ADDALY · THE AI CLASSROOM

The Maths You Actually Need

Eight ideas that carry almost all the weight in machine learning.

---

8 modules · 76 lessons · 27 figures · 8 case studies · 62,904 words · about 11 hours

Dr Mustafa Mun
Author and editor

ABOUT THIS BOOK

The Maths You Actually Need, published by Addaly, 2026. Author and editor: **Dr Mustafa Mun**.

The manuscript was drafted with the assistance of a large language model and was edited and published under his name. That is stated plainly here rather than left to be discovered, because a reader is entitled to know how a book they are trusting was made.

This book is the printed form of the course of the same name at addaly.com/learn/maths-you-need. The website is the living version and is corrected first; where the two differ, the website is right.

Free to read, free to download, free to print, and free to teach from. There is no paid edition and there never will be. If you paid for this, somebody has sold you something that was given away.

Every diagram in it is drawn from data by code, not generated as a picture, so the numbers on the page are the numbers in the argument. Answers to every exercise, test and examination question are at the back, with a note on why each wrong option attracts people — those notes are the teaching, not the marking.

Contents

2. Matrices, and what a layer really does

Multiply two matrices by hand and predict the output shape of any chain of layers, explain what the rank of a weight matrix costs and buys, and use singular values to justify replacing a 4096x4096 matrix with two thin ones and to state what that replacement gives up

1. Doing a matrix multiplication yourself, once
2. A matrix is a stack of questions
3. Shapes, batches, and the broadcasting rules
4. Undoing a matrix, and why you rarely should
5. The determinant, and what it says about collapse
6. The best fit, and what "best" was defined to mean
7. Rank, and why a huge matrix can be two small ones
8. The directions a matrix does not turn
9. Singular values, and the honest measure of importance
10. Case study: Rank eight, and the week it nearly cost
11. Module test

3. Change, slopes and the walk downhill

Compute a partial derivative and a chain-rule derivative by hand, trace a gradient backwards through a two-layer network and say where the memory goes, and diagnose from a loss curve and a gradient-norm plot whether the learning rate, the curvature or a vanishing gradient is the cause

1. Slope, before anyone says the word derivative
2. Slopes, and why training is walking downhill
3. Many knobs, one derivative each
4. The chain rule, which is the whole trick
5. Backpropagation, and where the memory goes
6. Convexity, and why nobody promises anything about deep networks
7. Curvature, and what the optimisers are actually doing
8. Reading a training failure from the numbers
9. The one hyperparameter worth your afternoon
10. Case study: Twenty minutes of a shared GPU night
11. Module test

4. Probability you can rely on

Compute a posterior probability from a base rate and a classifier's error rates and explain why a 95%-accurate filter can still be wrong most times it fires, choose a distribution that matches a stated generating process, and predict from the shape of a tail when a mean is the wrong summary

1. Counting, and the size of the spaces involved
2. Probability is a degree of belief
3. Probability once you know something
4. Bayes, done slowly, with real numbers
5. Correlation, dependence, and the gap between them
6. When you cannot solve it, simulate it
7. Distributions, and why the normal one keeps showing up
8. Six distributions, and the process each one describes
9. Expectation and variance in plain terms
10. Covariance, and the shape of a cloud of points
11. Heavy tails, and the average that describes nobody
12. Case study: One thousand seven hundred flags and two radiographers
13. Module test

5. Logarithms, information and the shape of a loss

Compute a cross-entropy and a KL divergence by hand from two small distributions and say which part of a training loss is irreducible, derive the squared-error and cross-entropy losses from a stated noise assumption, compute a softmax and its gradient for a given set of logits, and explain by the overflow limit of float32 why every library subtracts the maximum before exponentiating

1. What a log is, and why the axis got logged
2. Exponentials, decay, and the memory of a moving average
3. Logs, and why losses are logged
4. Surprise, bits, and why prediction is compression
5. Entropy: the average surprise, computed by hand
6. Cross-entropy and KL: the cost of believing the wrong distribution
7. Loss functions are not arbitrary: maximum likelihood
8. Softmax, its gradient, and the shift it cannot see
9. The log-sum-exp trick, and why exp(1000) is not a number
10. Power laws, and the straight line on a log-log plot
11. Case study: A loss of 2.34, and the floor nobody had computed
12. Module test

