixture prediction) on blank paper. Work Worked Example 10.1 from a blank table and B-1 closed book. Then, as the Part II capstone, sweep the whole Derivation Bank so far — every result from Chapters 3 through 10 — reproducing each from nothing. Part III assumes all of it.

READINGS FOR CHAPTER 10

Grünwald, *A Tutorial Introduction to the Minimum Description Length Principle* (2004)

Sections 1–2. The polynomial-fitting figure is the ancestor of Worked Example 10.1.

Hinton & van Camp, *Keeping Neural Networks Simple by Minimizing the Description Length of the Weights* (1993)

The origin of Section 10.3; read it as the paper that priced precision in bits.

Power et al., *Grokking: Generalization Beyond Overfitting* (2022)

The phenomenon of Section 10.6; note that the training curve is flat exactly as the code-length account predicts.

Nakkiran et al., *Deep Double Descent* (2019) and Belkin et al., *Reconciling Modern Machine Learning and the Bias–Variance Trade-off* (2018)

The model-wise double-descent figure. Check where its peak sits against your $\sigma^2 p/(n-p-1)$, and note that the peak is at $p \approx n$ exactly as the risk formula says.

Legg, *Machine Super Intelligence* (2008)

Chapters 2–4 for the measure of Section 10.7 and the reward-channel argument.

Chiang, *ChatGPT Is a Blurry JPEG of the Web* (2023)

Four pages, read last. Then write one paragraph on where the compression metaphor is right and where Section 10.2 corrects it — the essay treats compression as a loss of fidelity, and this chapter treats it as the objective.

PART III

The Modern Era

The Price of a Token

Training cost was a count of operations. Serving cost is something else entirely: generating one token requires reading every weight in the model, which makes decoding a memory problem wearing a compute costume.

Ask what limits how fast a large model can write text, and the natural answer is arithmetic: it must do a great many multiplications per word. The natural answer is wrong by a factor of about a hundred and fifty. Generating a single token requires reading every weight in the model out of memory, and it is that reading, not the arithmetic, that sets the pace.

The reason is that serving splits into two regimes that behave nothing alike. **Prefill** handles the whole prompt at once: large matrices meet large matrices, and the arithmetic units are genuinely the constraint. **Decode** emits one token at a time, so a matrix meets a single vector, and the entire parameter store must stream out of memory to support a comparatively tiny amount of useful multiplication. That second regime is memory-bound, and nearly every serving optimization of recent years is a response to it.

> REFERENCE HARDWARE
>
> **The accelerator we will cost everything against**
>
> Throughout Part III, take a reference accelerator with **312 TFLOP/s** of half-precision compute, **2 TB/s** of memory bandwidth, and about 20 MB of fast on-chip memory. Half precision means 2 bytes per value. These are given constants, like a beam's Young's modulus — you never memorize them, you are handed them and you compute.

11.1 The key–value cache

Attention at position t compares the current query against the keys of all previous positions and mixes their values. Recomputing those keys and values at every step would be quadratic waste, so they are stored — the *KV cache*. Its size is pure bookkeeping, and it is the first thing to compute about any serving setup.

DERIVATION 11.1

The size of the cache

Per layer, per attention head that owns its own keys and values, and per sequence position, we store one key vector and one value vector, each of dimension d_{head}. Hence for L layers, H_{kv} key/value heads, sequence length s, batch B, and β bytes per value:

$$\text{cache bytes} = 2 \cdot L \cdot H_{kv} \cdot d_{\text{head}} \cdot s \cdot B \cdot \beta.$$

The leading 2 counts keys and values separately — the single most commonly dropped factor in this calculation. Note that the cache is *linear in sequence length*: it is a per-token cost that accumulates, not a fixed overhead.

WORKED EXAMPLE 11.1

A cache that outgrows the model

Take a 7-billion-parameter model with $L = 32$ layers, 32 heads of dimension 128, half precision, at context $s = 4096$, batch 1.

Work the per-token figure first, since that is the quantity that accumulates. Per position the cache holds $2 \times 32 \times 32 \times 128 \times 2 = 524{,}288$ bytes — about **0.5 MB per token**. Over the full context: $524{,}288 \times 4096 =$ **2.15 GB**.

The weights themselves are $7 \times 10^9 \times 2 = 14$ GB. At batch 1 the cache is modest — but at batch 8 it becomes 17 GB and *exceeds the weights*. This, not arithmetic, is why long-context serving is expensive.

Now suppose the model shares keys and values across groups of heads, so that $H_{kv} = 8$ rather than 32. The cache falls by a factor of four, to **0.54 GB** — the entire motivation for grouped-query attention.

DRILL 11.1

A 70B model has $L = 80$ layers, $H_{kv} = 8$, $d_{\text{head}} = 128$, half precision. (a) Give the cache bytes per token. (b) At batch 4, what context length fills 40 GB?

Show answers

(a) $2 \times 80 \times 8 \times 128 \times 2 = 327{,}680$ bytes, about 0.33 MB per token. (b) $40 \times 10^9 / (4 \times 3.28 \times 10^5) \approx 30{,}500$ tokens.

11.2 The roofline: which resource is binding

Whether a computation is limited by arithmetic or by memory is settled by comparing two ratios. The *arithmetic intensity* of a computation is the number of operations it performs per byte it moves. The *ridge point* of a machine is its peak operation rate divided by its bandwidth. Below the ridge, you are memory-bound; above it, compute-bound.

Try predicting the number first. On a machine that can do three hundred trillion operations per second, what fraction of that capacity do you suppose a large model uses while writing text one word at a time? Most people guess something like half, or a quarter if they are being pessimistic. The real figure is below one percent, and seeing exactly why will change how you read every performance claim in this field.

DERIVATION 11.2

Why decode sits far below the ridge

Our reference machine has ridge point $312 \times 10^{12} / 2 \times 10^{12} = \mathbf{156}$ operations per byte.

A large square matrix multiply performs n^3 operations while moving about $3n^2$ values, an intensity of roughly $n/3$ — for large n this sits comfortably above the ridge, so prefill is compute-bound.

Batch-1 decode is the opposite. Producing one token performs about $2N$ operations (one multiply–add per parameter) while moving about $2N$ bytes (reading every parameter once in half precision). The intensity is about **1 operation per byte** — a factor of 156 below the ridge. Decoding uses well under one percent of the machine's arithmetic capability.

Batching B requests reuses each weight read B times, so the intensity rises to roughly B. You approach compute-bound only near $B \approx 156$ — which is precisely why serving systems are obsessed with batch size.

WORKED EXAMPLE 11.2

The decode speed limit

A 7B model in half precision holds 14 GB of weights, all of which must be read to emit one token. The ceiling is therefore

$$\frac{2\ \text{TB/s}}{14\ \text{GB}} = \frac{2 \times 10^{12}}{1.4 \times 10^{10}} \approx \mathbf{143\ tokens\ per\ second},$$

no matter how fast the arithmetic units are. Doubling the machine's FLOP rate changes nothing; halving the bytes per weight doubles the ceiling. This single division reframes what "a faster model" means.

TRAP · T7 · RATE VERSUS STOCK, AND T9 · NAME YOUR BUDGET

Two errors to guard against. First, cache figures are often quoted per token and often quoted as totals, and the two differ by a factor of the context length; always state which you mean. Second — and this is a habit for all of Part III — every efficiency claim must name the budget it improves. Grouped-query attention shrinks the *cache*, not the weight read, so it helps long-context and large-batch serving but does *not* raise the batch-1 decode ceiling. "It makes it faster" without naming a budget is not an answer.

11.3 Utilization, honestly measured

A final piece of accounting: what fraction of a machine's capability is a workload actually using? For training, the model FLOP utilization is the useful operations per second divided by peak — with $6N$ operations per token from Chapter 7,

$$\text{MFU} = \frac{\text{tokens/s} \times 6N}{\text{peak FLOP/s}}.$$

A well-tuned training run reaches a substantial fraction. Decode, as Derivation 11.2 showed, cannot: its utilization is structurally terrible, because it is not limited by the resource the denominator measures. Reporting decode MFU without saying so is a way of making a memory-bound workload look like an engineering failure rather than a physical one.

DRILL 11.2

A training run sustains 4200 tokens/s per accelerator on a 7B model. What is its MFU on the reference machine?

Show answer

$4200 \times 6 \times 7 \times 10^9 = 1.76 \times 10^{14}$ FLOP/s $= 176$ TFLOP/s. Divided by 312: **56%**.

11.4 What the chapter bought

You can now price a deployed model with three formulas, and none of them counts a floating-point operation. The cache size, linear in context and often larger than the weights. The roofline, which places batch-1 decode a hundred and fifty times below the point where arithmetic matters. And the decode ceiling, bandwidth divided by weight bytes, which says what a model can possibly emit per second. Together they explain why the modern era's optimizations target bytes rather than operations — and Chapter 12 takes up the two most elegant examples: an attention kernel that does more arithmetic to move less memory, and a decoding scheme that buys parallelism from a cheap draft model while provably changing nothing about the output.

Exercises

A · DRILLS

1. A model has $L = 48$, $H_{kv} = 16$, $d_{\text{head}} = 128$, half precision. Give the cache bytes per token.
2. Give the ridge point of a machine with 500 TFLOP/s and 3 TB/s.
3. A 13B model in half precision: give the batch-1 decode ceiling on the reference machine.
4. The same 13B model quantized to 4 bits per weight: give the new ceiling.

B · PROBLEMS

1. **Sizing a deployment.** A 70B model in half precision serves at context 8192, batch 16, with $L = 80$, $H_{kv} = 8$, $d_{\text{head}} = 128$. Give (a) the weight memory, (b) the total KV cache, (c) their sum, and (d) the number of 80 GB accelerators required. State which term you would attack first and why.

2. **The batching crossover.** Derive the arithmetic intensity of batch-B decode and show it is approximately B. Find the batch size at which the reference machine becomes compute-bound, and explain what happens to per-request latency as batch size approaches it.

3. **What quantization actually buys.** For a 70B model, give the decode ceiling at 16, 8, and 4 bits per weight. Then state precisely which budget quantization improves and which it leaves untouched, and use that to explain why quantization speeds up decode but not prefill.

C · CHALLENGE

1. **Two eras, two scarcities.** Chapter 7 minimized loss subject to a budget of operations; this chapter argues bytes are what bind at serving time. Construct a scenario in which the compute-optimal model of Chapter 7 is the *wrong* model to deploy, quantify the disagreement using this chapter's formulas, and state what additional term the objective would need for the two chapters to agree. (You are deriving, in outline, the argument Chapter 17 completes.)

GATE 11 · PASS BEFORE CHAPTER 12

Reproduce Derivations 10.1 (cache size, with the factor of 2 and the right head count) and 10.2 (intensities and the ridge point) on blank paper, and produce the 143-tokens-per-second ceiling from nothing. Attempt B-1 and B-3 closed book. You pass when you instinctively ask “which budget?” of every optimization claim.

READINGS FOR CHAPTER 11

Kipply, *Transformer Inference Arithmetic*

Its KV-cache section and its arithmetic-intensity section are this chapter with different notation. Recompute every number in both yourself; where you disagree, one of you has dropped a factor of 2.

Pope et al., *Efficiently Scaling Transformer Inference* (2022)

Sections 2–3. The systems view of the roofline argument, with real partitioning strategies.

Ainslie et al., *GQA: Grouped-Query Attention* (2023) and Shazeer, *Fast Transformer Decoding* (2019)

The quality-versus-speed table, for what sharing keys and values across head groups costs in accuracy. Worked Example 11.1 gives the cache saving; this gives the bill.

Austin et al., *How to Scale Your Model* — inference chapter

Its roofline section, worked alongside Derivation 11.2. The best free treatment of this material, and it uses the same ridge-point argument on different hardware.

Bytes over FLOPs

Two of the era's most elegant results trade arithmetic for memory traffic, and speed for nothing at all. Both are exact: neither changes a single output.

Consider two claims that sound like they cannot both be true. The fastest known way to compute attention does *more* arithmetic than the obvious way. And there is a method that makes a large model generate text two or three times faster while provably producing exactly the same distribution of outputs it would have produced anyway. Both are true, and both follow from the previous chapter's finding about what is actually scarce.

12.1 Attention without the score matrix

Chapter 6 counted attention's arithmetic as $2n^2d$. At realistic sequence lengths, however, the binding cost is not those operations but the $n \times n$ matrix of scores travelling to and from main memory. The output of attention is only $n \times d$ values; the intermediate is n^2. That gap — $\Theta(n^2)$ traffic for a $\Theta(nd)$ result — is what an IO-aware kernel closes.

> **WORKED EXAMPLE 12.1**
>
> **The intermediate does not fit**
>
> At $n = 4096$ in half precision, one head's score matrix occupies $2n^2 = 2 \times 4096^2 = 33.6$ MB — larger than the reference accelerator's entire 20 MB of fast on-chip memory, and that is per head, per layer. You cannot keep it on chip; you either round-trip it through main memory at great cost, or you never form it at all.

Never forming it requires computing a softmax over values you see only a block at a time. That is possible, and exactly, because the softmax normalizer can be maintained incrementally.

DERIVATION 12.1

Streaming softmax

To normalize safely we subtract the running maximum before exponentiating, so nothing overflows. Maintain a running maximum m and a running sum $\ell = \sum e^{s_i - m}$ over the scores seen so far. On receiving a new block of scores $\{s_i\}$:

$$m' = \max\bigl(m,\ \max_i s_i\bigr), \qquad \ell' = \ell\, e^{\,m - m'} + \sum_i e^{\,s_i - m'}.$$

The factor $e^{m-m'}$ rescales the old partial sum into the new frame — it is never larger than 1, so the update is numerically safe. The same rescaling is applied to a running weighted sum of value vectors, so the attention output can be accumulated block by block without the score matrix ever existing in full.

Verify that this is exact. Let scores arrive as $\{2, 1\}$ then $\{3, 0\}$. After the first block, $m = 2$ and $\ell = e^{0} + e^{-1} = 1.368$. After the second, $m' = 3$ and $\ell' = 1.368\, e^{-1} + (e^{0} + e^{-3}) = 0.503 + 1.050 = 1.553$. Computing in one shot instead: $e^{-1} + e^{-2} + e^{0} + e^{-3} = 0.368 + 0.135 + 1 + 0.050 =$ **1.553**. Identical. The streaming form is not an approximation.

With this, attention is computed in tiles small enough to sit in fast memory: load a block of queries, stream blocks of keys and values past it, accumulate. Main-memory traffic falls by roughly an order of magnitude. The backward pass *recomputes* the blocks it needs rather than storing them — more arithmetic, less memory — and the kernel wins anyway. That is the roofline lesson in its purest form.

TRAP · T8 · OBJECTIVE VERSUS ESTIMATOR

This technique is *exact*. It computes the same attention function, to floating-point tolerance, as the naive implementation. Approximate attention — sparse patterns, low-rank surrogates — is a different trade entirely, buying speed with fidelity. Confusing the two is the most common misreading here: an IO-aware kernel changes *how* a quantity is computed, not *what* is computed.

DRILL 12.1

Scores arrive as $\{1, 4\}$ then $\{2, 2, 5\}$. Run the streaming update and verify against the one-shot sum.

Show answer

Block 1: $m = 4$, $\ell = e^{-3} + e^{0} = 1.050$. Block 2: $m' = 5$, $\ell' = 1.050\, e^{-1} + (e^{-3} + e^{-3} + e^{0}) = 0.386 + 1.100 = 1.486$. One shot: $e^{-4} + e^{-1} + e^{-3} + e^{-3} + e^{0} = 0.018 + 0.368 + 0.050 + 0.050 + 1 = 1.486$. ✓

12.2 Speculative decoding

The second technique attacks the same waste from a different side. Decoding is memory-bound (Chapter 11), so the accelerator's arithmetic sits idle while weights stream past. Suppose a small, cheap *draft* model proposes the next k tokens; the large *target* model can then check all k in a *single* forward pass, because verifying several positions at once is a prefill-shaped, compute-bound operation — nearly free relative to k separate decode steps.

The obvious objection is that the output would then be the draft's, not the target's. The answer is a rejection-sampling scheme that makes the final distribution *exactly* the target's.

The obvious objection comes first, and you should feel its force before seeing it dissolved. If a small, weaker model is proposing the words, surely the output is contaminated by that weaker model — faster, perhaps, but subtly worse. It is an entirely reasonable worry. It is also, as the following shows, exactly wrong: the

acceptance rule is constructed so that the weaker model's preferences cancel out of the final distribution completely.

DERIVATION 12.2

Exactness by rejection sampling

Let the target assign probability $p(x)$ to token x and the draft assign $q(x)$. Sample $x \sim q$ and accept it with probability $\min\big(1, p(x)/q(x)\big)$. If rejected, sample instead from the normalized residual $\operatorname{norm}\big(\max(0,\ p - q)\big)$.

Then the probability of emitting x is the accepted mass plus the resampled mass. The accepted mass is $q(x)\min(1, p/q) = \min(q(x), p(x))$. Summed over all tokens, the total accepted probability is $\sum_x \min(p, q)$, so rejection occurs with probability $1 - \sum_x \min(p, q) = \sum_x \max(0, p - q)$. The residual distribution places mass $\max(0, p(x) - q(x))/\sum_x \max(0, p - q)$ on x. Multiplying and adding:

$$\min(p, q) + \max(0, p - q) = p(x).$$

The emitted distribution is exactly the target's. Speculation changes speed, never samples.

How much speed? If each drafted token is accepted independently with rate α, a verification cycle keeps the accepted run plus one token the target supplies regardless.

DERIVATION 12.3

Expected tokens per cycle

The run of accepted tokens has length i with probability $\alpha^i(1 - \alpha)$ for $i < k$, and length k with probability α^k; each case yields $i + 1$ tokens. Summing the truncated geometric series,

$$\mathbb{E}[\text{tokens per cycle}] = \frac{1 - \alpha^{k+1}}{1 - \alpha}.$$

If the draft costs c target-forwards per token, a cycle costs $kc + 1$ target-forwards. The speedup is the ratio.

WORKED EXAMPLE 12.2

The economics, end to end

Take $\alpha = 0.8$, $k = 4$, $c = 0.1$. Tokens per cycle: $(1 - 0.8^5)/0.2 = (1 - 0.328)/0.2 = 3.36$. Cycle cost: $4(0.1) + 1 = 1.4$ target-forwards. Speedup: $3.36/1.4 = \mathbf{2.4\times}$ — with provably identical output.

Sensitivity matters more than the headline. At $\alpha = 0.5$ the same configuration gives $(1 - 0.5^5)/0.5 = 1.94$ tokens per 1.4 forwards, a mere $1.38\times$. The draft's *agreement* with the target is everything; its standalone quality matters only through α.

DRILL 12.2

(a) The draft assigns $q(A) = 0.5$, the target $p(A) = 0.3$. What is the acceptance probability for A? (b) With $\alpha = 0.7, k = 3, c = 0.15$, what is the speedup?

Show answers

(a) $\min(1, 0.3/0.5) = 0.6$. (b) Tokens $= (1 - 0.7^4)/0.3 = (1 - 0.2401)/0.3 = 2.53$; cost $= 3(0.15) + 1 = 1.45$; speedup $= 1.75\times$.

Choosing k is a marginal calculation: extending the draft by one more token adds an expected α^{k+1} tokens and costs c more. When those balance, you are at the optimum — and because the curve is flat near it, a range of k performs about equally.

12.3 What the chapter bought

Both techniques in this chapter trade an abundant resource for a scarce one, and both are exact — a combination worth pausing on, since it is rarer than it sounds. Streaming softmax lets attention be computed in tiles, cutting memory traffic by an order of magnitude while doing slightly more arithmetic and recomputation. Speculative decoding converts k memory-bound passes into one compute-bound verification, with a rejection rule that provably preserves the target distribution. Both illustrate the discipline this book keeps returning to: identify the binding

budget, then spend freely from the other. Chapter 13 continues in the same vein but changes the object — instead of computing the same model more cleverly, it makes the model itself sparser and cheaper.

Exercises

A · DRILLS

1. At $n = 8192$, half precision: give one head's score-matrix size and its ratio to 20 MB of on-chip memory.
2. Run the streaming softmax on blocks $\{0, 2\}$ then $\{1\}$, and verify against the one-shot sum.
3. With $\alpha = 0.9, k = 5$, give the expected tokens per cycle.
4. State, in one sentence each, what streaming softmax and speculative decoding change and what they leave exactly unchanged.

B · PROBLEMS

1. **Traffic accounting.** For standard attention at $n = 4096$, $d = 128$, half precision, compare the score-matrix traffic (written once and read once) against the unavoidable traffic for queries, keys, values, and output. Give the ratio and comment on how it scales with n and d.
2. **Proving exactness.** Carry out Derivation 12.2 for a two-symbol alphabet with explicit numbers: $p = (0.3, 0.7)$, $q = (0.5, 0.5)$. Compute the acceptance probabilities, the rejection probability, the residual distribution, and verify that the emitted distribution equals p exactly.
3. **Choosing the draft length.** With $\alpha = 0.8$ and $c = 0.1$, compute the speedup at $k = 2, 4, 8, 16$. Identify where the optimum lies, and show that the marginal condition $\alpha^{k+1} \approx c$ predicts it.

C · CHALLENGE

1. **Why decode and not prefill.** Explain, using Chapter 11's arithmetic-intensity argument, why speculative decoding accelerates generation but offers essentially nothing for prompt processing. Then propose what property a workload must have for speculation to help it, stated in terms of intensity and the ridge point, and give one non-language example that would qualify.

GATE 12 · PASS BEFORE CHAPTER 13

Reproduce Derivation 12.1 (streaming softmax, including the value accumulator's rescaling), Derivation 12.2 (the exactness proof), and Derivation 12.3 (expected tokens per cycle) on blank paper. Verify a streaming example numerically by hand. Attempt B-2 and B-3 closed book. You pass when you can state, without hedging, that both techniques are exact and explain why.

READINGS FOR CHAPTER 12

Milakov & Gimelshein, *Online Normalizer Calculation for Softmax* (2018)

Three pages, read fully. Derivation 12.1 in its original form.

Dao et al., *FlashAttention* (2022)

Sections 1–3 and Figure 1. The IO analysis is Worked Example 12.1 made rigorous.

Dao, *FlashAttention-2* (2023)

The work-partitioning section, skimmed. Nothing mathematical changed from the first paper; a good lesson in where real speedups come from once the algorithm is settled.

Leviathan, Kalman & Matias, *Fast Inference from Transformers via Speculative Decoding* (2023)

Section 2 contains the one-page proof of Derivation 12.2; compare it to yours.

Chen et al., *Accelerating Large Language Model Decoding with Speculative Sampling* (2023)

Its algorithm box, for the same rejection rule as Derivation 12.2 presented differently. Useful for seeing that two independent derivations produce the identical acceptance test.

Sparsity and Thrift

Three ways to make a model cheaper: use only part of it per token, store it in fewer bits, or train only a thin slice of it. Each attacks a different budget, and saying which is the whole skill.

Three techniques in this chapter all get described with the same word: cheaper. Mixture-of-experts is cheaper. Quantization is cheaper. Low-rank adaptation is cheaper. They are cheaper in three entirely different currencies, and a great deal of professional confusion comes from not saying which. Before we begin, then, a rule: no claim of efficiency counts until it names the resource it saves.

If you take one habit from this chapter, take this one. Fix the vocabulary before anything else. Chapter 17 will name four budgets for the whole book — training operations, serving operations, memory bytes and bandwidth, and human preference information. This chapter works inside the third of them and splits it three ways: the **weights** themselves; the **gradients and optimizer state** that accompany any trainable weight, itemised in Derivation 8.2; and the **activations** held for the backward pass, priced in Section 8.3. Set beside per-token computation — a slice of the first two budgets rather than of memory — that gives the four resources in play here. Each technique below improves exactly one of them and leaves the rest untouched — and mistaking which is trap class T9, which is expensive in both senses of the word.

13.1 Mixture-of-experts: decoupling size from compute

A dense layer spends every parameter on every token. A mixture-of-experts layer instead holds E parallel expert sub-networks and a small router that sends each token to the top k of them. Total parameters scale with E; per-token computation scales with k. The model's "size" splits into two numbers, and every claim about such a model must say which it means.

WORKED EXAMPLE 13.1

Total versus active

Take 32 layers; in each, attention costs 50 M parameters, and there are 8 experts of 150 M each with top-2 routing.

Total: $32\,(50 + 8 \times 150)\ \mathrm{M} = 32 \times 1250\ \mathrm{M} =$ **40 B** parameters.
Active per token: $32\,(50 + 2 \times 150)\ \mathrm{M} = 32 \times 350\ \mathrm{M} =$ **11.2 B** parameters.

So training compute ($6N_{\text{active}}D$, Chapter 7) and decode arithmetic behave like an 11-billion-parameter model, while memory — every expert must be resident — behaves like a 40-billion one. A mixture-of-experts buys capacity at small-model compute prices while paying large-model rent.

The router must spread tokens evenly, or a few experts receive everything while others starve and never train. Balance is encouraged by an auxiliary loss. Writing f_i for the fraction of tokens routed to expert i and P_i for the mean router probability assigned to it,

$$\mathcal{L}_{\text{aux}} = E \sum_{i=1}^{E} f_i P_i,$$

which equals 1 under perfect balance and exceeds 1 otherwise. It is a pressure, not a guarantee. The hard backstop is *expert capacity*: each expert accepts at most $(\text{capacity factor}) \times \text{tokens}/E$ tokens per batch, and tokens beyond that are dropped — a quality cliff that no loss curve displays directly.

WORKED EXAMPLE 13.2

Imbalance, and what it costs

Two experts receive fractions $f = (0.7, 0.3)$ with mean router probabilities $P = (0.6, 0.4)$. Then $\mathcal{L}_{\text{aux}} = 2\,(0.7 \times 0.6 + 0.3 \times 0.4) = 2\,(0.42 + 0.12) = 1.08$, above the balanced minimum of 1.

Now capacity. With 1024 tokens, 8 experts, and a capacity factor of 1.25, each expert accepts $1.25 \times 1024/8 = 160$ tokens. If 200 tokens are routed to expert 3, then **40 are dropped** — they pass through without that expert's contribution.

DRILL 13.1

A model has 48 layers; each has 60 M shared parameters and 64 experts of 40 M, with top-2 routing. Give total and active parameter counts and their ratio.

Show answers

Total $= 48(60 + 64 \times 40)\text{M} = 48 \times 2620\text{M} = 125.8$ B. Active $= 48(60 + 2 \times 40)\text{M} = 48 \times 140\text{M} = 6.7$ B. Ratio ≈ 18.7.

13.2 Quantization: fewer bits per weight

Storing weights in b bits instead of 16 shrinks the weight budget proportionally — and, by Chapter 11's ceiling, *speeds up memory-bound decoding by the same factor*. That second consequence surprises people, so it is worth stating plainly: on a decode workload, quantization is a speedup, because the ceiling is bandwidth divided by bytes read.

The cost is precision. A uniform quantizer covering $[-R, R]$ with b bits has step size $\Delta = 2R/2^b$, and the resulting error, modeled as uniform over one step, has variance

$$\text{MSE} = \frac{\Delta^2}{12}.$$

DERIVATION 13.1

The quantization error, and two sensitivities

A value uniformly distributed on $[-\Delta/2, \Delta/2]$ has variance $\frac{1}{\Delta}\int_{-\Delta/2}^{\Delta/2} x^2\,dx = \frac{\Delta^2}{12}$. Two consequences follow immediately from $\Delta = 2R/2^b$:

Each extra bit quarters the error. Doubling b by one halves Δ, and the error goes as Δ^2.

The error grows as the square of the range. If a few outlier weights force R from 1 to 4, the step quadruples and the MSE grows sixteenfold — at unchanged bit width. This is precisely why a handful of outlier channels wreck naive low-bit quantization, and why practical methods handle those channels separately.

Which weights deserve the bits is itself the question Chapter 10 answered in principle: precision costs, so spend it where it buys accuracy. Modern quantization methods are, in this light, allocation schemes over a bit budget — the Hinton–van Camp argument with an engineering budget attached.

DRILL 13.2

(a) How many bits keep the MSE at or below 10^{-4} over the range $[-1, 1]$? (b) A 70B model at 16, 8, and 4 bits: give the weight memory and the batch-1 decode ceiling on the reference machine.

Show answers

(a) $\Delta \le \sqrt{12 \times 10^{-4}} = 0.0346$; $2/2^b \le 0.0346 \Rightarrow 2^b \ge 57.7 \Rightarrow b = 6$. (b) 140 / 70 / 35 GB; ceilings $2000/140 = 14.3$, 28.6, 57 tokens/s.

13.3 Low-rank adaptation: training a thin slice

The third technique changes what is *trainable*. Freeze a weight matrix $W \in \mathbb{R}^{d\times d}$ and learn only a low-rank update $\Delta W = BA$, with $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times d}$ for a small rank r. The trainable parameter count falls from d^2 to $2rd$.

WORKED EXAMPLE 13.3

Which budget this actually attacks

For $d = 4096$ and $r = 16$: trainable parameters $2 \times 16 \times 4096 = 131{,}072$ against $d^2 = 16.78$ M — **0.78%** of the matrix.

Now the point. The sixteen bytes per parameter of Derivation 8.2 are paid only on *trainable* parameters — fourteen of them, strictly, since the frozen weight must still be read. Fine-tuning a 7-billion-parameter model in full would demand about 112 GB of optimizer state; adapting roughly one percent of it demands about **1.1 GB**. That is why such fine-tuning fits on a single accelerator — and it has essentially nothing to do with arithmetic, since the forward and backward passes still traverse the full frozen model.

TRAP · T9 · NAME YOUR BUDGET

Three near-universal misstatements. Quantization does not slow decoding down by adding work — it speeds it up, because decode is bandwidth-bound. Low-rank adaptation does not make training arithmetic cheaper — it makes optimizer memory cheaper, which then buys larger batches or smaller hardware. And a mixture-of-experts with the same *active* parameters as a dense model has the same compute but far greater memory. In each case the claim is only meaningful once you name the budget.

DRILL 13.3

At $d = 8192$, adapting all four attention projection matrices at rank $r = 8$: give the total trainable parameters and the percentage of the $4d^2$ attention weights.

Show answer

$4 \times 2 \times 8 \times 8192 = 524{,}288$. Attention weights $4d^2 = 268.4$ M. Fraction $= 0.20\%$.

13.4 What the chapter bought

Each technique here saves something real, and no two of them save the same thing. Mixture-of-experts decouples capacity from per-token compute, at the price of resident memory and a routing problem whose failure mode is silently dropped tokens. Quantization shrinks weights, and because decode is bandwidth-bound it buys speed directly — with an error that quarters per bit and grows as the square of the range. Low-rank adaptation shrinks optimizer state by two orders of magnitude, which is what made fine-tuning broadly accessible. Chapter 14 uses that accessibility: it is the chapter where a pretrained model is taught what kind of answer is wanted, and it contains the last of the book's three protocol derivations.

Exercises

A · DRILLS

1. A layer has 16 experts of 100 M each plus 40 M shared, top-2 routed, over 24 layers. Give total and active parameters.
2. Router fractions $f = (0.9, 0.1)$, mean probabilities $P = (0.8, 0.2)$. Give the auxiliary loss.
3. 4096 tokens, 16 experts, capacity factor 1.0, and expert 7 receives 320 tokens. How many are dropped?
4. At $d = 2048$, $r = 32$: give the trainable parameter count and its fraction of d^2.

B · PROBLEMS

1. **Reading a sparse model honestly.** A mixture-of-experts model is advertised at 140 B parameters; it has 64 experts with top-2 routing and 20% of its parameters in shared components. Estimate its active parameter count. Then state which of the following behave like the total and which like the active figure: training FLOPs, serving memory, decode arithmetic, KV cache size.
2. **Outliers and bits.** A weight matrix normally spans $[-1, 1]$ and is quantized to 8 bits. A handful of outliers extend the range to $[-6, 6]$. Give the MSE before and after, the factor of degradation, and the number of additional bits

needed to restore the original error. Explain why handling outliers separately is cheaper than adding bits everywhere.

3. **Budgets, side by side.** For a 13-billion-parameter model, estimate (a) weight memory at half precision, (b) optimizer state under full fine-tuning at 16 bytes per parameter, and (c) optimizer state under rank-16 adaptation of matrices totalling 30% of the model. Present the three numbers and state which technique you would apply to fit training on a 80 GB accelerator.

C · CHALLENGE

1. **The balance loss, minimized.** Show for $E = 2$ that $\mathcal{L}_{\text{aux}} = E \sum_i f_i P_i$ attains its minimum value of 1 when routing is uniform, by substituting $f_1 = P_1 = x$, $f_2 = P_2 = 1 - x$ and minimizing over x. Then explain in a paragraph why an auxiliary *pressure* plus a hard capacity limit is the standard design, rather than either mechanism alone — and what failure mode each one alone would permit.

GATE 13 · PASS BEFORE CHAPTER 14

Reproduce the active-versus-total accounting, Derivation 13.1 (the $\Delta^2/12$ error with both sensitivities), and the $2rd$ count with its budget argument, on blank paper. Attempt B-1 and B-3 closed book. You pass when, for any efficiency claim, you name its budget before evaluating it.

READINGS FOR CHAPTER 13

Fedus, Zoph & Shazeer, *Switch Transformers* (2021)

Section 2 and the auxiliary-loss box; the capacity-factor discussion is Worked Example 13.2's source.

Jiang et al., *Mixtral of Experts* (2024)

Read the parameter table against Worked Example 13.1 and check you can predict both numbers.

Dettmers et al., *LLM.int8()* (2022)

The outlier-emergence figure, where a few channels grow far beyond the rest at scale. That is Derivation 13.1's R^2 sensitivity, discovered empirically before it was priced.

Hu et al., *LoRA* (2021) and Dettmers et al., *QLoRA* (2023)

Section 4 of the first; the memory table of the second. Note which budget each paper's headline number refers to.

Frantar et al., *GPTQ* (2022) and Lin et al., *AWQ* (2023)

Abstracts suffice. Read both as bit-allocation schemes and map them onto Chapter 10's precision-costs-bits argument.

Teaching Preferences

Pretraining produces a model of what text is. Post-training bends it toward what an answer should be. The pipeline that does this collapsed, in one derivation, from a reinforcement-learning apparatus into a supervised loss.

Pretraining gives you a model of what text *is*. What you usually want is a model of what a good answer *looks like*, and the two are not the same. Getting from one to the other used to require a reward model, a reinforcement-learning loop, and a second network to estimate baselines. Then someone noticed that a term in the algebra cancels, and most of that apparatus disappeared.

We proceed in four stages. First, how human comparisons become a numerical score. Second, how a policy is trained against that score without drifting into nonsense. Third, the collapse itself, which removes most of the machinery the first two stages introduced. Fourth, a variant that deletes what remains of it, and which now underpins how reasoning models are trained.

14.1 The vocabulary

This chapter borrows a set of terms from reinforcement learning, and they are the only place in this book where the words do more work than the mathematics. Six of them, stated once.

A **policy** π is the model itself, seen as a conditional distribution: given a prompt x, it assigns a probability $\pi(y \mid x)$ to every possible response y. Nothing new is being introduced — a language model already is such a distribution, and calling it a policy only signals that we are about to change it by something other than next-token prediction. The **reference policy** π_{ref} is a frozen copy of where it started.

A **reward** $r(x, y)$ is a number scoring a whole response, delivered at the end rather than per token. It may come from a model fitted to human comparisons, as in Section 14.2, or from a checker that returns 1 or 0, as in Section 14.5. Either

way it is a single scalar for the entire generation, which is the feature that makes the arithmetic below simple and the credit assignment hard.

Training against a reward means raising the probability of responses that scored well. That is what a **policy gradient** does: sample responses from the current policy, and push up the log-probability of each in proportion to how good it was. Done naively this has a defect worth understanding, because everything else in the section is a response to it. If every response to a prompt scores between 8 and 10, the procedure pushes up all of them, and the differences between them — the only information about which was better — are swamped by their common size.

The fix is to subtract off a **baseline** $b(x)$, a prediction of what this prompt typically scores, and weight by the difference. That difference is the **advantage**,

$$A(x, y) = r(x, y) - b(x),$$

which is positive for responses better than expected and negative for worse. Subtracting a baseline that depends only on the prompt leaves the update unbiased while shrinking its variance, which is why every method here has one. Where the baseline comes from is the whole of Section 14.5: classically a learned network of roughly the policy's size, called the **critic** or value network, and later just the mean of a group of samples.

Finally, samples are expensive, so the responses used for an update are usually generated by a slightly older policy π_{old}. The **probability ratio**

$$\rho = \frac{\pi(y \mid x)}{\pi_{\text{old}}(y \mid x)}$$

measures how far the policy has moved on that response since it was drawn: $\rho = 1$ means no movement, $\rho > 1$ that the response has become more likely. Section 14.3 is entirely about what to do when ρ drifts too far from 1.

DRILL 14.1

(a) Four responses to one prompt score $(9, 10, 8, 9)$. Using the group mean as the baseline, give the four advantages. (b) The same four score $(0.9, 1.0, 0.8, 0.9)$. Give the advantages, and say what a policy gradient without a baseline would do differently in the two cases.

Show answers

(a) Mean 9, so advantages $(0, +1, -1, 0)$. (b) Mean 0.9, so advantages $(0, +0.1, -0.1, 0)$ — the same shape, ten times smaller. Without a baseline the first case would push up all four responses hard and the second gently, though the two carry identical information about which response was best. The baseline removes what the responses have in common, which is exactly the part that says nothing.

14.2 From comparisons to a reward

People are unreliable at scoring a response in isolation and reliable at comparing two. So the data are preference pairs: given a prompt, response y_w was preferred to y_l. The standard model of such comparisons — the **Bradley–Terry** model — assumes an underlying scalar reward r and sets the probability of the observed preference to a logistic function of the reward difference:

$$P(y_w \succ y_l) = \sigma\big(r(y_w) - r(y_l)\big).$$

Fitting r means minimizing $-\log \sigma(r_w - r_l)$ over the dataset. One structural fact matters enormously and is easy to miss.