6. Estimation: from a sample to a number you can trust

Compute a standard error and a confidence interval for an accuracy figure by hand, say from the sample size and base rate whether the textbook interval can be trusted, bound a failure rate from a run with zero failures, derive an L2 or L1 penalty from a stated prior on the weights, and predict how much the best of k noisy candidates will fall back when re-measured

1. An estimator is a recipe, and why the variance divides by n – 1
2. The law of large numbers, and how fast it works
3. Why averages look normal, and when they refuse to
4. Standard error for a mean and a proportion, by hand
5. What a 95 per cent interval promises, and where the textbook one breaks
6. Zero failures in 300 tries, and what that does not prove
7. Updating a rate as evidence arrives: the Beta distribution
8. Regularisation is a prior: L2, L1 and the diamond
9. Regression to the mean, and the winner's curse in model selection
10. Case study: Ninety-four per cent on fifty scripts
11. Module test

7. Counting the cost: complexity and the arithmetic of scale

Count the FLOPs of a forward pass and a training run from a parameter count, compute the memory a model needs at each precision and in training, say from arithmetic intensity whether a workload is bound by compute or by memory bandwidth, recognise a quadratic cost in a pipeline and name the sub-quadratic replacement, and estimate the time or cost of an ML job to within a factor of three before running it

1. Big-O, and the four growth rates you will meet
2. Counting the floating-point operations in a layer and a training run
3. Anything pairwise is quadratic: distances, attention and deduplication
4. Bytes per parameter, and the arithmetic of what fits on a card
5. Why a GPU idles: memory-bound versus compute-bound
6. Hashing, collisions, and the square-root law behind them
7. A graph is a matrix, and its paths are its powers
8. Finding the closest vector among ten million: the arithmetic of approximate search
9. Estimating anything to within a factor of three
10. Case study: Eight million descriptions, and a server nobody needed to buy
11. Module test

8. Numbers inside a machine

Predict from the bit layout of float32, fp16 and bf16 which computation will overflow, underflow or lose its digits, explain why a long sum drifts and how to fix it, quantise a weight to int8 by hand and say what an outlier does to the rest of the tensor, derive the initialisation scale that keeps variance constant through a layer, and diagnose a NaN in training back to the operation that produced it

1. What a float32 actually stores, and where its seven digits run out
2. bf16, fp16, and why training keeps a float32 copy
3. Catastrophic cancellation, and why a sum depends on its order
4. Integers, overflow, and the token id that would not fit
5. Quantisation is rounding with a scale: int8, int4 and the outlier problem
6. The condition number, and how many digits a problem throws away
7. How variance travels through a layer, and the initialisation that follows
8. Random numbers that are not, and what a seed does and does not fix
9. Checking a gradient with finite differences, and the step size that lies
10. NaN, Inf, and the five numbers to print about any tensor
11. Case study: One channel, ninety times the rest
12. Module test

The Maths You Actually Need — final examination

The paper that opens the next course. Answers to everything follow it.

MODULE 1

Vectors, and the space they live in

How a photo, a word, a date and a category all become lists of numbers, and how you measure agreement and distance between them once they are.

> BY THE END OF THIS MODULE YOU CAN
>
> Turn a real piece of data into a vector and defend the choices made, compute a length, a distance, a projection and a cosine by hand, and explain using the 1/√d rule why a similarity threshold that works in a 3-dimensional picture is meaningless in a 768-dimensional embedding space

---

1 · 9 min

Everything a model sees is numbers

> THE ONE THING TO KEEP
>
> A model never sees your data, only the numbers you chose to represent it with, and information discarded in that step cannot be recovered by any amount of model capacity.

The step nobody writes about

Every course starts at "the model learns from data". It skips the step before, which is the one that decides how well anything afterwards can possibly work. A neural network cannot see a photograph, a Tuesday, or the word *shukriya*. It sees a list of numbers, and somebody chose that list.

That choice is called the representation, and it is not neutral. If you throw information away here, no amount of training recovers it.

Four things, and how they become numbers

A photograph. A 224×224 colour image is a grid of 224 rows, 224 columns and 3 colour channels. That is 224 × 224 × 3 = 150,528 numbers, each usually between 0 and 255 before scaling. The famous MNIST digit images are 28 × 28 greyscale, so 784 numbers each. When you read that a vision model "takes 224×224 input", that is the whole content of the claim: it eats a list of 150,528 numbers.

A word. Text is split into tokens, each token has an integer id, and each id indexes a row of a big lookup table. In a model with a 50,000-token vocabulary and 768-dimensional embeddings, that table is 50,000 × 768 = 38.4 million numbers. The word itself has vanished by the time the first layer runs. What remains is row number 8,417, whatever that row contains.

A category. Suppose a column holds `north`, `south`, `east`, `west`. The obvious move is to write 1, 2, 3, 4. This is almost always wrong, because it tells the model that west is four times north and that south sits between north and east. The standard fix is one-hot encoding: four columns, exactly one of them 1. Alternatively you learn an embedding, which is what a model does with words.

The same trap catches postcodes, phone area codes, and customer ids. A pincode of 400001 is a name written with digits, not a quantity. Feeding it as a number invites the model to conclude that 400001 and 400002 are nearly identical, which is true of Mumbai's Fort and Ballard Estate but false of the boundary between two states.

A time. Hours run 0 to 23, and then wrap. Encode the hour as a plain number and the model believes 23:00 and 00:00 are as far apart as two numbers can be within a day. The usual repair is two columns, `sin(2π·hour/24)` and `cos(2π·hour/24)`, which places the hours on a circle so that 23 and 0 sit next to each other. It is a small trick and it routinely improves a demand-forecasting model more than a bigger network does.

Two ways to write down eleven at night

Hour as one number, 0 to 23

- 23:00 and 00:00 are 23 apart, the largest gap the column allows
- Midnight looks as far from 11 p.m. as 11 p.m. is from midday
- The model learns a cliff at midnight that is not in the world
- Nothing warns you: the column is numeric and the fit converges

Hour as a point on a circle

- Two columns: sin of 2 pi h over 24, and cos of the same
- 23:00 and 00:00 land next to each other, as they should
- The distance between two hours is the distance round the clock
- One extra column, and no extra model capacity needed

The same fact, two encodings, and the model can only see the numbers. The circular version routinely improves a demand forecast more than a larger network does, and it costs two lines of pandas.

Scaling, and why it is not cosmetic

Put two columns side by side: annual income in rupees, ranging over hundreds of thousands, and number of children, ranging 0 to 5. Any method based on distance — nearest neighbours, k-means, anything with a gradient — is now driven almost entirely by income, because a difference of ₹50,000 dwarfs a difference of 2 children numerically. The maths has no idea these are different units.

Two standard repairs:

- **Min-max scaling** maps each column onto 0 to 1: `(x - min) / (max - min)`. Simple, and badly damaged by a single outlier.
- **Standardisation** subtracts the mean and divides by the standard deviation, so each column has mean 0 and spread 1. This is the default in most pipelines and the one to reach for first.

Both are computed on the training data and then applied unchanged to test data. Computing them over the whole dataset first is a genuine leak: your test numbers have quietly informed the scaling.

What you have already decided

By the time the first matrix multiplication happens, you have committed to several claims without stating them: that these columns matter and others do not, that this ordering is meaningful and that one is not, that a missing value should be filled with the median rather than flagged as missing. Each is a

modelling decision made in the data-loading code, usually by whoever wrote it in a hurry.

The honest limitation is this: representation errors are invisible in the loss curve. A model fed pincodes as quantities will train perfectly happily, converge, and produce a respectable-looking number. It has learned a real pattern in a fictional space. Nothing in the training process can tell you that the space was fictional, because the training process only ever saw the numbers.

The free path

You do not need anything paid to do any of this. `pandas` and `NumPy` handle loading and scaling; `scikit-learn` has `StandardScaler`, `MinMaxScaler` and `OneHotEncoder`; all three are free and run on a laptop with no GPU. If you have no Python at all, GNU Octave does the array arithmetic and Google Colab gives you a free browser notebook. The maths in this course never needs hardware you have to buy.

BEFORE YOU MOVE ON

A team encodes a six-digit Indian pincode as a single numeric column and trains a delivery-time model. The model trains cleanly and reports a good validation score. What has most likely gone wrong?

A. The pincodes should have been scaled to 0-1 first, and the unscaled magnitude is dominating the gradient.

B. The model has learned real structure in an ordering that does not exist, and neither the loss curve nor the validation score can reveal that.