DERIVATION 14.1

Only differences are identified

The loss depends on r_w and r_l only through their difference. Adding any constant κ to every reward leaves $(r_w + \kappa) - (r_l + \kappa) = r_w - r_l$ unchanged, so the loss is identical. The reward function is therefore determined only up to an additive constant — there is no meaningful "absolute" reward scale, and any claim resting on one is unfounded.

Check numerically. With $r_w = 1.2$ and $r_l = 0.7$, the loss is $-\ln \sigma(0.5) = -\ln 0.622 = \mathbf{0.474}$ nats. Shift both by $+100$: the difference is still 0.5 and the loss is still 0.474.

14.3 Optimizing against a reward, on a leash

Given a reward, one could simply maximize it. One should not. A policy that maximizes a learned reward without constraint will find the regions where the reward model is wrong and exploit them — the engineering face of the reward-seizing argument from Chapter 10. The remedy is a leash: penalize divergence from the pretrained reference policy π_{ref}. The objective becomes

$$\max_{\pi} \; \mathbb{E}_{y \sim \pi}\big[r(x, y)\big] - \beta \, \mathrm{KL}\big(\pi \,\|\, \pi_{\text{ref}}\big),$$

with β setting how tight the leash is. This objective — reward minus a scaled divergence from the reference — is the target for the rest of the chapter. *Every method below optimizes this same objective; they differ only in how.*

The classical optimizer is a policy-gradient method with a clipped surrogate, **proximal policy optimization** or PPO. Writing $\rho = \pi(y)/\pi_{\text{old}}(y)$ for the probability ratio against the policy that generated the data and A for the advantage, the surrogate is

$$\min\big(\rho A,\; \mathrm{clip}(\rho,\, 1-\epsilon,\, 1+\epsilon)\, A\big).$$

The $\min$ makes the objective *pessimistic*: it always takes the less favorable of the clipped and unclipped values, in both directions.

WORKED EXAMPLE 14.1

Where the clip kills the gradient

Take $\epsilon = 0.2$.

Case (i): $\rho = 1.3$, $A = +2$. Unclipped gives 2.6; clipped gives $1.2 \times 2 = 2.4$. The minimum is 2.4, the clipped branch — which is constant in ρ, so the **gradient is zero**. The policy has already moved far enough in the profitable direction; further movement earns nothing.

Case (ii): $\rho = 0.7$, $A = -1$. Unclipped gives -0.7; clipped gives $0.8 \times (-1) = -0.8$. The minimum is -0.8, again the clipped branch, again zero gradient. Pessimism applies symmetrically.

For ρ inside $[0.8, 1.2]$, clipped and unclipped agree and the gradient flows normally. The clip is a trust region enforced by making excursions unprofitable rather than forbidden.

DRILL 14.2

With $\epsilon = 0.2$, give the surrogate value and say whether the gradient is alive, for $A = +1$ at $\rho = 0.7$, and for $A = -1$ at $\rho = 1.3$.

Show answers

$A = +1, \rho = 0.7$: unclipped 0.7, clipped 0.8; min is 0.7 — the *unclipped* branch, so gradient alive. $A = -1, \rho = 1.3$: unclipped -1.3, clipped -1.2; min is -1.3 — unclipped, gradient alive. Only the two cases in Worked Example 14.1 are clipped.

14.4 The collapse

Now the chapter's centerpiece. The KL-regularized objective of Section 14.3 has a closed-form optimum. Knowing it lets us eliminate the reward model entirely.

Read the next derivation slowly; it is three short moves and the third one is the whole discovery. If you have seen this result stated before, you may remember it as “the partition function is intractable, so we ignore it.” That is not what hap-

pens, and the difference is the point. The term is never ignored and never approximated. It is carried through the algebra in full and then, at the last step, it eliminates itself — for a reason you can see coming if you watch what the preference model does with two responses to the *same* prompt.

DERIVATION 14.2 · THE CROWN JEWEL

From a reinforcement-learning objective to a supervised loss

Move 1 — the optimum. We maximize $\mathbb{E}_\pi[r] - \beta\,\mathrm{KL}(\pi\|\pi_{\text{ref}})$ over all distributions π. Write both terms as one sum and pull out $-\beta$:

$$\sum_y \pi(y)\,r(y) - \beta\sum_y \pi(y)\log\frac{\pi(y)}{\pi_{\text{ref}}(y)} = -\beta\sum_y \pi(y)\log\frac{\pi(y)}{\pi_{\text{ref}}(y)\,e^{\,r(y)/\beta}}.$$

The denominator inside the logarithm is almost a distribution: it is non-negative, but it does not sum to one. Divide it by whatever it does sum to,

$$Z(x) = \sum_y \pi_{\text{ref}}(y\mid x)\,e^{\,r(x,y)/\beta}, \qquad \pi^*(y\mid x) = \frac{1}{Z(x)}\,\pi_{\text{ref}}(y\mid x)\,e^{\,r(x,y)/\beta},$$

and π^* is a genuine distribution. Substituting it back, the objective becomes

$$-\beta\sum_y \pi(y)\log\frac{\pi(y)}{Z(x)\,\pi^*(y)} = \beta\log Z(x) - \beta\,\mathrm{KL}(\pi\|\pi^*).$$

Now read it off. The first term does not contain π at all. The second is a divergence, which by Section 2.6 is non-negative and vanishes only when its two arguments are equal. So the objective is maximized exactly when $\pi = \pi^*$, and no calculus was required — only the fact that a KL divergence is zero at its floor and positive everywhere else.

Z is intractable: it sums over all possible responses, and there are more of those than atoms in anything. That is why this form alone does not give an algorithm — and why what happens to Z in Move 3 is the whole of the discovery.

Move 2 — invert. Solve the same equation for the reward:

$$r(x,y) = \beta\log\frac{\pi^*(y\mid x)}{\pi_{\text{ref}}(y\mid x)} + \beta\log Z(x).$$

The reward is a log-ratio of the optimal policy to the reference, plus an intractable term that depends only on the prompt x, not on the response y.

Move 3 — substitute, and watch Z cancel. The preference model of Section 14.2 involves only the *difference* of two rewards for the same prompt. Substituting,

$$r(x, y_w) - r(x, y_l) = \beta \log \frac{\pi^*(y_w)}{\pi_{\text{ref}}(y_w)} - \beta \log \frac{\pi^*(y_l)}{\pi_{\text{ref}}(y_l)} + \underbrace{\beta \log Z(x) - \beta \log Z(x)}_{=\,0}.$$

The intractable term cancels exactly, because it is the same for both responses. Putting this difference into the Bradley–Terry loss gives a loss over the policy alone:

$$\mathcal{L} = -\log \sigma\left(\beta \log \frac{\pi(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right).$$

No reward model. No sampling loop. No value network. A supervised loss on preference pairs that optimizes the same KL-regularized objective — this is **direct preference optimization**, DPO, and the collapse above is the whole of it. The whole apparatus collapsed because an intractable constant was identical on both sides of a difference — the same cancellation-by-differencing that made Section 14.2's rewards identifiable only up to a constant.

The gradient has an instructive shape. Writing M for the margin inside the logistic, the loss is $-\log \sigma(M)$ and its derivative carries a factor $\sigma(-M)$. Pairs the model currently gets *wrong* — small or negative margin — receive weight near 1; pairs it already handles well receive vanishing weight. The method automatically concentrates on what it has not yet learned.

WORKED EXAMPLE 14.2

One step, by hand

Let $\beta = 0.2$, with log-ratios $\log \frac{\pi(y_w)}{\pi_{\text{ref}}(y_w)} = 1.0$ and $\log \frac{\pi(y_l)}{\pi_{\text{ref}}(y_l)} = -0.5$.

Margin: $M = 0.2\,(1.0 - (-0.5)) = 0.3$. Loss: $-\ln \sigma(0.3) = -\ln 0.574 =$ **0.554** nats. Gradient weight: $\sigma(-0.3) = 0.426$.

Train further until the margin reaches 1.0: the weight falls to $\sigma(-1) = 0.269$. The pair now matters less — exactly the desired behavior.

A sanity check worth remembering. At initialization the policy *is* the reference, so both log-ratios are zero, the margin is zero, and the loss is $-\ln \sigma(0) = \ln 2 =$ **0.693**. Any implementation whose loss does not begin at $\ln 2$ has a bug. This is the acceptance criterion for Lab 6.

TRAP · T8 · OBJECTIVE VERSUS ESTIMATOR

This derivation changed the *estimator*, not the objective. Both the clipped policy-gradient method and the collapsed loss optimize the same KL-regularized target; one does it by sampling and one by a closed-form substitution. Statements like "the direct method optimizes a different alignment goal" are false as stated. A second, subtler error: dropping Z too early. It does not vanish because it is small or ignorable — it vanishes because it is *identical on both sides of a difference*. That cancellation is the entire discovery, and a derivation that discards Z before taking the difference has skipped the content.

DRILL 14.3

With $\beta = 0.1$ and log-ratios 0.8 and 0.2, give the margin, the loss in nats, and the gradient weight.

Show answers

Margin $= 0.1(0.8 - 0.2) = 0.06$. Loss $= -\ln \sigma(0.06) = -\ln 0.515 = 0.664$ nats. Weight $= \sigma(-0.06) = 0.485$.

14.5 Deleting the critic

Policy-gradient training needs a baseline to reduce variance, conventionally supplied by a learned value network of roughly the policy's size. Chapter 8's arithmetic makes the cost concrete: carrying policy and critic together roughly doubles the memory bill. An alternative — **group-relative policy optimization**, GRPO — computes the baseline from data instead of a network: sample a *group* of G responses to the same prompt, and standardize their rewards within the group,

$$\hat{A}_i = \frac{r_i - \bar{r}}{s_r},$$

where $\bar{r}$ and s_r are the group's mean and standard deviation. The group supplies its own baseline; no critic is needed. This suits *verifiable* rewards especially well — mathematics and code, where a checker returns 1 or 0 — which is why it underpins modern reasoning training.

> **WORKED EXAMPLE 14.3**
>
> **Group advantages, and a degenerate case**
>
> A group of five responses earns rewards $(1, 1, 1, 0, 0)$. Mean $\bar{r} = 0.6$; variance $= \frac{3(0.4)^2 + 2(0.6)^2}{5} = \frac{0.48 + 0.72}{5} = 0.24$, so $s_r = 0.49$. Advantages: correct responses get $0.4 / 0.49 =$ **+0.82**, incorrect get $-0.6 / 0.49 =$ **−1.22**.
>
> Now the case that must be known: a group where *all* responses are correct, or all incorrect. Then every r_i equals the mean, every numerator is zero, and **every advantage is zero** — the group contributes no learning signal at all. This is the arithmetic reason such training curates prompts near the edge of the model's ability: problems it always solves and problems it never solves are both worthless.

DRILL 14.4

A group of eight responses earns $(1, 0, 0, 0, 0, 0, 0, 0)$. Give all eight advantages.

Show answer

Mean $= 0.125$; variance $= \frac{(0.875)^2 + 7(0.125)^2}{8} = \frac{0.7656 + 0.1094}{8} = 0.1094$, $s_r = 0.331$. Correct: $0.875 / 0.331 = +2.65$. Each incorrect: $-0.125 / 0.331 = -0.378$.

14.6 What the chapter bought

Preferences become a reward through a logistic model of comparisons, identified only up to an additive constant. Optimization against that reward must be leashed to a reference policy, or it degenerates into exploiting the reward model's errors. The leashed objective has a closed-form optimum, and inverting it and substituting into the preference model cancels the intractable normalizer — collapsing the whole pipeline into a supervised loss whose gradient automatically weights the pairs still being gotten wrong. And replacing a learned baseline with a group's own statistics deletes the critic, halving the memory and fitting verifiable-reward tasks exactly. Chapter 15 takes up what happens when the model is asked to spend more compute at answer time rather than training time.

Exercises

A · DRILLS

1. Rewards $r_w = 2.0$, $r_l = 1.1$. Give the preference loss in nats. Then add 50 to both and give it again.
2. With $\epsilon = 0.15$, state for which ρ the clip is active when $A > 0$.
3. $\beta = 0.5$, log-ratios 0.4 and 0.1. Give margin, loss, and gradient weight.
4. A group earns $(1, 1, 0, 0)$. Give the four advantages.

B · PROBLEMS

1. **The clip table.** With $\epsilon = 0.2$, tabulate the surrogate value for $A = +1$ and $A = -1$ at $\rho \in \{0.7, 1.0, 1.3\}$ — six cells — and mark which have zero gradient. State in one sentence what the pattern means for how far a single update may move the policy.

2. **The collapse, reproduced.** Starting from the KL-regularized objective, state the closed-form optimum, invert it for the reward, substitute into the Bradley–Terry preference probability, and show explicitly where and why the normalizer cancels. Write the resulting loss. Then explain, in one sentence, why the cancellation would fail if the two responses came from *different* prompts.

3. **The cost of a critic.** For a 7-billion-parameter policy trained in half precision, estimate total training memory with and without a critic of the same size, taking roughly 20 bytes per parameter for weights, gradients, and optimizer state combined. Give both figures and the ratio.

4. **Degenerate groups.** Show algebraically that a group whose rewards are all equal yields zero advantage for every member, regardless of the common value. Explain what this implies for curriculum design in verifiable-reward training.

C · CHALLENGE

1. **Two cancellations, one technique.** In Section 14.2 the reward's additive constant is unidentifiable because the loss sees only differences. In Section 14.4 the intractable normalizer cancels because it is the same for both responses to a prompt. Argue that these are the same mathematical phenomenon, state the general principle at work, and find one further place in this book where an inconvenient quantity is eliminated by the same manoeuvre. (Chapters 7 and 10 both contain candidates.)

GATE 14 · PASS BEFORE CHAPTER 15 — THREE-DAYS PROTOCOL

Reproduce Derivation 14.2 — all three moves, with the cancellation shown explicitly — on blank paper, from nothing, on three separate days. Do not reread between attempts. Also reproduce the clip table (B-1) perfectly and the group-advantage computation including the degenerate case. This is the third and last protocol derivation, and it is the one most likely to be asked of you in a professional setting: the collapse is short, elegant, and widely misremembered.

READINGS FOR CHAPTER 14

Ouyang et al., *Training Language Models to Follow Instructions with Human Feedback* (2022)

Section 3 and Figure 2 — the full pipeline in one picture, before the collapse simplified it.

Rafailov et al., *Direct Preference Optimization* (2023)

Section 4 and Appendix A. Read it only *after* your first blank-paper attempt at Derivation 14.2, then compare.

Schulman et al., *Proximal Policy Optimization Algorithms* (2017)

Section 3 for the clipped surrogate of Worked Example 14.1.

Schulman, *Approximating KL Divergence*

Short and load-bearing: how the leash of Section 14.3 is actually estimated in practice.

Shao et al., *DeepSeekMath* (2024)

The group-relative method of Section 14.5, in its original presentation.

DeepSeek-AI, *DeepSeek-R1* (2025)

Read the training recipe and label every arrow with the chapter of this book that prices it.

Thinking at Inference Time

Compute can be spent when the answer is needed, not only when the model is built. But sampling more answers only helps if you can tell which one is right — and that distinction decides everything.

All the compute so far has been spent before the user arrives. But you can also spend it afterwards: sample the same question ten times, or a hundred, and pick the best answer. Whether that helps turns entirely on a question people routinely skip — if a correct answer is somewhere in your hundred samples, how exactly do you intend to find it?

15.1 Coverage: the arithmetic of many attempts

Sample k independent attempts at a problem, each succeeding with probability p. The probability that *at least one* succeeds is one minus the probability all fail:

$$\text{coverage}(k) = 1 - (1 - p)^k.$$

This grows quickly — faster than most people expect, which is exactly what makes the next point so easy to miss. A model that solves a problem one time in twenty covers half of such problems within fourteen attempts. But note carefully what has been computed: the probability that a correct answer is *somewhere in the set*. Whether you can deliver it depends on whether you can identify it.

> **TRAP · T4 · COVERAGE IS NOT ACCURACY**
>
> Coverage is the most over-quoted number in this part of the field. It measures whether a correct answer exists among the samples, not whether the system returns one. You only realize coverage in production if you have a *selector* — a verifier, a test suite, a reward model — that picks the right sample. Reporting coverage as though it were accuracy overstates a system by exactly the selector's fallibility, which is often large.

With a selector of reliability v — the probability it identifies a correct answer when one is present — the delivered accuracy is roughly $v\left[1-(1-p)^k\right]$. The selector, not the sampler, is usually the binding constraint.

WORKED EXAMPLE 15.1

Small model, many tries, against big model, one try

Compare at equal compute. A small model costs 1 unit per attempt and succeeds with $p = 0.2$. A large model costs 10 units and succeeds with $p = 0.6$. Budget: 10 units.

The small model gets 10 attempts: coverage $= 1 - 0.8^{10} = 1 - 0.107 =$ **0.893**. The large model gets one: 0.6. With a *perfect* selector, sampling wins decisively.

Now introduce a realistic selector at $v = 0.7$: delivered accuracy becomes $0.7 \times 0.893 =$ **0.625** — barely better than the large model's single attempt. The entire advantage lived in the assumption of perfect selection.

This is why domains with *automatic* verification — code that must pass tests, mathematics with checkable answers, where $v \approx 1$ — led the inference-scaling era. Where verification is hard, the strategy degrades sharply.

DRILL 15.1

(a) At $p = 0.05$, how many samples give 50% coverage? (b) Budget 32 units: model A costs 1 with $p = 0.1$; model B costs 8 with $p = 0.45$ (so four attempts). With a perfect selector, which wins?

Show answers

(a) $0.95^k = 0.5 \Rightarrow k = \ln 0.5 / \ln 0.95 = 13.5$, so 14. (b) A: $1 - 0.9^{32} = 0.966$. B: $1 - 0.55^4 = 0.908$. A wins narrowly.

15.2 Voting, and its threshold

A cheaper form of selection is to let the samples vote: return the most common answer. Chapter 1 computed the case of five attempts at $p = 0.6$, giving 0.683. The governing fact is a threshold.

> **DERIVATION 15.1**
>
> **Voting helps only above one half**
>
> For $2m + 1$ independent attempts each correct with probability p (assuming errors do not coordinate on a single wrong answer), the majority is correct with probability $\sum_{j=m+1}^{2m+1} \binom{2m+1}{j} p^j (1-p)^{2m+1-j}$. By the law of large numbers this tends to 1 when $p > \frac{1}{2}$ and to 0 when $p < \frac{1}{2}$, as the number of attempts grows.
>
> The failure case is worth computing once. At $p = 0.4$ with five attempts, symmetry gives majority accuracy $1 - 0.683 = \mathbf{0.317}$ — *worse* than a single attempt's 0.4. Below the threshold, voting reliably amplifies the error rather than correcting it.

15.3 Eliciting versus creating

The most important distinction in this chapter is not arithmetic but conceptual, and it is the one worth carrying away. Sampling many times and selecting redistributes probability mass the model *already had*. It surfaces a capability; it does not add one. If the model's per-attempt probability of solving a class of problem is zero, no amount of sampling produces a solution, and no verifier can select one from a set that contains none.

Training a model to reason at length — using verifiable rewards and the group-relative method of Chapter 14 — is different in kind. It changes the weights, and therefore changes p itself. That mechanism can create a capability. Prompting a model to think step by step, at fixed weights, elicits; training it on verified reasoning traces creates.