C. Nothing has gone wrong, because a neural network with enough capacity can learn any mapping from the numeric pincode.

D. The validation score is inflated because pincode leaks the answer from the training set.

2 · 7 min

A vector is just a list

THE ONE THING TO KEEP

A vector is an ordered list of numbers, and in a model its direction carries the meaning, not its length.

Start with a spreadsheet row

A flat in Bengaluru: rent 32000, area 850 square feet, floor 4, walking minutes to the metro 12. Write those numbers in a fixed order and you have a vector:

```
[32000, 850, 4, 12]
```

That is the whole definition. An ordered list of numbers. The order is the only rule — the third slot always means floor, in every flat you describe.

The word scares people because school introduced it as an arrow in physics, with force and velocity and diagrams. In machine learning, forget the arrow for a moment. It is a row.

What models actually store

When a model reads the word "chai", it does not store the letters. It looks up a list of numbers — 384 of them, or 768, or 1536, depending on the model. That list is called an embedding. Nobody sat down and chose those numbers. Training produced them.

Here is the honest part. Individual slots in that list almost never mean anything you could name. Dimension 412 is not "spiciness" or "how Indian the word is". The meaning is smeared across all 768 numbers at once, and any single one is close to unreadable. You may have heard that `king − man +`

`woman` lands near `queen`. That is real for some older word embeddings, and much weaker and more cherry-picked than the story suggests. Enjoy it as a demo. Do not build on it.

What "direction" means

Two vectors point the same way when one is a multiple of the other. `[2, 1]` and `[6, 3]` have different sizes and identical direction.

In an embedding space, the direction is where the meaning lives. The length usually tracks something else entirely — how common the word is, how long the document was, how much text the encoder had to work with.

Make that concrete. A 400-word review of a restaurant in Lagos and a 40-word review saying the same thing produce vectors pointing in nearly the same direction, but the long one is a longer vector. Compare them by angle and they match, correctly. Compare them by raw size and the long one wins, for a reason that has nothing to do with what it says.

This is why almost every retrieval system normalises: divide every number in the vector by the vector's length, so each vector sits on a sphere of radius 1 and only direction is left.

```
import math

v = [3.0, 4.0]
length = math.sqrt(3*3 + 4*4)      # 5.0
unit = [x / length for x in v]     # [0.6, 0.8]
```

That `sqrt(sum of squares)` is Pythagoras, extended to as many slots as you like. It works identically for 768 numbers. Nothing new happens.

Adding and scaling

Add two vectors by adding matching slots. Scale by multiplying every slot. That is it.

This has a real use. Take every sentence embedding in a paragraph and average them, slot by slot, and you get a crude paragraph embedding. It is used in production more than anyone admits. It also throws away word order completely, so "the dog bit the man" and "the man bit the dog" land in the same place. Cheap, useful, and wrong in a specific way you should know about.

Why people say "space"

A list of 768 numbers is a point in 768-dimensional space. You cannot picture that, and you never need to. Everything you will do with it is one of three things: measure a distance, measure an angle, or add.

One warning where 2D intuition breaks. High-dimensional space is mostly empty, and points in it tend to be roughly the same distance from each other. So a rule like "call them similar if the distance is under 0.5" gets tuned on one embedding model and then quietly means nothing when you switch models. Angles survive that switch a little better than distances. Neither survives it completely.

BEFORE YOU MOVE ON

A retrieval system ranks results by the angle between vectors. You multiply every number in one stored embedding by 10 and re-run the same search.

A. It ranks higher for every query now, because a bigger vector scores higher.

B. Its rank is unchanged, because scaling stretches a vector without turning it.

C. It moves to a different region of the space, so it gets different neighbours.

D. It now reads as a stronger, more emphatic version of the same meaning.

3 · 7 min

What a dot product measures

THE ONE THING TO KEEP

A dot product scores how much two lists agree, but it also grows with their length, so normalise before you compare.

Multiply the matching slots, add them up

Two lists of the same length. Multiply slot 1 by slot 1, slot 2 by slot 2, and so on, then add every result into one number.

```
a = [2, 0, 3]
b = [1, 5, −1]
dot = 2∗1 + 0∗5 + 3∗−1   # 2 + 0 − 3 = −1
```

One number out. That number is a score of how much the two lists agree.

Read the sign first. Positive means the two vectors mostly lean the same way. Zero means they are unrelated in the specific sense of being at right angles. Negative means they lean opposite. In the example above, `a` and `b` disagree slightly.

Where the number comes from

The dot product equals `|a| × |b| × cos(θ)` — the length of the first, times the length of the second, times the cosine of the angle between them.

You do not need to work with that formula. You need one consequence of it: a dot product mixes two different things together. It is large when the vectors agree, and it is also large when the vectors are simply long. Length can drown out agreement.

Divide the dot product by both lengths and the lengths cancel out. What remains is `cos(θ)`, a number from -1 to 1 that depends on angle alone. That is cosine similarity, and it is a dot product with the size effect removed.

The same two documents, ranked twice against one query

Ranked by raw dot product

- Product manual, 6,000 words: score 9.6
- Help page, 180 words: score 4.1
- The manual wins because its vector is long, not because it answers
- The top result mentions everything and settles nothing

Ranked by cosine, both lengths divided out

- Help page, 180 words: score 0.71
- Product manual, 6,000 words: score 0.34
- Only the angle survives the division
- The top result is the page that answers the question

Dividing by both lengths turns a dot product into a cosine and removes the size effect. Normalising the stored vectors but forgetting the query is the commonest bug in a first search system, and it never raises an error.

```
def cosine(a, b):
    dot = sum(x*y for x, y in zip(a, b))
    na = sum(x*x for x in a) ** 0.5
    nb = sum(y*y for y in b) ** 0.5
    return dot / (na * nb)
```

Where models use it

Almost every number in a model that answers "how relevant is this to that" is a dot product.

Attention. Each token produces a query vector, each token produces a key vector. The attention score between two tokens is the dot product of one query and one key. A big score means "this token should look at that one". When people say attention is expensive, this is why: for 4000 tokens, that is 16 million dot products, per head, per layer.

Retrieval. Your question becomes a vector, every document is already a vector, and the search is a dot product against each one. Vector databases exist to do that fast.

The final layer. A language model ends with a hidden state, one vector. To score the next token, it dot-products that state against one row per vocabulary

item. 50,000 dot products, 50,000 scores, softmax over them, and you have the next-token distribution.

The honest limits

Cosine similarity measures whether two pieces of text get used in similar contexts. That is not the same as meaning the same thing, and people forget it constantly.

Antonyms score high. "Hot" and "cold" appear in near-identical sentences, so their vectors are close. "I approved the loan" and "I rejected the loan" are near neighbours. If you are building a retrieval system where the difference between approved and rejected matters — and in banking, health, or law it always matters — cosine similarity alone will hurt you, and it will do it quietly, with high confidence scores.

The second limit is the one from the formula. If you skip normalisation and rank by raw dot product, long documents win because long documents have long vectors. Your top result is then a 6000-word page that mentions everything, not the 200-word page that answers the question. This is one of the most common bugs in a first search system, and it never throws an error.

BEFORE YOU MOVE ON

A search over unnormalised embeddings keeps putting very long documents at the top, no matter what the query is. What is going on?

A. Long documents contain more of the query's words, so they genuinely are the better matches.

B. A dot product counts how many terms the query and document share, and longer text shares more terms.

C. A dot product grows with vector length, and long documents produce longer vectors, so size is outvoting angle.

D. The embedding model was trained mostly on long text and is biased, so it needs retraining.

4 · 8 min

How long, and how far

THE ONE THING TO KEEP

A norm turns a whole vector into one number measuring size, and which norm you pick decides whether outliers dominate your answer or barely register.

One number for a whole list

You have a vector. Sometimes you want a single number saying how big it is. That number is called a norm, and there is more than one sensible answer.

Take `v = [3, 4]`.

The L2 norm, also called the Euclidean norm, is the one from school geometry: square each element, add them, take the square root.

```
||v||2 = sqrt(3^2 + 4^2) = sqrt(9 + 16) = sqrt(25) = 5
```

The L1 norm, also called Manhattan distance, adds absolute values:

```
||v||1 = |3| + |4| = 7
```

The L-infinity norm takes the largest absolute value:

```
||v||inf = max(|3|, |4|) = 4
```

Three answers, all correct, all measuring something slightly different. The names come from a family: the Lp norm raises each element to the power p, sums, and takes the p-th root. L2 is p = 2; L1 is p = 1; L-infinity is the limit.