DRILL 15.2

Classify each as eliciting or creating: (a) best-of-64 sampling with unit tests; (b) majority voting; (c) reinforcement training on verified mathematics; (d) a longer step-by-step prompt at fixed weights.

Show answers

(a) elicit — selects from what the model already produces. (b) elicit. (c) **create** — the weights change, so p itself changes. (d) elicit — no weight change; it changes which of the model's existing behaviors is expressed.

15.4 What the chapter bought

Coverage grows as $1-(1-p)^k$ and is not accuracy; the selector's reliability multiplies it, and in domains without automatic verification that multiplier dominates. Voting is a cheap selector with a hard threshold at one half, below which it actively hurts. And the elicit–create distinction separates the mechanisms that surface existing ability from the one that manufactures it. Chapter 16 leaves language entirely for the era's other generative pillar, where an image is produced by reversing a controlled corruption.

Exercises

A · DRILLS

1. At $p = 0.3$, give the coverage of 8 samples.
2. How many samples reach 90% coverage at $p = 0.15$?
3. Five attempts at $p = 0.55$: give the majority-correct probability.
4. Coverage 0.95 with a selector of reliability 0.6: give the delivered accuracy.

B · PROBLEMS

1. **Matched compute, honestly.** A small model costs 1 unit at $p = 0.15$; a large model costs 24 units at $p = 0.55$. At a budget of 24 units — so exactly one large-model attempt, or twenty-four small ones — compare delivered accuracy

under selectors of reliability $v = 1.0$, 0.8, and 0.6 (applying v to the sampling strategy only, since the single large-model answer needs no selection). At what v do they tie?

2. **The voting threshold.** For three independent attempts, derive the majority-correct probability $3p^2(1-p) + p^3$ and show it exceeds p exactly when $p > \frac{1}{2}$. Then evaluate at $p = 0.45$ and $p = 0.55$ and comment.

3. **When to stop sampling.** Each additional sample adds expected coverage $(1-p)^k \cdot p$ at a cost of one unit. Given a value V per solved problem and a cost c per sample, derive the stopping condition and evaluate it for $p = 0.1$, $V = 100$, $c = 1$.

C · CHALLENGE

1. **Why temperature zero breaks everything here.** All of this chapter's arithmetic assumes independent samples. Explain what happens to coverage when sampling temperature approaches zero, why the independence assumption fails, and what that implies about the relationship between diversity and inference-time scaling. Then connect this to Chapter 6's result that temperature never reorders candidates: what exactly does temperature control, and why is that the quantity that matters here?

GATE 15 · PASS BEFORE CHAPTER 16

Reproduce the coverage formula, the matched-compute comparison template, and Derivation 15.1's threshold on blank paper. Attempt B-1 closed book, and defend each classification in Drill 15.2 with one sentence. You pass when you never again quote a coverage figure without naming the selector.

READINGS FOR CHAPTER 15

Brown et al., *Large Language Monkeys: Scaling Inference Compute with Repeated Sampling* (2024)

The coverage curves of Section 15.1, measured across many domains; note where verification is automatic.

Snell et al., *Scaling LLM Test-Time Compute Optimally* (2024)

The compute-matched figure, which is Worked Example 15.1 done carefully and at scale. Note where their curves cross, and which side of the crossing a real selector puts you on.

Wei et al., *Chain-of-Thought Prompting* (2022)

The prompting-examples figure, for where sequential inference-time scaling began. Classify it yourself as eliciting or creating before you read their explanation.

Wang et al., *Self-Consistency Improves Chain of Thought Reasoning* (2022)

The accuracy-versus-samples figure, for voting as a selector. Check their gains against Derivation 15.1's threshold, and note what happens on the tasks where the single-sample rate is below one half.

Generation by Denoising

Destroy an image gradually with noise until nothing remains, then train a network to undo one step of the destruction. Run it backward and you have a generator — and the whole construction rests on the fact that Gaussians compose.

Take a photograph and add a little noise. Add a little more. Keep going until nothing is left but static. Now train a network to undo one step of that, and run it backwards from pure noise. What comes out is a photograph that never existed. It is a strange way to build a generator, and the reason it is tractable at all is a fact about Gaussians you met in Chapter 1.

16.1 The forward process collapses

The corruption is defined one step at a time: blend the current image with fresh Gaussian noise,

$$x_t = \sqrt{1-\beta_t}\, x_{t-1} + \sqrt{\beta_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I),$$

with a small schedule β_t. Applied a thousand times, this leaves pure noise. Simulating a thousand steps to produce a training example would be intolerable — but we do not have to, because a chain of Gaussian blends is itself a single Gaussian blend.

There is a practical problem to notice before the mathematics. Training requires showing the network images corrupted to every possible degree. If reaching corruption level eight hundred meant actually running eight hundred steps of noising for every training example, the method would be far too slow to use. So the question that has to be answered first is whether we can jump straight to any level in one operation. The answer is yes, and the reason is a fact about Gaussians you already know.

DERIVATION 16.1

A thousand steps in one jump

Take two steps with constants β_1, β_2, and write $\alpha_t = 1 - \beta_t$. Then $x_1 = \sqrt{\alpha_1}\,x_0 + \sqrt{1-\alpha_1}\,\epsilon_1$ and

$$x_2 = \sqrt{\alpha_2}\,x_1 + \sqrt{1-\alpha_2}\,\epsilon_2 = \sqrt{\alpha_2\alpha_1}\,x_0 + \sqrt{\alpha_2(1-\alpha_1)}\,\epsilon_1 + \sqrt{1-\alpha_2}\,\epsilon_2.$$

The two noise terms are independent Gaussians, so — variances add — their sum is a single Gaussian of variance $\alpha_2(1-\alpha_1) + (1-\alpha_2) = 1 - \alpha_1\alpha_2$. Hence $x_2 = \sqrt{\bar\alpha_2}\,x_0 + \sqrt{1-\bar\alpha_2}\,\epsilon$ with $\bar\alpha_2 = \alpha_1\alpha_2$. By induction,

$$q(x_t \mid x_0) = \mathcal{N}\big(\sqrt{\bar\alpha_t}\,x_0,\ (1-\bar\alpha_t)I\big), \qquad \bar\alpha_t = \prod_{s \le t}(1-\beta_s).$$

Any noise level is reachable in one draw. Note the coefficients: $\sqrt{\bar\alpha}$ multiplies the signal and $\sqrt{1-\bar\alpha}$ the noise, so their variances sum to 1 — the check that catches nearly every algebra slip here.

The quantity that gives the schedule meaning is the signal-to-noise ratio, $\mathrm{SNR}(t) = \bar\alpha_t/(1-\bar\alpha_t)$, which slides from very large to nearly zero. Training samples a random t, corrupts a real image to that level, and asks the network to predict the noise that was added. The network thus learns to denoise at every level at once — a curriculum in a single objective.

WORKED EXAMPLE 16.1

Reading a schedule

Take a constant $\beta = 0.02$, so $\bar\alpha_t = 0.98^t$. The half-signal point, where $\mathrm{SNR} = 1$, is where $0.98^t = 0.5$: $t = \ln 0.5 / \ln 0.98 = 0.693/0.0202 = \mathbf{34}$. By $t = 100$, $\bar\alpha = 0.98^{100} = e^{-2.02} = 0.133$ and the SNR is $0.133/0.867 = 0.15$ — structure nearly gone.

Observe that this is arithmetically the same computation as Chapter 9's emergence frontier $p^k = \frac{1}{2}$. Different subject, identical reflex.

DRILL 16.1

(a) With $\beta = 0.01$, at what step does SNR $= 1$? (b) If $\bar{\alpha} = 0.36$, write x_t explicitly in terms of x_0 and ϵ.

Show answers

(a) $0.99^t = 0.5 \Rightarrow t = 0.693/0.01005 = 69$. (b) $x_t = 0.6\,x_0 + 0.8\,\epsilon$, since $\sqrt{0.36} = 0.6$ and $\sqrt{0.64} = 0.8$; note $0.36 + 0.64 = 1$. ✓

16.2 Sampling, and how to shorten it

Generation reverses the ladder: start from noise and repeatedly remove the network's predicted noise. One reverse step is

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\epsilon_\theta(x_t, t)\right) + \sigma_t z, \qquad z \sim \mathcal{N}(0, I),$$

where ϵ_θ is the network's estimate of the noise in x_t and σ_t sets how much fresh noise is injected on the way back down. This update is **given**: deriving it requires the variational argument this chapter's readings set aside, and nothing below depends on where it came from. What matters is its shape — one network evaluation per step, and a step index that must walk down the ladder it walked up.

That shape is the cost. Run it as written and generating one image takes as many network evaluations as the forward process had steps, often a thousand. Setting $\sigma_t = 0$ makes the reverse process deterministic, and a deterministic trajectory can be sampled at a subset of its steps: take every twentieth t instead of every t, and the same schedule needs fifty evaluations rather than a thousand.

Now price it, because the arithmetic here is the whole practical story and it is one line. Cost is proportional to the number of network evaluations n_{eval}, so a stride k over a T-step schedule costs

$$n_{\text{eval}} = \left\lceil \frac{T}{k} \right\rceil,$$

and the saving is the factor k, exactly and with nothing hidden in it. At $T = 1000$ and $k = 20$ that is fifty evaluations against a thousand, a twentyfold reduction in the cost of every image the model ever generates. Quality degrades as k grows,

gently at first and then not, and where that knee sits is measured rather than derived — the readings give the table.

16.3 Guidance is extrapolation

To make generation follow a condition — a text prompt, say — the model is trained both with the condition and, some fraction of the time, without it. At sampling time the two predictions are combined:

$$\tilde{\epsilon} = \epsilon_u + w\,(\epsilon_c - \epsilon_u),$$

where ϵ_u is the unconditional prediction and ϵ_c the conditional. At $w = 0$ this is unconditional; at $w = 1$ it is exactly the conditional prediction. At $w > 1$ — the usual setting — it goes *past* the conditional prediction, in the direction that distinguishes conditional from unconditional.

WORKED EXAMPLE 16.2

Outside the interval

Let $\epsilon_u = 0.20$, $\epsilon_c = 0.30$, $w = 7.5$. Then $\tilde{\epsilon} = 0.20 + 7.5(0.10) = \mathbf{0.95}$ — far outside $[0.20, 0.30]$. Guidance is extrapolation, not interpolation. That is both why it sharpens adherence to the prompt and why large w over-saturates and collapses diversity: you are pushing well beyond where either prediction was trained to be valid.

TRAP · T5 AND T1

Three recurring slips. Reading w as a mixing weight in $[0, 1]$ — it is not; values of 5 to 10 are routine. Confusing $\alpha_t = 1 - \beta_t$ with the cumulative $\bar{\alpha}_t$ — the bar denotes the product over all steps so far. And placing the square roots wrongly: it is $\sqrt{\bar{\alpha}}$ on the signal and $\sqrt{1 - \bar{\alpha}}$ on the noise, verifiable at a glance because the variances must sum to one.

DRILL 16.2

With $w = 3$, $\epsilon_u = 0.5$, $\epsilon_c = 0.4$, compute $\tilde{\epsilon}$ and note its direction relative to the two inputs.

Show answer

$0.5 + 3(0.4 - 0.5) = 0.5 - 0.3 = 0.2$ — below both, because the conditional lies below the unconditional and guidance extrapolates in that direction.

16.4 Denoising in a compressed space

Running this process at full image resolution is wasteful, because most pixel-level detail is texture that a decoder can supply. The remedy is to compress the image into a much smaller latent representation, run the entire diffusion process there, and decode at the end. The saving is roughly the compression ratio in both computation and activation memory — often a factor of dozens.

Structurally this is the same division of labour we saw in Chapter 4's bottleneck block and Chapter 13's expert routing: do the expensive work in the narrow space, and let cheap machinery handle the wide one. It is worth recognizing the pattern, because it recurs whenever a field gets serious about cost.

16.5 What the chapter bought

The forward corruption collapses to a single Gaussian because variances add — making training tractable at any noise level in one draw. The signal-to-noise ratio gives the schedule its meaning, and its half-signal point is a logarithm. Sampling can be strided, turning a thousand evaluations into fifty. Guidance extrapolates beyond the conditional prediction, buying adherence at the cost of diversity. And running the whole process in a compressed latent space is the same narrow-middle economy that recurs throughout this book. One chapter remains: what all of this costs over a model's entire life, and how that changes what model you should have trained in the first place.

Exercises

A · DRILLS

1. With $\beta = 0.005$, find the step at which $\mathrm{SNR} = 1$.
2. At $\bar{\alpha} = 0.81$, write the coefficients on x_0 and ϵ, and verify they are consistent.
3. Compute $\tilde{\epsilon}$ for $w = 5$, $\epsilon_u = 0.1$, $\epsilon_c = 0.25$.
4. A 1000-step schedule is strided by 25. How many network evaluations does sampling take?

B · PROBLEMS

1. **Composing two steps.** Carry out Derivation 16.1 in full for two steps with distinct $\beta_1 = 0.01$ and $\beta_2 = 0.03$: give $\bar{\alpha}_2$, the coefficients on x_0 and on the combined noise, and verify the variances sum to one. Then state the induction step for general t.
2. **Designing a schedule.** You want $\mathrm{SNR} = 1$ to occur at step 200 under a constant β. Find β. Then compute $\bar{\alpha}$ at step 500 under that schedule.
3. **The cost of guidance.** Classifier-free guidance requires both a conditional and an unconditional prediction at every sampling step. State the multiplier this puts on sampling cost, and combine it with a stride of 20 on a 1000-step schedule to give the total number of network evaluations. Compare against unguided, unstrided sampling.

C · CHALLENGE

1. **The narrow middle, three times.** The bottleneck block of Chapter 4, the expert routing of Chapter 13, and the latent space of Section 16.4 all perform expensive computation in a compressed representation. For each, identify what is compressed, what does the compressing and decompressing, which budget is saved, and what is risked by compressing too aggressively. Then state the general design principle in one sentence.

GATE 16 · PASS BEFORE CHAPTER 17

Reproduce Derivation 16.1 — the two-step composition and the induction — on blank paper, and produce the SNR half-point calculation from nothing. Attempt B-1 and B-2 closed book. You pass when the coefficient placement is automatic and you check it every time by summing variances to one.

READINGS FOR CHAPTER 16

Ho, Jain & Abbeel, *Denoising Diffusion Probabilistic Models* (2020)

Sections 2–3. Skip the variational bound on a first pass and take Derivation 16.1's closed form as your target.

Song, Meng & Ermon, *Denoising Diffusion Implicit Models* (2020)

The strided sampling of Section 16.2; read the quality-versus-steps table.

Ho & Salimans, *Classifier-Free Diffusion Guidance* (2022)

Four pages, read fully; Worked Example 16.2 is their equation with numbers attached.

Rombach et al., *High-Resolution Image Synthesis with Latent Diffusion Models* (2022)

Section 3 for the compressed-space argument of Section 16.4.

Weng, *What Are Diffusion Models?*

Its DDPM section, read after the derivations, to place Derivation 16.1 among the variants that reparameterize the same forward process.

The Whole Lifecycle

Chapter 7 minimized the cost of building a model. A deployed model also costs something every time it answers — and once that is in the objective, the optimal model changes.

We opened, in Chapter 7, by minimizing what it costs to build a model. That was only half the bill. A deployed model charges you again every time it answers, for as long as it is in service — and once that second charge is in the objective, the model you should have built is not the one Chapter 7 told you to build.

17.1 The objective with serving in it

Training a model of N parameters on D tokens costs about $6ND$ operations (Chapter 7). Serving it costs about $2N$ operations per generated token — one multiply–add per parameter, forward only. Over a deployment lifetime of T tokens, total cost is

$$C_{\text{total}} = \underbrace{6ND_{\text{train}}}_{\text{build}} + \underbrace{2NT}_{\text{serve}}.$$

Look at the second term before going further, because everything in this chapter follows from its shape. It is proportional to N alone. So if two models reach the *same quality* by different routes — one large and briefly trained, one smaller and trained far longer — the smaller one is cheaper to serve forever, and as T grows that advantage eventually dominates whatever extra it cost to build.

WORKED EXAMPLE 17.1

The smaller model wins the long run

Take two options of equal quality (this equivalence is *given*; establishing it empirically is the hard part).

Option A: $N = 70$ B, $D = 1.4$ T. **Option B:** $N = 35$ B, $D = 4.2$ T.

At a lifetime of $T = 10^{13}$ tokens:

$C_A = 6(7\times10^{10})(1.4\times10^{12}) + 2(7\times10^{10})(10^{13}) = 5.88\times10^{23} + 1.40\times10^{24} =$ **1.99×10^{24}**.

$C_B = 6(3.5\times10^{10})(4.2\times10^{12}) + 2(3.5\times10^{10})(10^{13}) = 8.82\times10^{23} + 7.00\times10^{23} =$ **1.58×10^{24}**.

Option B costs 50% more to *train* and still wins the lifetime by about 20% — and every serving number from Chapter 11 (cache size, decode ceiling, memory footprint) improves at 35 B as well. The compute-optimal recipe of Chapter 7 was answering a different question than the one a deploying organization actually faces.

DERIVATION 17.1

The break-even lifetime

Two iso-quality options tie when their total costs are equal:

$$6(ND)_A + 2N_A T = 6(ND)_B + 2N_B T \implies T^* = \frac{6\left[(ND)_A - (ND)_B\right]}{2\left(N_B - N_A\right)}.$$

For the example above: $(ND)_A = 9.8 \times 10^{22}$, $(ND)_B = 1.47 \times 10^{23}$, so the numerator is $6(-4.9 \times 10^{22})$; with $N_B - N_A = -3.5 \times 10^{10}$, we get $T^* = \frac{2.94\times10^{23}}{7.0\times10^{10}} =$ **4.2×10^{12}** tokens. Below that lifetime the larger model is cheaper overall; above it, the smaller one. *The right model to train is a function of expected demand.*

17.2 But the optimum is interior

If smaller-and-longer-trained is better, why not push it arbitrarily far? Because iso-quality has diminishing returns: matching a given quality with an ever-smaller model demands disproportionately more data, and the training term eventually overwhelms the serving saving.

> **WORKED EXAMPLE 17.2**
>
> **Pushing too far**
>
> Add **Option C:** $N = 20$ B, $D = 12$ T, also of equal quality. At $T = 10^{13}$:
>
> $C_C = 6(2\times10^{10})(1.2\times10^{13}) + 2(2\times10^{10})(10^{13}) = 1.44\times10^{24} + 4.00\times10^{23} =$ **1.84×10^{24}**.
>
> Worse than Option B's 1.58×10^{24}, though still better than A. The optimum lies *between* A and C — an interior point, not a corner. This is why the answer is a calculation rather than a slogan: "train smaller models longer" is directionally right and quantitatively unbounded, and only the arithmetic locates the stopping point.

DRILL 17.1

At a lifetime of only $T = 10^{11}$ tokens, recompute the totals for Options A and B above and say which wins.

Show answer

$C_A = 5.88\times10^{23} + 2(7\times10^{10})(10^{11}) = 5.88\times10^{23} + 1.4\times10^{22} = 6.02\times10^{23}$. $C_B = 8.82\times10^{23} + 7\times10^{21} = 8.89\times10^{23}$. **A wins** — below the break-even lifetime, the cheaper-to-train model is correct.