Distance is the norm of a difference

Distance between two vectors is just the norm of what separates them. For `a = [1, 2, 3]` and `b = [4, 6, 3]`:

```
a − b = [−3, −4, 0]
L2 distance = sqrt(9 + 16 + 0) = 5
L1 distance = 3 + 4 + 0 = 7
```

The L1 name comes from walking a street grid: you cannot cut the diagonal, so you go three blocks then four blocks, seven blocks total. L2 is the crow's route, five blocks.

Why the choice changes your answer

The difference is entirely about outliers. Squaring makes a large element enormously more important than a small one. Compare `[10, 0, 0, 0]` with `[5, 5, 5, 5]`:

- L1: both are 10. They look identical.
- L2: `sqrt(100) = 10` against `sqrt(100) = 10`. Also identical, by coincidence of the numbers chosen.

Now compare `[10, 0, 0, 0]` with `[3, 3, 3, 3]`:

- L1: 10 against 12. The spread-out one is bigger.
- L2: 10 against 6. The spiky one is bigger.

That reversal is the whole point. L2 says "one big deviation is worse than four small ones". L1 says "add up the damage, however it is distributed". This is why L2 loss (mean squared error) is pulled around by a single wild data point while L1 loss (mean absolute error) mostly ignores it, and why L1 regularisation drives weights exactly to zero while L2 only shrinks them.

Unit vectors: keeping the direction, dropping the size

Divide a vector by its own L2 norm and you get a vector of length exactly 1, pointing the same way.

```
v = [3, 4],  ||v||2 = 5
v_hat = [3/5, 4/5] = [0.6, 0.8]
check: sqrt(0.36 + 0.64) = sqrt(1) = 1
```

This is called normalising, and it is everywhere in machine learning. Embedding search libraries normalise every vector on insert, so that a dot product *is* a cosine and no division is needed at query time. Layer normalisation inside a transformer does a related thing to each token's activation vector on every layer.

The reason is the one from the dot-product lesson: raw dot products grow with length, so a long vector scores high against everything. Normalising strips length out and leaves only direction, which is where the meaning lives.

What normalising costs you

Length is not always noise. In a bag-of-words representation, the length of a document vector carries the document's size, and sometimes a long document genuinely is a better match. In an embedding from a modern model, vector norms have been shown to correlate loosely with token frequency and with how confident the model is — normalise, and you discard that.

There is also a practical trap. If you normalise your database vectors but forget to normalise the query, your scores are neither dot products nor cosines, and the ranking is quietly wrong in a way that still returns plausible-looking results. This is one of the most common bugs in a hand-built search system, and it never raises an error.

The rule to keep

Ask what you want the number to mean. If a single large deviation should dominate, square it: use L2. If you want a total that treats every unit of error alike, use L1. If you only care about the worst element — a latency budget, a safety margin — use L-infinity. And when you compare directions rather than sizes, normalise first, on both sides.

BEFORE YOU MOVE ON

A search index stores 200,000 product-description embeddings, all normalised to unit length on insert. A developer adds a new query path that sends raw, un-normalised query vectors. What happens?

A. Every score comes back as exactly zero, because the dimensions no longer match.
B. Results are unaffected, because scaling the query by a constant scales every score by the same constant and leaves the ranking unchanged.
C. Nothing detectable, and the index silently returns the reverse ranking.
D. Ranking within one query survives, but any fixed similarity threshold or cross-query score comparison becomes meaningless, and no error is ever raised.

5 · 9 min

Cosine or distance, and why it often does not matter

THE ONE THING TO KEEP

For unit-length vectors, cosine similarity and Euclidean distance are two readings of the same quantity and rank results identically, so the choice only matters when lengths differ.

The identity that settles the argument

Teams argue about whether to use cosine similarity or Euclidean distance in a vector search. Most of the argument disappears once you write down one line of algebra.

For any two vectors a and b:

```
||a − b||^2  =  ||a||^2 + ||b||^2 − 2 (a . b)
```

Expand the squared distance term by term and you get exactly this; it is the law of cosines wearing different clothes.

Now suppose both vectors have been normalised to length 1. Then `||a||^2 = 1` and `||b||^2 = 1`, and the dot product of two unit vectors *is* the cosine of the angle between them. So:

```
||a − b||^2 = 2 − 2 cos(theta)
```

Distance squared is a straight, decreasing function of cosine. Bigger cosine, smaller distance, always, with no exceptions. So if every vector in your index is normalised, sorting by cosine descending and sorting by Euclidean distance ascending give **the same order**. Not a similar order. The same one.