17.3 Distillation

A second lever changes the iso-quality menu itself. Train a large, expensive teacher once; then train a small student to imitate not just the teacher's answers

but its full output distribution. The soft distribution carries more information per example than a hard label — a statement Chapter 10 lets us make precisely, since a distribution over outcomes specifies more bits than a single outcome does. The student reaches a quality its own size and data would not otherwise buy, and it is the student that gets served.

The modern refinement grades the *student's own* samples rather than only imitating the teacher's, which connects this chapter directly to Chapters 14 and 15: the teacher (or a verifier) supplies the reward signal, and the student improves on the distribution it actually produces.

17.4 The four budgets

The book has priced four distinct scarce resources, and naming them is the last piece of vocabulary to carry away.

- **Training operations** — Chapter 7's $6ND$, minimized subject to a budget.
- **Serving operations** — this chapter's $2NT$, proportional to size and demand.
- **Memory bytes and bandwidth** — Chapters 11 through 13: caches, weight reads, optimizer state, and the roofline that says which of them binds.
- **Human preference information** — Chapter 14's comparisons, the scarcest and least substitutable of the four, since no amount of compute manufactures a judgment about what a good answer is.

Nearly every technique in Part III is an arbitrage between two of these: spend an abundant budget to relieve a binding one. Streaming softmax spends arithmetic to save bandwidth. Speculation spends draft compute to save weight reads. Quantization spends precision to save bytes. Low-rank adaptation spends expressiveness to save optimizer state. Mixture-of-experts spends memory to save per-token compute. Distillation spends a large training run once to save serving forever. Once you see the four budgets, the field's papers stop being a list of tricks and become a small set of trades.

17.5 What the book bought

You began with six mathematical reflexes and have ended able to derive, from nothing: why deep networks were hard to train and what fixed it; why residual connections work, three separate ways; how memory decays in a recurrence and what pins it; why attention scales its scores and what it costs; how to allocate a compute budget optimally, and what a halving of loss really costs; what pipelining and sharding cost; how a benchmark can lie by threshold or by leakage; why compression is the master currency and Occam a theorem; why serving is a memory problem; how two exact techniques buy speed for free; what sparsity and quantization and adaptation each save; how a reinforcement-learning pipeline collapsed into a supervised loss; what sampling more answers can and cannot achieve; how noise-reversal generates images; and how the whole lifecycle changes what you should have built.

None of it required memorizing a benchmark score or a date. The thirty-eight derivations of Appendix A are the entire syllabus, and the trap taxonomy of Appendix B names the ways they are misapplied. The real examination is not the mock paper. It is next quarter's arXiv posting: read a page, name the budget it arbitrages, and identify the derivation that prices it. If you can do that, the program worked — and unlike any particular result in this book, that skill does not expire.

Exercises

A · DRILLS

1. A 40 B model serves 5×10^{12} tokens. Give the serving cost in operations.
2. The same model was trained on 2 T tokens. Give the training cost, and the ratio of serving to training.
3. State the break-even formula for two iso-quality options.
4. Name the four budgets and give one technique that relieves each.

B · PROBLEMS

1. **A deployment decision.** Two iso-quality options: P has $N = 100$ B, $D = 2$ T; Q has $N = 40$ B, $D = 6$ T. Compute total lifetime cost at $T = 10^{12}$ and $T = 10^{14}$,

find the break-even lifetime, and recommend one, stating your assumption about demand.

2. **Where the interior optimum sits.** Given iso-quality options at (N, D) of (80 B, 1 T), (40 B, 3 T), (20 B, 10 T), and (10 B, 40 T), compute lifetime cost at $T = 5 \times 10^{12}$ for each and identify the best. Explain why both extremes lose.

3. **Serving beyond FLOPs.** Worked Example 17.1 counts only operations. Using Chapter 11, list three further ways the 35 B option beats the 70 B option in production, quantifying at least one of them. Then state why an operations-only account understates the case for the smaller model.

C · CHALLENGE

1. **The unified objective.** Write an objective that, minimized, would jointly determine the model size, training tokens, and quantity of preference data for a deployment with known lifetime T and known per-unit costs of compute and human annotation. State what empirical inputs it needs that this book has treated as given, and identify which of them is hardest to obtain and why. (You are being asked to state honestly where derivation ends and measurement must begin — the most valuable judgment in the field.)

GATE 17 · THE FINAL GATE

Reproduce Derivation 17.1 and both worked examples on blank paper. Then sweep the entire Derivation Bank — all thirty-eight results, Chapters 3 through 17 — reproducing each from nothing over a few sittings. Finally, take the mixed examination described in Appendix D's preface under timed conditions. You pass the book, not merely the chapter, when the sweep is complete and the trap taxonomy in Appendix B contains no class you have not personally fallen into and logged.

READINGS FOR CHAPTER 17

Sardana & Frankle, *Beyond Chinchilla-Optimal: Accounting for Inference in Language Model Scaling Laws* (2024)

This chapter's argument in full. Its iso-quality figures supply exactly what Worked Example 17.1 takes as given — the curve along which a smaller model trained longer matches a larger one.

Hinton, Vinyals & Dean, *Distilling the Knowledge in a Neural Network* (2015)

Sections 1–2. Read the soft-target argument alongside Chapter 10's account of information in a distribution.

Hoffmann et al., *Chinchilla* (2022)

Reread Section 5 now. The post-publication drift toward far higher token-per-parameter ratios reads as an obvious consequence once serving is in the objective.

Touvron et al., *LLaMA* (2023)

An explicit statement of the deployment-cost reasoning of Section 17.1, made before the formal analysis existed.

Appendices

APPENDIX A

The Derivation Bank

The syllabus of this book, in 50 items. You are ready when each comes off a blank page, unprompted, with given constants slotted in.

This is not a summary to read. It is a checklist to be tested against — the statement of each result, without its derivation, so that you must supply the derivation yourself.

Use it three ways. As a **gate**: at the end of each chapter, cover the book and reproduce that chapter's entries. As a **sweep**: at the end of each Part, reproduce every entry so far in a few sittings. As a **diagnostic**: when a new paper resists you, scan this list and ask which entry prices it. Three items — marked **protocol** — are to be reproduced from nothing on three separate days, without rereading between attempts.

Part I · Foundations

1 Training and serving cost: 2 FLOPs per multiply–accumulate, one forward pass and two backward, giving $6ND$ to train and $2N$ per generated token to serve. Ch. 2

Part II · The Classical Era

2 The stable step-size window $0 < \eta < 2/\lambda_{\max}$, derived from $|1 - \eta\lambda| < 1$, and why the steepest curvature binds. Ch. 3

3 Steps to converge, $t = \ln F / \ln(1/\rho)$, with the given rates $\rho_{\text{GD}} = \frac{\kappa-1}{\kappa+1}$ and $\rho_{\text{mom}} = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$; the speedup ratio tends to $\sqrt{\kappa}$. **protocol** Ch. 3

4 Fan-in variance $m\sigma^2$ and the choice $\sigma^2 = 1/m$; the symmetry-breaking argument for random initialization. Ch. 3

5 Convolution arithmetic — output size with the floor, parameters with the bias, multiply–accumulates — chained through a network. Ch. 4

6 Expansion of $\prod_\ell (I + J_\ell)$: $\binom{L}{k}$ terms with k Jacobians, and the indestructible $k = 0$ identity. Ch. 4

7 Residual path lengths as $\mathrm{Binomial}(L, \frac{1}{2})$, with the normal approximation and continuity correction. Ch. 4

8 The $\sqrt{L}$ activation drift from independent per-block perturbations, and where normalization must sit. Ch. 4

9 Bottleneck-versus-plain multiply–accumulate ratio at general channel compression c. Ch. 4

10 Dilated receptive field: a dilation-r layer adds $2r$, so the schedule $1, 2, \ldots, 2^{n-1}$ reaches $2^{n+1} - 1$ against $2n + 1$ undilated; and the parameter-ratio payoff. Ch. 4

11 Gradient horizon $t > \ln \epsilon / \ln f$; the gate bias $b = \ln \frac{f}{1-f}$; dropout compounding per time step versus per layer. Ch. 5

12 Recurrent layer parameter count $4d_h(d_{\text{in}} + d_h + 1)$, assembled into a full model, and the embedding-plus-readout fraction. Ch. 5

13 Power-law slope extraction from a “per tenfold” statement, and extrapolation to a target improvement. Ch. 5

14 Score variance $\mathrm{Var}(q \cdot k) = d$ forcing the $1/\sqrt{d}$ scale; attention $2n^2 d$ and feed-forward $8nd^2$ costs and their crossover at $n = 4d$; temperature as monotone logit scaling. Ch. 6

15

Reversal-trick lags: unreversed lag n everywhere, reversed lag $2j-1$, mean exactly n, maximum worse. Ch. 6

16 The transformer block's parameter count: $3d^2$ of query, key and value projections with the head count cancelling, d^2 of output projection and $8d^2$ of feed-forward — $12d^2$ per layer. Ch. 6

17 The compute-optimal allocation: substitute, differentiate, obtain $aAN^{-a} = bBD^{-b}$, and read off $N \propto C^{b/(a+b)}$, $D \propto C^{a/(a+b)}$. **protocol** Ch. 7

18 The halving cost $2^{1/\alpha}$ for a loss term $\propto N^{-\alpha}$, and re-budgeting a model at a target token ratio under fixed compute. Ch. 7

19 Pipeline completion time $M+P-1$ from a timing diagram, and the bubble fraction $\frac{P-1}{M+P-1}$ with both boundary cases. Ch. 8

20 Optimizer-state sharding arithmetic, always with the unsharded total computed first. Ch. 8

21 The sixteen bytes per parameter of mixed-precision training, itemised, and which of them freezing a parameter removes. Ch. 8

22 Activation memory per token per layer, and the checkpointing optimum $g = \sqrt{L}$ from minimising $g + L/g$, bought with about a third more arithmetic. Ch. 8

23 Communication per axis of parallelism: $2N$ bytes per device per step for data parallelism, activations per layer for tensor parallelism, stage boundaries for pipelining. Ch. 8

24 Exact-match accuracy p^k, its 50% frontier at $p = 2^{-1/k}$, and the manufactured-cliff argument. Ch. 9

25 Contamination: observed $= c + p(1-c)$, inverted for c, and the Bayes flip $P(\text{contaminated} \mid \text{correct}) = c/\text{observed}$. Ch. 9

26 Two-part codes: model bits plus residual bits, model selection by the minimum, and hypothesis odds 2^{Δ}. Ch. 10

27 Cross-entropy as a literal file size; tokens times bits per token divided by eight. Ch. 10

28 The Gaussian coding cost of a weight, its zero case, its divergence as precision rises, and the noisy-weights-are-cheap reading. Ch. 10

29 The length prior $2^{-\ell}$: Kraft, posterior mixtures over consistent programs, and the theorem that each extra bit halves the vote. Ch. 10

30 Typicality versus membership, via randomness deficiency $\log_2 |M| - K(x)$. Ch. 10

31 The OLS excess risk $\sigma^2 p/(n-p-1)$ and its divergence at the interpolation threshold; grokking as a code-length transition. Ch. 10

32 The universal intelligence measure $\Upsilon = \sum_i 2^{-K_i} V_i$ with its three properties. Ch. 10

33 The reward-seizing threshold $r > (1-\gamma)/\gamma$, from the geometric series. Ch. 10

34 Bits per token divided by the tokenizer's compression ratio gives bits per character, the only figure that compares two models with different vocabularies. Ch. 10

Part III · The Modern Era

35 KV cache size $2\,L\,H_{kv}\,d_{\text{head}}\,s\,B\,\beta$, derived from what is stored and per what. Ch. 11

36

Arithmetic intensity and the ridge point; the intensity of a large matmul, of batch-1 decode, and of batch-B decode. Ch. 11

37 The decode ceiling — bandwidth divided by weight bytes — and the model FLOP utilization formula. Ch. 11

38 Streaming softmax: the running maximum and normalizer update, with the value accumulator's rescaling, and a numerical verification that it is exact. Ch. 12

39 Speculative decoding's exactness: acceptance at $\min(1, p/q)$, residual resampling, and the proof that the emitted distribution is p. Ch. 12

40 Expected tokens per verification cycle $\frac{1-\alpha^{k+1}}{1-\alpha}$, the speedup expression, and the marginal condition for k. Ch. 12

41 Mixture-of-experts accounting: total versus active parameters, the auxiliary balance loss and its uniform minimum, and capacity-based dropping. Ch. 13

42 Quantization error $\Delta^2/12$, with the quarter-per-bit law and the R^2 range sensitivity. Ch. 13

43 Low-rank adaptation's $2rd$ trainable parameters, and the three-budget map of which technique relieves which resource. Ch. 13

44 Preference modelling: the logistic comparison loss, identifiability up to a constant, and the KL-tethered objective. Plus the clipped surrogate's four regimes and where the gradient dies. Ch. 14

45 The preference-optimization collapse: closed-form optimum, inversion for the reward, substitution into the comparison model, and the exact cancellation of the normalizer. Plus group-relative advantages and the degenerate-group case. **protocol** Ch. 14

46 Coverage $1-(1-p)^k$ with a selector's multiplier, and the voting threshold at $p = \frac{1}{2}$ below which majority vote amplifies error. Ch. 15

47 Eliciting versus creating: which interventions redistribute probability the model already has, and which change p itself. Ch. 15

48 The diffusion forward process $q(x_t \mid x_0)$ by Gaussian composition, the signal-to-noise schedule, and the variance check that the coefficients square to one. Ch. 16

49 Guidance as extrapolation beyond the conditional prediction, and the stride cost $\lceil T/k \rceil$. Ch. 16

50 The lifecycle cost $6ND + 2NT$, its break-even lifetime, and the interior optimum that makes "train smaller and longer" a calculation rather than a slogan. Ch. 17

Synthesis

Four essays, to be written once, at the end. Each is a page, and each is a test of whether the bank has become a way of seeing rather than a list.

S1 Restate a modern quantization method's goal in the language of entry 23 — precision purchased where precision pays.

S2 Explain the mixture-of-experts auxiliary loss as a pressure toward routing entropy, and say what the capacity limit adds that the loss alone cannot provide.

S3 Diagram a modern reasoning-model training pipeline as a dataflow, labelling every arrow with the entry above that prices it.

S4 The four budgets — training operations, serving operations, memory bytes and bandwidth, human preference information. For each, name which chapters price it and which techniques arbitrage it against the others.

APPENDIX B

The Trap Taxonomy

Ten ways to be wrong, each with a diagnostic. Label every entry in your error log with one of these; the pattern that emerges is your actual curriculum.

Mistakes in this material are not random. They fall into ten families, and knowing which family you are prone to is worth more than knowing another result.

Use the log honestly. One line per error: what you did, what was right, and the class. After twenty entries the distribution will be lopsided, and the two heaviest classes are where your remaining study time should go.

T1 · OFF-BY-ONE AND FORGOTTEN TERMS

Symptom: the method is right and the number is slightly wrong. Dropped bias terms in a parameter count; a floor not taken in a convolution size; a missing continuity correction; a strict inequality read as non-strict; "the smallest t exceeding" confused with "the t solving."

Diagnostic: recompute the smallest instance by hand. A one-layer network, a three-block stack, a two-token vocabulary — if the small case disagrees with your formula, the formula has a missing term.

T2 · THE WRONG BALANCE

Symptom: an optimum that looks plausible and is wrong. The canonical instance is writing the compute-optimal condition as $AN^{-a} = BD^{-b}$ rather than the exponent-weighted $aAN^{-a} = bBD^{-b}$. A close cousin is adding standard deviations where variances should be added.

Diagnostic: never assert a balance condition; derive it by substitution. If you did not differentiate, you guessed.

T3 · MEAN AND MAXIMUM CONFUSED

Symptom: concluding that a technique which helps must have improved the aggregate you first reached for. The reversal trick leaves the mean lag exactly unchanged and makes the maximum worse, yet helps. Two agents with equal average performance across environments need not have equal weighted scores.

Diagnostic: when a technique helps, ask which statistic actually moved — and be prepared for the answer to be "the shape of the distribution, not its mean."

T4 · METRIC MISTAKEN FOR CAPABILITY

Symptom: attributing to the model what belongs to the ruler. An exact-match cliff read as a discontinuous capability; a contaminated benchmark read as skill; coverage quoted as accuracy; a perplexity drop that sounds enormous and is half a bit.

Diagnostic: for any reported number, ask what would change it besides the model. If the answer is "the threshold," "the test set's provenance," or "the selector," you are looking at a measurement, not a capability.

T5 · DIRECTION REVERSED

Symptom: the magnitude is right and the sign of the reasoning is inverted. Believing noisy weights are the expensive ones (they are the cheap ones); that quantization slows decoding (it speeds it up); that factorizing a large kernel costs expressiveness (it adds a nonlinearity); that voting always helps (below $p = \frac{1}{2}$ it hurts); that dilation reduces resolution (it preserves it).

Diagnostic: state the claim, then construct the extreme case. Infinite precision, zero bits, $p = 0$, one micro-batch. Extremes make reversed signs obvious.

T6 · LOGARITHM AND EXPONENT SLIPS

Symptom: a factor of $\ln 2$ adrift, or a decade confused with a factor. Mixing natural and base-two logarithms; losing the sign of $\ln(0.x)$; reporting "1.36" when the answer is $10^{1.36}$.

Diagnostic: carry units. "Decades" and "factors" are different quantities; bits and nats are different quantities. Writing the unit after every number catches this class almost entirely.

T7 · RATE CONFUSED WITH STOCK

Symptom: a per-unit quantity reported as a total or vice versa. A KV cache quoted per token when the total was wanted; tokens per cycle reported as tokens per second; a bubble fraction confused with an absolute idle time.

Diagnostic: say the quantity aloud with its denominator. "Bytes per token," "tokens per second," "operations per byte." A quantity you cannot name with a denominator is one you have not pinned down.

T8 · OBJECTIVE CONFUSED WITH ESTIMATOR

Symptom: believing a computational improvement changed what is being computed. Streaming softmax is exact; speculative decoding is exact; the preference-optimization collapse changed the estimator, not the objective. Quantization is the one technique in this group that genuinely does change the function — which is precisely why the distinction is worth policing.

Diagnostic: for each technique, ask whether two implementations would produce identical outputs given identical randomness. If yes, it is an estimator change and nothing about the model's behavior may be attributed to it.

T9 · THE UNNAMED BUDGET

Symptom: "this makes it faster" or "this makes it smaller," with no resource specified. Low-rank adaptation saves optimizer state, not training arithmetic. Grouped-query attention saves cache, not weight reads. A mixture-of-experts saves per-token compute, not memory.

Diagnostic: refuse to evaluate any efficiency claim until the budget is named. There are four: training operations, serving operations, memory bytes and bandwidth — which splits into weights, optimizer state and activations — and human preference data. Most confused arguments in this field dissolve at this question.

T10 · DEGENERATE REGIMES IGNORED

Symptom: a formula applied where its assumptions have collapsed. A group of samples all correct, giving zero advantage and no learning signal. Voting below the threshold. A pipeline with one micro-batch. "Diverse" samples drawn at temperature zero. Guidance with a negative weight.

Diagnostic: for every formula you derive, evaluate it at both extremes of its parameter before using it. This is also where examiners and interviewers live, because the degenerate case reveals whether you derived the result or memorized it.

Using the log

Two habits make the taxonomy work. First, log the error *when it happens*, not at the end of a session — the reasoning that produced it is only recoverable while it is fresh. Second, when a class accumulates five entries, do not simply resolve to be careful. Go back to the chapter that introduced the underlying derivation and reproduce it from scratch; a repeated trap almost always signals that a derivation was memorized rather than built.

A note on T4 and T9, which between them account for most professional disagreements in this field: they are not really arithmetic errors at all. They are

failures to ask what a number refers to. That question — what exactly is being measured, and out of which budget — is the habit this book most wants to leave you with.

APPENDIX C

The Lab Manual

Eight builds. Each is accepted only when it produces a number your derivation predicted in advance — because a derivation confirmed by your own running code is learned twice.

Reading a derivation convinces you. Writing code that reproduces its number to three decimal places convinces you differently, and more permanently.

Each lab is anchored to the chapter whose derivation supplies its acceptance number, so they cluster where Part III does its densest work rather than spreading one to a chapter. Each takes four to six hours. The rule that makes them worth the time: **predict the number before you run anything.** Write the expected value in your notebook, then run the code. When they disagree, one of the two is wrong, and finding out which is where the learning happens — more often than you would expect, it is the code.

Scaffolding: a minimal transformer implementation such as nanoGPT and the Zero to Hero lecture series cover Labs 1 through 4 comfortably. minbpe is a reference for Lab 1. From Lab 6, a preference-training library such as TRL is permitted, but write the loss function yourself first.

LAB 1 · ALONGSIDE CHAPTER 11

A byte-pair tokenizer, from scratch

Implement byte-pair encoding: count adjacent pairs, merge the most frequent, repeat. Train it on a megabyte of text and use it to encode and decode round-trip.

ACCEPTANCE

Round-trip encoding and decoding returns the input exactly, for a corpus containing multi-byte characters. Report the compression ratio — bytes in, tokens out — and reproduce a known merge sequence on a short test string. Predict the vocabulary size after k merges before running.

LAB 2 · ALONGSIDE CHAPTER 12

Attention, and a streaming softmax

Implement scaled dot-product attention and a full transformer block from primitives. Then, separately, implement the streaming softmax of Derivation 12.1 — running maximum, running normalizer, block-wise rescaling.

ACCEPTANCE

Your streaming softmax matches a library softmax to within 10^{-6} on random inputs of at least a thousand elements, processed in at least five blocks. Before running, compute by hand the two-block example of Derivation 12.1 and confirm your implementation reproduces **`1.553`**.

LAB 3 · ALONGSIDE CHAPTER 13

Train a character-level language model

Train a small transformer on a few megabytes of text until the validation loss is stable. Report loss in nats per character, then convert to bits per character.

ACCEPTANCE

Validation loss at or below 1.6 nats per character. Then the real criterion: using Chapter 10's identity, *predict* the compressed size of the corpus from your bits-per-character figure, write the prediction down, and compare against what a general-purpose compressor achieves on the same file. Explain any gap.

LAB 4 · ALONGSIDE CHAPTER 11

A key–value cache

Add incremental decoding with a KV cache to your Lab 3 model, so that generating token $t+1$ reuses the keys and values computed for tokens $1 \dots t$.

ACCEPTANCE

Generated output is bit-identical with and without the cache, given the same random seed — this is the correctness test, and it catches nearly every indexing bug. Then measure tokens per second with and without, and explain the speedup using Chapter 11's roofline argument. Predict the cache's size in bytes from Derivation 11.1 and verify against measured memory.

LAB 5 · ALONGSIDE CHAPTER 13

Low-rank adaptation, from scratch

Implement a low-rank adapter on one weight matrix of your model: freeze the original, add a trainable BA product of rank r, and train only the adapter.

ACCEPTANCE

Your printed count of trainable parameters equals $2rd$ *exactly* — not approximately. Measure optimizer-state memory before and after, and confirm the ratio matches the fraction of parameters made trainable. State which budget you saved and which you did not.

LAB 6 · ALONGSIDE CHAPTER 14

Preference optimization

Take a small open pretrained model as both policy and frozen reference. Implement the collapsed preference loss of Derivation 14.2 yourself — the log-ratio difference, scaled by β, through a logistic. Train on a public preference dataset.

ACCEPTANCE

The loss at step zero equals $\ln 2 = 0.693$. *Derive why before you run it* — at initialization the policy is the reference, both log-ratios vanish, the margin is zero, and $-\ln \sigma(0) = \ln 2$. This single check catches almost every implementation error. Then: the mean margin increases monotonically over training, and implied-reward accuracy on held-out pairs exceeds 60%.

LAB 7 · ALONGSIDE CHAPTER 13

Post-training quantization

Quantize your Lab 3 model's weights to 8 bits with a uniform quantizer, and measure both the weight error and the effect on model quality.

ACCEPTANCE

Measured mean squared quantization error is within a factor of two of the $\Delta^2/12$ prediction — compute the prediction first. Report the change in validation perplexity, and separately report the error for a layer with outlier weights, confirming Derivation 13.1's R^2 sensitivity.

LAB 8 · ALONGSIDE CHAPTER 12

Speculative decoding

Train or obtain a smaller draft model compatible with your Lab 3 model's tokenizer. Implement draft-and-verify decoding with the rejection rule of Derivation 12.2.

ACCEPTANCE

Two criteria. *Correctness:* over many samples at fixed temperature, the output token distribution is statistically indistinguishable from the target model's alone — the technique is exact, and your implementation must demonstrate it. *Performance:* measure the empirical acceptance rate α, then confirm that measured tokens per cycle falls within 10% of $\frac{1-\alpha^{k+1}}{1-\alpha}$.

On what the labs are for

They are not for learning a framework. They are for closing the loop between a symbol and a measurement — so that $\Delta^2/12$ stops being a formula you can re-

state and becomes a number you have watched appear on your own screen. Three of the eight acceptance criteria (Labs 2, 6, and 8) are exactness checks: they verify that a technique you derived as mathematically equivalent really is equivalent in floating point. Those three are the most valuable, because exactness is the property most often assumed and least often verified.

If time is short, do Labs 2, 4, and 6. They cover the streaming derivation, the memory-versus-compute distinction that organizes all of Part III, and the collapse that organizes modern post-training.

APPENDIX D

Solutions

Worked answers to the B and C exercises. Consult them after attempting, never before — a solution read is worth a fraction of a solution found.

Each solution gives the reasoning, not only the number, because the number is the least transferable part.

> **THE FINAL EXAMINATION**
>
> When the Derivation Bank sweep is complete, sit a mixed paper under timed conditions: three and a half hours, drawing on all seventeen chapters. A good blueprint is sixteen questions on Part II, sixteen on systems (Chapters 11–13), fourteen on post-training and inference-time compute (Chapters 14–15, with the collapse of Derivation 14.2 compulsory), and ten on generation and lifecycle (Chapters 16–17), including cross-era items that force a Part II tool onto a Part III object — for instance: *derive the cache saving from sharing keys and values across head groups, then find the batch size at which decode becomes compute-bound, before and after.*

Part I

Chapter 1 · The Mathematical Toolkit

B-1 · Compounding two ways

$0.9^{100} = e^{100 \ln 0.9} = e^{-10.54} = 2.7 \times 10^{-5}$. $0.99^{100} = e^{-1.005} = 0.366$. The first design annihilates the signal; the second preserves over a third of it. The lesson is the sensitivity of a compounded factor: a change of 0.09 in the per-step multiplier changes the hundred-step survival by four orders of magnitude.

B-2 · Square-root drift

Independent variances add, so the accumulated variance is $0.01L$ and the standard deviation is $0.1\sqrt{L}$. At $L = 100$, it is 1.0 — comparable to a unit-scale signal.

B-3 · Exponent-weighted balance

Substitute $D = C/N$ into $f = AN^{-a} + BD^{-b}$ to get $f(N) = AN^{-a} + BC^{-b}N^{b}$. Then $f'(N) = -aAN^{-a-1} + bBC^{-b}N^{b-1} = 0$. Multiply by N and recognize $BC^{-b}N^{b} = BD^{-b}$: the condition is $aAN^{-a} = bBD^{-b}$. The bare terms are equal only if $a = b$, because the constraint exchanges N for D at a rate set by how fast each term responds — and that rate is the exponent.

B-4 · A leaked benchmark

$0.7 = c + 0.6(1 - c) = 0.6 + 0.4c \Rightarrow c = 0.25$. By Bayes, $P(\text{contaminated} \mid \text{correct}) = c/0.7 = 0.357$. Over a third of the successes are recall.

C-1 · The halving cost of scale

Halving requires $k^{-\alpha} = \frac{1}{2}$, so $k = 2^{1/\alpha}$. At $\alpha = 0.076$: $1/\alpha = 13.16$ and $k = 2^{13.16} \approx 9{,}100$. Separately, $1000^{0.74} = 10^{2.22} \approx 166$. Together: a nine-thousand-fold parameter increase for one halving, with data required to grow 166-fold per thousandfold of parameters. Progress by scale alone is dependable and brutally priced — which is why the allocation question of Chapter 7 matters so much.

Chapter 2 · The Objects

B-1 · Where the parameters are

Tables: $2 \times 32{,}000 \times 1024 = 65.54$ M. Each hidden layer holds $1024 \times 1024 + 1024 = 1{,}049{,}600$, so twelve hold 12.60 M. Total 78.13 M, of which the tables are $65.54/78.13 = 84\%$.

The tables' size is fixed by V and d; the layers' size grows with depth. So the fraction falls as layers are added — at 120 layers the same tables would be about a third, and at 1200 about four percent. Scaling studies relate capability to parameter count and want a number that tracks computational depth; a count dominated by a vocabulary table would make two models with different tokenizers look like different sizes while computing identically. Hence non-embedding parameters.

B-2 · Saturation, numerically

The softmax of $(a, 0)$ gives the first option $\sigma(a)$, and $\sigma'(a) = \sigma(a)(1 - \sigma(a))$.

a = 1 → p = 0.731, slope 0.197 · a = 4 → p = 0.982, slope 0.0177 · a = 10 → p = 0.99995, slope 0.0000454

A gap of 10 leaves the slope four thousand times smaller than a gap of 1. As the leading logit pulls away, the probability it produces stops responding to it, so the gradient reaching the logits vanishes and learning stalls — the network is confident and therefore deaf. Nothing has gone wrong with the parameters; the arithmetic of the exponential simply ran out of room. This is why every mechanism in the book that feeds a softmax also controls the scale of what it feeds it, from the $1/\sqrt{d}$ of Chapter 6 to the temperature of Section 6.4.

B-3 · The product of Jacobians

With every $J_\ell = cI$, the product over L layers is $c^L I$. At $L = 50$: $0.9^{50} = e^{50 \ln 0.9} = e^{-5.27} = 5.2 \times 10^{-3}$, and $1.1^{50} = e^{50 \ln 1.1} = e^{4.77} = 117$.

A ten percent error in either direction, compounded fifty times, spans a factor of about $23{,}000$ between the two outcomes — one gradient has all but vanished, the other has grown by two orders of magnitude. The usable range of c is therefore not a range at all but a knife edge near 1, and it narrows as L grows: holding c^L within a factor of ten of 1 requires c within about $2.3/L$ of unity, which at $L = 50$ is five percent and at $L = 500$ is half of one percent. Depth does not merely make this problem worse; it makes it worse in proportion to itself. Chapter 4 does not solve it by tuning c — it changes the factor from J_ℓ to $I + J_\ell$, so that the identity survives no matter what J_ℓ does.

C-1 · Why the loss is a length

Take the expectation of the cross-entropy under the true distribution p and add and subtract $\ln p_i$ inside the sum:

$$\mathbb{E}_p[-\ln q_i] = -\sum_i p_i \ln q_i = -\sum_i p_i \ln p_i + \sum_i p_i \ln \tfrac{p_i}{q_i} = H(p) + \mathrm{KL}(p\|q).$$

So the expected loss is the entropy of the data plus the divergence of the model from it. The second term is what training reduces, and it is non-negative and zero only when $q = p$. The first term does not contain q at all: no model, of any size, trained on any amount of data, can push the loss below it.

In a language model $H(p)$ is the entropy of language itself — the genuine unpredictability of the next token given everything before it, which no amount of modelling removes because it is a property of the text and not of the predictor. That floor is the E term in Chapter 7's loss surface $L(N, D) = AN^{-a} + BD^{-b} + E$, and this is why it appears there as a constant rather than as something to be optimized away. It is also why it drops out of the compute-optimal derivation entirely: a term with no N and no D in it vanishes on differentiation, so the best split of a budget between model and data does not depend on how predictable language is.

Part II

Chapter 3 · The Geometry of Training

B-1 · The valley, end to end

(a) $\eta < 2/\lambda_{\max} = 2/200 = 0.01$. (b) At $\eta = 0.009$: along the steep axis the factor is $|1 - 0.009(200)| = 0.8$; along the shallow axis $|1 - 0.009(2)| = 0.982$. (c) The shallow direction needs $\ln(0.01)/\ln(0.982) = 254$ steps to contract a hundredfold. The steep direction reaches the same contraction in $\ln(0.01)/\ln(0.8) = 21$ steps — and then spends the remaining 233 steps oscillating about the valley floor, contributing nothing. That idle majority is what conditioning costs.

B-2 · The square-root promise

For large κ, $\rho_{\mathrm{GD}} = \frac{\kappa-1}{\kappa+1} \approx 1 - 2/\kappa$, so $\ln(1/\rho_{\mathrm{GD}}) \approx 2/\kappa$. Similarly $\rho_{\mathrm{mom}} \approx 1 - 2/\sqrt{\kappa}$ gives $\ln(1/\rho_{\mathrm{mom}}) \approx 2/\sqrt{\kappa}$. Since step count is $\ln F/\ln(1/\rho)$, the ratio of counts is $\frac{2/\sqrt{\kappa}}{2/\kappa} = \sqrt{\kappa}$.

B-3 · Compounded initialization error

Per layer the variance multiplier is $m\sigma^2 = 500 \times (1/1000) = 0.5$. Over twelve layers the activation variance is multiplied by $0.5^{12} = 1/4096 = 2.4 \times 10^{-4}$ relative to the variance-preserving case. A single factor of $\sqrt{2}$ in σ, compounded twelve times, costs nearly four orders of magnitude.

C-1 · Reconditioning by rescaling

If the update along axis i becomes $x_i \leftarrow (1-\eta)x_i$, every axis contracts at the same rate, so the effective condition number is 1 and one well-chosen step suffices in principle. Real second-order methods approximate the inverse curvature to achieve this; adaptive optimizers approximate it far more cheaply with per-coordinate scale estimates accumulated from gradient history. Both are answering the question this chapter posed: the step size should not be hostage to the single steepest direction.

Chapter 4 · The Architecture of Depth

B-1 · The path ensemble at 100 blocks

Path length is $\mathrm{Binomial}(100, \frac{1}{2})$: mean 50, variance 25, standard deviation 5. With continuity correction the endpoints standardize to $(39.5-50)/5 = -2.1$ and $(60.5-50)/5 = +2.1$. The enclosed fraction is $2\Phi(2.1) - 1 = 2(0.9821) - 1 = 0.964$ — about 96% of paths traverse 40 to 60 blocks.

B-2 · Where the saving lives

Plain: $18C^2$. Bottleneck: $C(C/c) + 9(C/c)^2 + (C/c)C = \frac{2C^2}{c} + \frac{9C^2}{c^2}$. Ratio $= \frac{18}{2/c + 9/c^2} = \frac{18c^2}{2c+9}$. At $c=2$: $72/13 = 5.5$. At $c=4$: $288/17 = 16.9$. For large c the ratio approaches $9c$, growing without bound — but so does the information lost in the narrow middle, which is why compression factors in practice stay modest.

B-3 · Reach versus depth

Dilated: $2^{n+1} - 1 \geq 255 \Rightarrow n = 7$ (giving exactly 255). Undilated: $2n + 1 \geq 255 \Rightarrow n = 127$. A single kernel of that field holds $255^2 = 65{,}025$ weights per channel pair against the stack's $7 \times 9 = 63$ — a ratio of about 1,030.

B-4 · Drift, and where normalization goes

Accumulated variance $60 \times 0.02 = 1.2$, so the standard deviation is $\sqrt{1.2} = 1.10$ — already larger than a unit-scale signal after sixty blocks.

Putting normalization on the skip path replaces the identity in Derivation 4.1 with a scaling: the $k = 0$ term becomes $\prod_\ell (1/\sqrt{\ell})\, I$ rather than I, and that product decays without bound. The gradient floor, which was the point of the whole construction, is gone. So the drift argument does justify controlling the scale, and does not justify controlling it *there* — the branch is the only place where rescaling costs nothing, because the branch is the term the expansion can afford to lose.

B-5 · The cost of one design decision

Single 7×7: $49 \times 256^2 = 3{,}211{,}264$ weights and the same count of multiply–accumulates. Three 3×3: $3 \times 9 \times 256^2 = 1{,}769{,}472$ — the same 7×7 receptive field for 55% of the cost, plus two extra nonlinearities. Bottleneck at $c = 4$: $256 \cdot 64 + 9 \cdot 64^2 + 64 \cdot 256 = 69{,}632$, but its receptive field is only 3×3.

Ranked by cost the bottleneck wins by a factor of 46, and the ranking is misleading: it is not doing the same job. Against equal receptive field the stack of three is the right comparison and it is both cheaper and more expressive. The bottleneck buys depth per unit cost rather than reach, which is why real networks use both — stacks of small kernels for reach, bottlenecks for depth.

C-1 · Why identity mappings help twice

In Derivation 4.1 the surviving $k = 0$ term is the identity only if the skip path is genuinely unmodified; inserting any transformation there replaces I with that transformation's Jacobian, and a product of those can again vanish. Keeping the skip clean preserves the floor exactly. In Worked Example 4.2 the drift comes from the accumulated branch outputs; placing normalization inside the branch controls each contribution before it is added, whereas normalizing the skip would rescale the carefully preserved identity signal itself. The unifying principle: *keep the pathway that guarantees signal flow free of anything that could attenuate it, and place all control on the pathway that adds new content.*

Chapter 5 · Memory and Gates

B-1 · Two places to put noise

(a) Recurrent-path dropout at keep-probability 0.9 over 200 steps leaves $0.9^{200} = e^{-21.1} = 7 \times 10^{-10}$ of the signal — total annihilation. (b) Vertical-only dropout perturbs a given piece of information once per layer, so three times, regardless of the sequence's length. The vertical design preserves memory; the difference is that one exposure count scales with time and the other with depth.

B-2 · Designing a horizon

Require $\ln(0.01)/\ln f = 500 \Rightarrow \ln f = -0.00921 \Rightarrow f = 0.9908$. The bias is $b = \ln \frac{0.9908}{0.0092} = \ln(107.7) = 4.68$.

B-3 · Anatomy of a model

Embedding $1024 \times 30{,}000 = 30.72\text{M}$. Each recurrent layer $4 \times 1024 \times (1024 + 1024 + 1) = 4096 \times 2049 = 8.39\text{M}$, so three layers give 25.18M. Output projection $1024 \times 30{,}000 + 30{,}000 = 30.75\text{M}$. Total $\approx 86.7\text{M}$, of which embedding plus projection is 61.5M — about 71%. Since that fraction scales with vocabulary rather than with computational depth, including it would corrupt any attempt to relate capability to model size; hence scaling studies count non-embedding parameters.

C-1 · The carousel as a limit

At $f = 1$, $f^t = 1$ for every t: the horizon in Derivation 5.1 is unbounded, since $\ln f = 0$ makes the required t infinite. A cell fixed at $f = 1$ could never discard information, so its state would accumulate every input it ever saw and the useful signal would be swamped — and it could not represent tasks requiring the state to reset. Making f input-dependent lets the network hold near 1 when carrying information matters and drop it when the context changes, so the horizon becomes a per-token decision rather than an architectural constant.

Chapter 6 · Attention

B-1 · Which regime

Attention $= 2n^2 d = 2(8192)^2(1024) = 1.37 \times 10^{11}$. Feed-forward $= 8nd^2 = 8(8192)(1024)^2 = 6.87 \times 10^{10}$. The ratio is $2:1$ — attention dominates at this length. They balance at $n = 4d = 4096$, which is where the model crosses from feed-forward-dominated to attention-dominated.

B-2 · The reversal trick in full

Unreversed, source token j is read at step j and target token j emitted at step $n + j$, so every lag is n. Reversed, source token j is read at step $n - j + 1$, so the lag is $(n + j) - (n - j + 1) = 2j - 1$. The mean is $\frac{1}{n}\sum_{j=1}^{n}(2j-1) = \frac{n^2}{n} = n$ — unchanged. The maximum rises to $2n - 1$. It aids optimization because the earliest alignments become nearly immediate, giving gradient descent a foothold it can learn from at once; once the early alignments are established the rest follow. It does not aid representation, because the total distance information must travel is provably identical.

B-3 · Temperature cannot reorder

For $T > 0$, the map $z \mapsto z/T$ is strictly increasing, so $z_a > z_b \Rightarrow z_a/T > z_b/T$. The softmax is a strictly increasing function of its argument relative to the others (the exponential is increasing and the shared denominator is positive), so $P(a) > P(b)$ for every positive T. Temperature therefore alters only how concentrated the distribution is, never which candidate leads.

C-1 · One instinct, three chapters

The principle: *hold the variance of a signal at order one as it crosses any operation, because both vanishing and saturating regimes destroy gradients.* In Chapter 3, the quantity is the pre-activation of a layer, held at unit variance by $\sigma^2 = 1/m$; otherwise activations shrink or explode geometrically with depth. In Chapter 4, the quantity is the accumulated residual signal, whose $\sqrt{L}$ drift is contained by normalizing inside the branch; otherwise the signal outgrows the scale the later layers were tuned for. In Chapter 6, the quantity is the attention logit, held at unit variance by dividing by $\sqrt{d}$; otherwise the softmax saturates and its gradient vanishes. Same instinct, three operations.

Chapter 7 · The Economics of Scale

B-1 · The full allocation

Substituting $D = C/(kN)$ gives $L(N) = AN^{-a} + B(k/C)^b N^b$. Differentiating: $-aAN^{-a-1} + bB(k/C)^b N^{b-1} = 0$. Multiplying by N and rewriting the second term as bBD^{-b} yields $aAN^{-a} = bBD^{-b}$. Combining with $N^a \propto D^b$ and $ND \propto C$: $D^{1+b/a} \propto C$, so $D \propto C^{a/(a+b)}$ and $N \propto C^{b/(a+b)}$. The bare-term balance is wrong because the constraint's exchange rate between N and D depends on each term's responsiveness, which is exactly its exponent.

B-2 · Re-budgeting

$C = 6(7 \times 10^{10})(1.4 \times 10^{12}) = 5.88 \times 10^{23}$. At ratio 20, $C = 120N^2 \Rightarrow N^2 = 4.9 \times 10^{21} \Rightarrow N = 7 \times 10^{10}$. The model is *already* at 70 B with $D/N = 1.4 \times 10^{12}/7 \times 10^{10} = 20$ exactly — it is compute-optimal as given, neither over- nor under-trained. (Contrast Worked Example 7.2, where the ratio was under 2 and the re-budgeting cut the model by a factor of three.)

B-3 · The cost of a halving

Requiring $k^{-a} = \frac{1}{2}$ gives $k = 2^{1/a}$; at $a = 0.076$, $k = 2^{13.16} \approx 9{,}100$. Since compute scales with N (and, at fixed ratio, with N^2), each halving of this loss term multiplies the training bill by four orders of magnitude or more. Steady progress on loss is therefore exponentially expensive in compute — the arithmetic behind the field's escalating training budgets.

C-1 · When the ratio drifts

$D/N \propto C^{(a-b)/(a+b)}$. If $a < b$ the exponent is negative and the optimal tokens-per-parameter ratio *falls* as budgets grow; if $a > b$ it *rises*. Interpretation: the exponent measures how quickly a term responds to its resource, so a larger a means parameters buy improvement faster, and the optimum then shifts spending toward the *slower*-improving resource — data — to keep the weighted marginal returns equal. Real recipes have drifted toward far more tokens per parameter, but Chapter 17 shows the dominant reason is not this drift at all: it is the serving term entering the objective.

Chapter 8 · The Machinery of Scale

B-1 · Deriving the bubble

Micro-batch m enters stage 1 at time $m - 1$ and advances one stage per unit, so it exits stage P at $m + P - 1$; the last exits at $M + P - 1$. Useful work is MP stage-units; capacity is $P(M + P - 1)$; the bubble is $1 - \frac{MP}{P(M+P-1)} = \frac{P-1}{M+P-1}$. At $(8, 1)$: $7/8 = 87.5\%$ idle — naive model parallelism wastes seven eighths of the fleet. At $(8, 32)$: $7/39 = 17.9\%$ — still nearly a fifth idle even with thirty-two micro-batches.

B-2 · Sizing micro-batches

$\frac{11}{M+11} \leq 0.08 \Rightarrow M + 11 \geq 137.5 \Rightarrow M \geq 127$.

B-3 · The wall

$16 \times 1.75 \times 10^{11} = 2.8 \times 10^{12}$ bytes $= 2{,}800$ GB, requiring $2800/80 = 35$ accelerators for the state alone. Computing the unsharded figure first is right because it is the quantity that decides whether the run is possible at all; the sharded per-device number tells you nothing about feasibility until you know how many devices the total demands.

C-1 · Two penalties at once

At fixed M, raising P increases the bubble $\frac{P-1}{M+P-1}$ monotonically toward 1, while per-accelerator memory falls roughly as $1/P$. Pipelining therefore trades time for memory at a worsening rate. Real systems combine it with tensor parallelism (splitting individual operations, which costs communication rather than idle time) and with state sharding (which costs communication but no idle time at all), so that the memory requirement can be met without pushing P into the regime where the bubble dominates. The general lesson: when one axis of parallelism has a penalty that grows with its degree, use several axes at modest degree rather than one at high degree.

Chapter 9 · Measurement

B-1 · The shape of emergence

At $k = 8$: $p = 0.5 \to 0.0039$; $0.7 \to 0.058$; $0.8 \to 0.168$; $0.9 \to 0.430$; $0.95 \to 0.663$. The steepest rise is between 0.8 and 0.95, bracketing the 50% frontier $2^{-1/8} = 0.917$. A per-token log-loss curve shows no cliff because it measures p itself, which moves smoothly; the cliff is created by raising that smooth quantity to the eighth power and comparing against a threshold.

B-2 · Inverting a leak

$0.90 = c + 0.65(1 - c) = 0.65 + 0.35c \Rightarrow c = 0.714$. Then $P(\text{contaminated} \mid \text{correct}) = 0.714/0.90 = 0.794$. Nearly four fifths of the correct answers would be recall rather than skill — the headline score describes the test set's provenance more than the model.

B-3 · Voting's threshold

Majority of three is correct with probability $3p^2(1-p) + p^3 = 3p^2 - 2p^3$. This exceeds p iff $3p - 2p^2 > 1$, i.e. $2p^2 - 3p + 1 < 0$, i.e. $(2p-1)(p-1) < 0$. Since $p < 1$ makes the second factor negative, the product is negative exactly when $2p - 1 > 0$, i.e. $p > \frac{1}{2}$.

C-1 · Two illusions, one theme

Emergence: the model owns p; the ruler owns k and the exact-match threshold. Attributing the cliff to the model asserts a discontinuous capability change that the evidence does not support. *Contamination:* the model owns its clean accuracy; the ruler owns c, a property of the test set's construction. Attributing the inflation to the model overstates skill by exactly $c(1-p)$. *Voting:* the model owns p; the procedure owns the number of samples and the aggregation rule. Attributing the gain to the model claims a capability increase where only variance was reduced. In all three, the observed number is a function of model and measurement jointly, and the error is always the same: crediting the model with the measurement's contribution.

Chapter 10 · Compression and Occam

B-1 · Model selection, fully

P: $45 + 300 = 345$. Q: $225 + 95 = 320$. R: $450 + 0 = 450$. MDL selects **Q**; the zero-error model R loses by 130 bits. Overfitting here is the quantity by which a model's parameter cost exceeds the residual bits those parameters save — a number, not a judgment.

B-2 · The length prior in action

Scale weights by 2^{14}: the four programs carry $2^8 = 256$, $2^5 = 32$, $2^5 = 32$, and 1; total 321. Those predicting A hold $256 + 32 = 288$, so $P(A) = 288/321 = 0.897$. General rule: a program δ bits longer than another holds $2^{-\delta}$ times its weight. Proof: weights are $2^{-\ell}$, so the ratio of two is $2^{-\ell_1}/2^{-\ell_2} = 2^{\ell_2 - \ell_1}$.

B-3 · Approaching the threshold

With $n = 200$: $p = 100 \to 100/99 = 1.01$; $p = 150 \to 150/49 = 3.06$; $p = 190 \to 190/9 = 21.1$; $p = 195 \to 195/4 = 48.8$. The risk accelerates violently as $p \to n - 1$. The implication is that model sizes near the number of training samples are the worst possible choice — and that the classical instinct to stop growing the model before it interpolates lands you precisely in the danger zone.

B-4 · Precision is what costs

(a) $\ln(1/0.5) + \frac{0.25}{2} - 0.5 = 0.693 + 0.125 - 0.5 = 0.318$ nats. (b) $\ln(1/0.05) + \frac{0.0025}{2} - 0.5 = 3.00 + 0.001 - 0.5 = 2.50$ nats. The precise weight costs nearly eight times as much. A description-length penalty therefore leaves imprecise every weight whose exactness does not purchase a commensurate reduction in residual bits — which is most of them.

B-5 · The same corpus, three tokenizers

Bits per character: $1.9/4.5 = 0.422$; $1.5/3.2 = 0.469$; $2.4/6.0 = 0.400$. Ranked best to worst: the third, the first, the second. On 5×10^9 characters: 264 MB, 293 MB, 250 MB respectively (bits divided by 8).

A per-token leaderboard ranks them 1.5, 1.9, 2.4 — exactly backwards. The best compressor has the worst per-token loss, because its tokens are the largest. Any comparison across tokenizers that does not divide by γ is measuring the vocabulary.

B-6 · What a bit of precision is worth

With $\mu = 0$ the cost is $\ln(1/\sigma_q) + \sigma_q^2/2 - 1/2$.

$\sigma_q = 1 \to 0$; $\quad 0.5 \to 0.693 + 0.125 - 0.5 = 0.318$; $\quad 0.25 \to 1.386 + 0.031 - 0.5 = 0.918$; $0.1 \to 2.303 + 0.005 - 0.5 = 1.808$; $0.01 \to 4.605 + 0.00005 - 0.5 = 4.105$ nats.

Once σ_q is small the $\sigma_q^2/2$ term is negligible and the cost is $\ln(1/\sigma_q) - 1/2$, so halving σ_q adds exactly $\ln 2$ in the limit. Check it on successive halvings: $0.5 \to 0.25$ adds 0.600, $0.25 \to 0.125$ adds 0.669, $0.125 \to 0.0625$ adds 0.688 — converging on $\ln 2 = 0.693$ from below. Each further bit of precision therefore costs the same as the last rather than less, which is what makes precision worth buying only where it pays — there is no volume discount.

C-1 · One currency, eight costumes

Model selection: model bits are the parameter encoding, residual bits the unexplained data. Hypothesis odds: model bits are the hypothesis specification, residual bits the negative log-likelihood. Weight precision: model bits are the coding cost of the posterior against the prior; residual bits are the loss the weight's imprecision causes. Cross-entropy as file size: *only* residual bits — the model is assumed already shared. The length prior: *only* model bits, since the programs considered fit perfectly. Typicality: the model is the constraint set, and residual bits are the deficiency between a typical member's length and this member's. Grokking: two competing model encodings at identical (zero) residual. Double descent: the parameter count is model bits and the noise-fitting is residual, with the divergence marking where residual savings stop justifying parameters. A single principle wears this many disguises because *any* inference problem can be posed as choosing a description, and the only currency a description has is its length. The reading lesson: when a new result seems novel, ask what it is trading between model bits and residual bits — often the novelty is in the encoding, not the principle.

Part III

Chapter 11 · The Price of a Token

B-1 · Sizing a deployment

(a) Weights: $7 \times 10^{10} \times 2 = 140$ GB. (b) Cache: $2 \times 80 \times 8 \times 128 \times 8192 \times 16 \times 2 = 4.3 \times 10^{10}$ bytes $= 43$ GB. (c) Sum $= 183$ GB. (d) At 80 GB each, three accelerators — barely, with no room for activations, so four in practice. Attack the weights first: quantization to 8 bits halves the dominant term and simultaneously doubles the decode ceiling, whereas cache reductions help only at long context or large batch.

B-2 · The batching crossover

Batch-B decode performs about $2NB$ operations while still moving about $2N$ bytes of weights (read once, reused across the batch), so intensity $\approx B$. Compute-bound requires $B \gtrsim 156$ on the reference machine. As batch size approaches that, throughput rises nearly linearly while per-request latency degrades, because each request waits for the whole batch — the fundamental throughput-versus-latency tension in serving.

B-3 · What quantization actually buys

140 / 70 / 35 GB at 16 / 8 / 4 bits, giving ceilings of $2000/140 = 14.3$, 28.6, and 57 tokens per second. Quantization improves the *bytes-moved* budget and leaves the operation count essentially unchanged. Decode is bandwidth-bound, so shrinking bytes raises its ceiling proportionally; prefill is compute-bound, so shrinking bytes does not help it. Same change, opposite consequences, decided entirely by which side of the ridge the workload sits on.

C-1 · Two eras, two scarcities

Take a model trained compute-optimally at, say, 70 B parameters, versus a smaller 35 B model trained on proportionally more data to equal quality. Chapter 7's objective is indifferent to serving and picks whichever minimizes training cost. But by this chapter's formulas the 35 B model has half the weight memory, twice the decode ceiling, and a proportionally smaller cache — advantages paid on every request forever. The missing term is the serving cost: the objective must become $6ND_{\text{train}} + 2NT$ for a lifetime of T generated tokens, which is exactly Chapter 17's construction.

Chapter 12 · Bytes over FLOPs

B-1 · Traffic accounting

Score-matrix traffic (written once, read once) is $2 \times 2n^2 = 4n^2$ bytes $= 4(4096)^2 = 6.7 \times 10^7$. The unavoidable traffic for queries, keys, values, and output is $4 \times 2nd = 8(4096)(128) = 4.2 \times 10^6$. The ratio is $\frac{4n^2}{8nd} = \frac{n}{2d} = 16:1$. It grows linearly in n and falls inversely with d, so the technique matters most at long context and modest head dimension — exactly the regime that became standard.

B-2 · Proving exactness

With $p = (0.3, 0.7)$, $q = (0.5, 0.5)$: acceptance for symbol 1 is $\min(1, 0.6) = 0.6$; for symbol 2, $\min(1, 1.4) = 1$. Accepted mass $= 0.5(0.6) + 0.5(1) = 0.8$, so rejection occurs with probability 0.2. The residual $\max(0, p - q) = (0, 0.2)$ normalizes to $(0, 1)$. Emitted probabilities: symbol 1 gets $0.5(0.6) = 0.3$ ✓; symbol 2 gets $0.5(1) + 0.2(1) = 0.7$ ✓. Exactly p.

B-3 · Choosing the draft length

With $\alpha = 0.8$, $c = 0.1$: $k = 2$ gives $2.44/1.2 = 2.03\times$; $k = 4$ gives $3.36/1.4 = 2.40\times$; $k = 8$ gives $4.33/1.8 = 2.41\times$; $k = 16$ gives $4.89/2.6 = 1.88\times$. The optimum is a broad plateau around $k \approx 6$ to 9. The marginal condition $\alpha^{k+1} \approx c$ gives $0.8^{k+1} = 0.1 \Rightarrow k + 1 = 10.3 \Rightarrow k \approx 9$, consistent with the plateau's upper end.

C-1 · Why decode and not prefill

Prefill already processes many positions in one pass, so its arithmetic intensity is high and it sits above the ridge point — the accelerator is already busy, and there is no idle capacity for speculation to exploit. Decode sits far below the ridge, so verifying k positions at once costs almost the same as verifying one, and the speculation converts wasted capacity into tokens. The general condition: speculation helps a workload whose per-step arithmetic intensity is far below the ridge point *and* whose steps are sequentially dependent, so that batching within a single request is otherwise impossible. Autoregressive sampling from any large model qualifies; so does sequential simulation where a cheap surrogate can propose several steps for a costly verifier to check in parallel.

Chapter 13 · Sparsity and Thrift

B-1 · Reading a sparse model honestly

Shared components: $0.2 \times 140 = 28$ B. Expert parameters: 112 B across 64 experts $= 1.75$ B each; top-2 routing activates 3.5 B. Active total ≈ 31.5 B. Behaves like the *active* figure: training operations and decode arithmetic. Behaves like the *total* figure: serving memory, since every expert must be resident. The KV cache belongs to neither — it depends on layers, key/value heads, head dimension, and context, and is unaffected by expert count.

B-2 · Outliers and bits

MSE scales as R^2, so the range moving from 1 to 6 multiplies the error by 36. Each additional bit quarters the error, so restoring it needs $4^b \geq 36 \Rightarrow b = \log_4 36 = 2.58$, i.e. 3 more bits everywhere. Handling outliers separately is cheaper because the extra precision is then spent only on the few channels that need it, rather than on every weight in the matrix — the same allocation logic as Chapter 10's account of paying for precision only where it buys accuracy.

B-3 · Budgets, side by side

(a) Weights: $1.3 \times 10^{10} \times 2 = 26$ GB. (b) Full fine-tuning optimizer state: $16 \times 1.3 \times 10^{10} = 208$ GB. (c) Rank-16 adaptation of matrices totalling 30% of the model makes well under one percent of parameters trainable, so optimizer state falls to roughly 1–2 GB. To fit training on a single 80 GB accelerator, the binding constraint is (b), so low-rank adaptation is the necessary technique; quantizing the frozen base then reduces (a) as well, and the two compose.

C-1 · The balance loss, minimized

With $E = 2$ and $f_1 = P_1 = x$, $f_2 = P_2 = 1 - x$: $\mathcal{L} = 2[x^2 + (1-x)^2]$. Differentiating, $4x - 4(1-x) = 0 \Rightarrow x = \frac{1}{2}$, giving $\mathcal{L} = 2(\frac{1}{4} + \frac{1}{4}) = 1$. The second derivative is positive, so this is the minimum. Why both mechanisms: the auxiliary loss is a soft gradient signal that shapes the router over training but guarantees nothing at any particular step, so alone it would permit transient severe imbalance — and early in training, when the router is near-random, that imbalance can be self-reinforcing as favored experts improve fastest. The capacity limit alone would prevent overload but supplies no gradient pressure toward balance, so the router would keep proposing imbalanced assignments and the system would silently drop tokens forever. Pressure plus a hard limit gives both a direction and a bound.

B-1 · The clip table

With $\epsilon = 0.2$, clip range $[0.8, 1.2]$.

$A = +1$: at $\rho = 0.7$, $\min(0.7, 0.8) = 0.7$, unclipped branch, gradient alive. At $\rho = 1.0$, both give 1.0, alive. At $\rho = 1.3$, $\min(1.3, 1.2) = 1.2$, clipped, **gradient dead**.

$A = -1$: at $\rho = 0.7$, $\min(-0.7, -0.8) = -0.8$, clipped, **gradient dead**. At $\rho = 1.0$, both -1.0, alive. At $\rho = 1.3$, $\min(-1.3, -1.2) = -1.3$, unclipped, alive.

The pattern: the gradient dies exactly when the policy has already moved *beyond* the trust region in the direction the advantage favors. A single update can therefore never be rewarded for moving further than ϵ in the profitable direction, which bounds how far one batch can push the policy.

B-2 · The collapse, reproduced

The optimum is $\pi^*(y \mid x) = \pi_{\text{ref}}(y \mid x) e^{r(x,y)/\beta} / Z(x)$. Inverting: $r(x, y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x)$. Substituting into $P(y_w \succ y_l) = \sigma(r(x, y_w) - r(x, y_l))$, the two $\beta \log Z(x)$ terms appear with opposite signs and cancel, because Z depends on x alone and both responses share the prompt. The loss is $-\log \sigma\big(\beta \log \frac{\pi(y_w)}{\pi_{\text{ref}}(y_w)} - \beta \log \frac{\pi(y_l)}{\pi_{\text{ref}}(y_l)}\big)$. If the two responses came from different prompts, the normalizers would be $Z(x_1)$ and $Z(x_2)$, which do not cancel — and the method would be intractable, since Z sums over all possible responses.

B-3 · The cost of a critic

Policy alone: $20 \times 7 \times 10^9 = 1.4 \times 10^{11}$ bytes $= 140$ GB. Policy plus a same-sized critic: 280 GB. The ratio is 2 — the critic is half the total bill, which is precisely the resource that group-relative advantage estimation recovers by computing the baseline from sampled rewards instead of a learned network.

B-4 · Degenerate groups

If every $r_i = r$, then $\bar{r} = r$ and each numerator $r_i - \bar{r} = 0$, so every advantage is zero regardless of r (the standard deviation is also zero, but the numerator vanishing is what matters — the update is identically zero). Consequence for curriculum: prompts the model always solves and prompts it never solves both produce all-equal groups and contribute nothing. Training must therefore be concentrated on problems near the model's current success boundary, and the boundary moves as the model improves — so the curriculum must be continuously re-selected rather than fixed in advance.

C-1 · Two cancellations, one technique

Both are instances of: *a quantity that enters an expression only through a difference of two terms in which it takes the same value is unidentifiable from, and irrelevant to, that expression.* In Section 14.2 the additive reward constant is shared by both responses and cancels in $r_w - r_l$, so no data can determine it. In Section 14.4 the log-normalizer is shared by both responses to the same prompt and cancels in the same difference, so its intractability does not matter. A third instance in this book: Chapter 7's irreducible loss term E, which is constant in both N and D and therefore vanishes on differentiation, so the compute-optimal allocation is entirely independent of it. A fourth: Chapter 10's normalization of posterior weights, where a common scaling factor across all programs cancels in the ratio — which is exactly why we may rescale by $2^{\ell_{\max}}$ and work in integers.

Chapter 15 · Thinking at Inference Time

B-1 · Matched compute, honestly

Budget 24: the small model gets 24 samples, coverage $1 - 0.85^{24} = 1 - e^{-3.90} = 0.980$. The large model's single attempt exhausts the same budget and delivers 0.55, with nothing to select between. Delivered accuracy for sampling: $v = 1.0 \to 0.980$; $v = 0.8 \to 0.784$; $v = 0.6 \to 0.588$. They tie when $0.980v = 0.55$, i.e. $v = 0.561$. Above that selector reliability, sampling the small model wins; below it, the large model's single answer is better.

B-2 · The voting threshold

Majority of three: $3p^2(1-p) + p^3 = 3p^2 - 2p^3$. Exceeds p iff $3p - 2p^2 > 1$ iff $(2p-1)(p-1) < 0$ iff $p > \frac{1}{2}$. At $p = 0.45$: $3(0.2025) - 2(0.0911) = 0.425 < 0.45$ — voting made it worse. At $p = 0.55$: $3(0.3025) - 2(0.1664) = 0.575 > 0.55$ — better. The threshold is exact, and below it the procedure reliably amplifies error.

B-3 · When to stop sampling

The $(k+1)$-th sample converts a failure into a success with probability $(1-p)^k p$, worth $V(1-p)^k p$, against a cost c. Continue while $Vp(1-p)^k > c$. With $p = 0.1$, $V = 100$, $c = 1$: $10(0.9)^k > 1 \Rightarrow 0.9^k > 0.1 \Rightarrow k < \ln(0.1)/\ln(0.9) = 21.9$. So draw about 22 samples; beyond that the marginal sample costs more than the expected value it adds.

C-1 · Why temperature zero breaks everything here

At temperature approaching zero the sampler becomes deterministic: every sample is the argmax, so all k samples are identical. Coverage collapses from $1-(1-p)^k$ to simply p, because the samples are perfectly correlated rather than independent — the formula's premise has failed. Temperature therefore controls the *diversity* of samples, and diversity is precisely the resource that inference-time scaling consumes: without it there is nothing for a selector to select among. Connecting to Chapter 6: temperature is a monotone rescaling of the logits, so it cannot change which candidate is most probable — it changes only how much probability mass sits away from that leader. That mass is exactly what makes repeated sampling explore, which is why the quantity temperature controls is the one that matters here, and why very low temperatures and inference-time scaling are fundamentally incompatible.

Chapter 16 · Generation by Denoising

B-1 · Composing two steps

$\alpha_1 = 0.99$, $\alpha_2 = 0.97$, so $\bar{\alpha}_2 = 0.9603$. The signal coefficient is $\sqrt{0.9603} = 0.980$ and the noise coefficient $\sqrt{1-0.9603} = \sqrt{0.0397} = 0.199$; the squared coefficients sum to $0.9603 + 0.0397 = 1$ ✓. The induction step: assuming $x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}x_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\epsilon$, substituting into $x_t = \sqrt{\alpha_t}x_{t-1} + \sqrt{1-\alpha_t}\,\epsilon_t$ gives signal coefficient $\sqrt{\alpha_t\bar{\alpha}_{t-1}} = \sqrt{\bar{\alpha}_t}$ and total noise variance $\alpha_t(1-\bar{\alpha}_{t-1}) + (1-\alpha_t) = 1-\bar{\alpha}_t$.

B-2 · Designing a schedule

Require $(1-\beta)^{200} = 0.5 \Rightarrow 1-\beta = 0.5^{1/200} = e^{-0.003466} = 0.99654 \Rightarrow \beta = 0.00346$. At $t = 500$: $\bar{\alpha} = 0.99654^{500} = e^{-1.733} = 0.177$, so the signal-to-noise ratio is $0.177/0.823 = 0.215$.

B-3 · The cost of guidance

Guidance requires both a conditional and an unconditional prediction at each step, so it doubles the per-step cost. With a stride of 20 on a 1000-step schedule there are 50 steps, hence $50 \times 2 = 100$ network evaluations. Unguided, unstrided sampling would take 1000. So guided strided sampling is ten times cheaper than the naive baseline despite paying a factor of two for guidance — the stride is doing all the work and more.

C-1 · The narrow middle, three times

Bottleneck block: channels are compressed by 1×1 convolutions; the expensive spatial convolution runs in the narrow space; the budget saved is arithmetic; compressing too far starves the block's representational capacity. *Expert routing:* the token population is compressed — each token uses only k of E experts; the router does the compressing and the residual stream carries information around it; the budget saved is per-token compute; routing too sparsely starves experts of training signal and invites dropped tokens. *Latent diffusion:* the image is compressed by an encoder; the entire denoising process runs in the latent; the decoder restores resolution; the budgets saved are compute and activation memory; compressing too aggressively discards detail the decoder cannot invent. The principle: *perform the expensive, repeated computation in the smallest representation that still carries what that computation needs, and let cheap machinery move between that representation and the full one.*

Chapter 17 · The Whole Lifecycle

B-1 · A deployment decision

P: $(ND) = 2 \times 10^{23}$, so training $= 1.2 \times 10^{24}$. Q: $(ND) = 2.4 \times 10^{23}$, training $= 1.44 \times 10^{24}$.

At $T = 10^{12}$: $C_P = 1.2 \times 10^{24} + 2 \times 10^{23} = 1.4 \times 10^{24}$; $C_Q = 1.44 \times 10^{24} + 8 \times 10^{22} = 1.52 \times 10^{24}$. **P wins.**

At $T = 10^{14}$: $C_P = 1.2 \times 10^{24} + 2 \times 10^{25} = 2.12 \times 10^{25}$; $C_Q = 1.44 \times 10^{24} + 8 \times 10^{24} = 9.44 \times 10^{24}$. **Q wins.**

Break-even: $T^* = \frac{6(2\times10^{23} - 2.4\times10^{23})}{2(4\times10^{10} - 10^{11})} = \frac{-2.4\times10^{23}}{-1.2\times10^{11}} = 2 \times 10^{12}$ tokens. Recommend Q if you expect to serve more than about two trillion tokens over the model's life — a threshold most production deployments cross quickly, so Q unless the model is experimental or narrowly used.

B-2 · Where the interior optimum sits

At $T = 5 \times 10^{12}$: (80 B, 1 T) gives $4.8 \times 10^{23} + 8 \times 10^{23} = 1.28 \times 10^{24}$. (40 B, 3 T) gives $7.2 \times 10^{23} + 4 \times 10^{23} = 1.12 \times 10^{24}$. (20 B, 10 T) gives $1.2 \times 10^{24} + 2 \times 10^{23} = 1.4 \times 10^{24}$. (10 B, 40 T) gives $2.4 \times 10^{24} + 10^{23} = 2.5 \times 10^{24}$. Best is **(40 B, 3 T)**. The large extreme loses because serving a big model over a long life is expensive; the small extreme loses because matching quality with very few parameters demands so much data that training cost explodes. The optimum is interior because one term grows and the other shrinks as size falls.

B-3 · Serving beyond FLOPs

Three further advantages of the 35 B option: its weight memory is halved (70 GB against 140 GB at half precision), which may change how many accelerators a replica needs; its batch-1 decode ceiling doubles, since the ceiling is bandwidth over weight bytes — roughly 29 tokens per second against 14 on the reference machine; and its KV cache per token is smaller in proportion to its layer count, easing long-context serving. An operations-only account understates the case because it prices only arithmetic, while the binding constraint in decoding is bandwidth and memory — the very budgets that shrink fastest as the model does.

C-1 · The unified objective

A workable form: minimize $\lambda_{\text{compute}}\left[6ND + 2NT\right] + \lambda_{\text{human}}\, H$ subject to a quality target $L(N, D, H) \le L^{\star}$, where H is the quantity of preference data and λ are per-unit prices. The empirical inputs this book has taken as given: the loss surface's coefficients and exponents A, a, B, b, E; the shape of the iso-quality curve that lets one trade N against D; and, hardest of all, how quality depends on H — the return to additional human preference data. That last is hardest because preference data is heterogeneous in quality, its value depends on where it sits relative to the model's current failures, and it interacts with the other two variables rather than contributing independently. This is exactly the boundary at which derivation must hand off to measurement, and recognizing that boundary honestly — rather than extrapolating a fitted curve past its evidence — is the most valuable judgment the subject asks for.

APPENDIX E

Notation and Glossary

What every symbol means, which ones mean more than one thing, and where each term was defined.

Look here when a symbol stops meaning what you thought it meant. Several carry more than one meaning, because the field's own conventions collide and renaming them would make this book harder to read against the papers rather than easier.

Symbols that mean one thing throughout

- N Number of parameters in a model. Ch. 2
- D Number of training tokens. Ch. 7
- C A compute budget, in floating-point operations. Ch. 7
- T Tokens generated over a deployment's lifetime. Ch. 17
- θ All parameters of a network, collected into one vector. Ch. 2
- η Step size, also called the learning rate. Ch. 2
- λ A curvature of the loss surface along one axis. Ch. 3
- H Number of attention heads; H_{kv} when keys and values are shared. Ch. 6, 11
- V Vocabulary size. Ch. 2
- s Sequence position, or sequence length. Ch. 11
- Φ The standard normal cumulative distribution; values in Section 1.3. Ch. 1
- π A policy: the model as a distribution over responses. π_{ref} is the frozen starting copy. Ch. 14
- Z A partition function — whatever a non-negative quantity must be divided by to sum to one. Ch. 14
- Δ Quantizer step size. Ch. 13
- Υ The complexity-weighted intelligence measure. Ch. 10

Symbols that carry more than one meaning

Each is redefined at first use in every chapter that uses it. If one appears without a definition nearby, that is a defect — report it.

- β**Five meanings.** A power-law exponent (Ch. 1, 5); the momentum coefficient (Ch. 2, 3); bytes per stored value (Ch. 11); the strength of the KL leash (Ch. 14); the per-step noise rate of a diffusion schedule (Ch. 16). The last two are adjacent and are the pair to watch.
- k**Seven meanings.** Kernel side (Ch. 4); the constant in $C = kND$, which is 6 (Ch. 7); number of independent parts of a task (Ch. 9); draft length (Ch. 12); the k of top-k routing (Ch. 13); number of samples drawn (Ch. 15); sampling stride (Ch. 16).
- σA weight's standard deviation (Ch. 1, 3); the logistic function $\sigma(u) = 1/(1 + e^{-u})$ (Ch. 2, 5, 14); coding widths σ_p, σ_q and a noise standard deviation (Ch. 10).
- αA loss exponent (Ch. 7); the acceptance rate of a draft token (Ch. 12); the noise-retention factor, with $\bar{\alpha}_t$ its running product (Ch. 16).
- ϵA tolerance (Ch. 1, 5); the clip width of the surrogate (Ch. 14); the noise vector a diffusion model predicts (Ch. 16).
- A, BCoefficients of the loss surface (Ch. 7); the low-rank factors of an adapter (Ch. 13); A alone is an advantage (Ch. 14); B alone is a batch size (Ch. 11). Also option labels in comparisons.
- dModel width; $d_{\text{head}} = d/H$ is the width one head works in, and it is d_{head} that the attention scaling divides by.
- LThe loss (Ch. 2, 7); the number of layers or residual blocks (Ch. 4, 8, 11).
- EThe irreducible term of the loss surface (Ch. 7); the number of experts (Ch. 13); an expectation, written $\mathbb{E}$.
- pA probability (Ch. 9, 15); the parameter count in the OLS excess risk (Ch. 10); the target distribution in speculative decoding (Ch. 12).
- γA discount factor (Ch. 1, 10); the tokenizer's compression ratio in characters per token (Ch. 2, 10).
- ρA per-step contraction rate (Ch. 3); the probability ratio π/π_{old} (Ch. 14).
- κThe condition number (Ch. 3); the additive constant a reward is identified only up to (Ch. 14).

Glossary

Where each term is defined. Terms stated in Chapter 2 are the objects the book computes with; the rest are introduced where they are first needed.

- **Activation**§2.1 — an entry of a layer's output. Activations are retained for the backward pass, which is why they occupy memory (§8.3).
- **Advantage**§14.1 — a response's reward minus a baseline. Positive for better than expected.
- **Arithmetic intensity**§11.2 — operations performed per byte moved. Compared against the machine's ridge point to decide what binds.
- **Backpropagation**§2.3 — evaluating the product of Jacobians efficiently, by sweeping backward and accumulating.
- **Baseline**§14.1 — a prediction of a prompt's typical reward, subtracted to leave only what distinguishes responses.
- **Bubble**§8.1 — the fraction of accelerator-time a pipeline spends idle filling and draining.
- **Checkpointing**§8.3 — storing activations only at segment boundaries and recomputing the rest, at $g = \sqrt{L}$.
- **Compression ratio**§2.4 — characters per token. Converts bits per token into bits per character.
- **Condition number**§3.1 — largest curvature over smallest. Sets how slowly gradient descent travels.
- **Contamination**§9.2 — the fraction of a test set seen during training.
- **Coverage**§15.1 — the probability that a correct answer is somewhere among k samples. Not accuracy.
- **Critic**§14.1 — a learned network supplying the baseline, roughly the policy's size, and the thing group-relative methods delete.
- **Cross-entropy**§2.2 — $-\ln q$ for the outcome that occurred; literally a code length (§10.2).
- **Decode**§11.1 — generating one token at a time. Memory-bound.
- **Forget gate**§5.1 — the factor multiplying a recurrent cell's memory each step, and therefore its gradient horizon.

- **Head**§6.1 — one of H parallel attention computations, each of width d/H. Splitting is free.
- **Jacobian**§2.3 — the matrix of partial derivatives of a vector output with respect to a vector input.
- **KL divergence**§2.6 — the extra cost of coding p with a code built for q. Non-negative, zero only at equality, asymmetric.
- **KV cache**§11.1 — stored keys and values for previous positions, so they need not be recomputed. Linear in context.
- **Logit**§2.2 — an unconstrained score, before the softmax.
- **MFU**§11.3 — model FLOP utilization: useful operations per second over the machine's peak.
- **Mixed precision**§8.2 — half-precision arithmetic with a full-precision master copy of the weights.
- **Multiply–accumulate**§1.8 — one multiplication and one addition, taken together. Two FLOPs.
- **Normalization**§2.5 — subtract the mean, divide by the standard deviation, apply a learned scale and shift.
- **Perplexity**§2.2 — cross-entropy re-expressed as an effective branching factor, 2^H.
- **Policy**§14.1 — the model seen as a distribution over responses to a prompt.
- **Prefill**§11.1 — processing a whole prompt at once. Compute-bound.
- **Randomness deficiency**§10.5 — how much shorter a member's description is than a typical member's.
- **Residual connection**§4.2 — adding a block's input to its output, so the Jacobian gains an identity.
- **Ridge point**§11.2 — a machine's peak operation rate divided by its bandwidth. Operations per byte.
- **Softmax**§2.2 — exponentiate and normalize. Sees only differences of logits; saturates.
- **Token**§2.4 — a vocabulary entry. What D counts.
- **Typicality**§10.5 — being a representative member of a set, as against merely belonging to it.

ON THE INDEX

This book has no page index. In the web edition the browser's own search does the job better than any list. The glossary above and the derivation bank of Appendix A are the two ways in.