That is worth knowing before you spend an afternoon benchmarking one against the other.

Worked, with numbers

Take `a = [0.6, 0.8]` and `b = [0, 1]`, both unit length.

```
a . b = 0.6*0 + 0.8*1 = 0.8          so cos(theta) = 0.8, theta
= 36.9 degrees
a − b = [0.6, −0.2]
||a − b||^2 = 0.36 + 0.04 = 0.40
check: 2 − 2(0.8) = 2 − 1.6 = 0.40   matches
```

Try a third vector `c = [1, 0]`: `a . c = 0.6`, and `||a − c||² = 0.16 + 0.64 = 0.80 = 2 − 2(0.6)`. Cosine says b beats c; distance says b is closer. They agree, and the identity guarantees they always will.

When they genuinely differ

The identity depends on the lengths being equal to one. Drop that and the two measures pull apart, because `||a||^2` and `||b||^2` re-enter the equation.

Consider two documents about the same subject, one a 200-word note and one a 4,000-word article, represented as raw word-count vectors. Their directions are similar — same vocabulary, similar proportions — so the cosine is high. Their Euclidean distance is large, because one vector's entries are roughly twenty times the other's. Cosine calls them near-duplicates. Euclidean calls them far apart.

Which is right depends on the question. For "are these about the same thing", cosine. For "is this the same size of thing", the raw numbers matter and you should not have thrown length away.

Three practical cases where length carries meaning:

- **Counts and frequencies.** A word appearing 40 times is different evidence from it appearing twice, even at the same proportion.
- **Physical measurements.** Height, weight, price. Two flats with the same shape of features but doubled everything are not the same flat.
- **Model activations you are inspecting.** The magnitude of an activation is often the interesting part; normalising it away deletes what you were looking at.

The similarity numbers people quote

Cosine runs from −1 (opposite) through 0 (perpendicular) to 1 (identical direction). People report thresholds — "we accept matches above 0.8" — and those thresholds do not transfer between models. One embedding model may put unrelated sentence pairs around 0.1; another routinely puts them at 0.6 because its embeddings are squeezed into a narrow cone of the space, a property called anisotropy. A 0.7 in one model can mean less than a 0.4 in another.

The only defensible way to set a threshold is empirical: take a few hundred pairs you have labelled as matching or not matching, compute the scores with your actual model, and read the threshold off the distribution. Copying a number from a blog post about a different model is guessing.

What both measures share, and cannot fix

Neither cosine nor Euclidean distance knows anything about meaning. They measure agreement between two lists of numbers produced by a model, and they inherit everything that model got wrong. If the embedding model was trained mostly on English web text, two Hindi sentences that mean opposite things may sit close together simply because the model represents both weakly. The geometry will report a confident 0.91 and be entirely wrong, because the metric is doing its job perfectly on numbers that were bad before it saw them.

The rule to keep

Normalise on insert and on query, then use dot product — it is the cheapest of the three to compute, it equals cosine once vectors are unit length, and it ranks identically to Euclidean distance. Depart from that only when you have a stated reason why length matters, and write the reason down next to the code.

BEFORE YOU MOVE ON

An engineer benchmarks cosine similarity against Euclidean distance on a vector index whose entries are all normalised to unit length, and reports that cosine gives "noticeably better recall@10". What is the most likely explanation?

A. Cosine handles high-dimensional spaces better, because it ignores magnitude which becomes unreliable as dimensions grow.

B. There is a bug or an inconsistency in the benchmark — with unit-length vectors the two produce identical rankings, so recall@10 cannot differ.

C. Euclidean distance is being computed without the square root, which changes the ordering.

D. Recall@10 measures a different quantity from similarity, so the two metrics are not comparable in the first place.

6 · 9 min

Projection, and the art of removing a direction

THE ONE THING TO KEEP

Projection splits a vector into the part that lies along a chosen direction and the part that does not, which is how you both read a feature out and attempt to strip one away.

Splitting a vector in two

Take a vector `a` and a direction `b`. Every vector `a` can be written as the sum of two pieces: one lying along `b`, and one perpendicular to it. The first piece is the **projection** of a onto b, and it is one formula:

```
proj_b(a) = ( (a . b) / (b . b) ) * b
```

If `b` is already a unit vector, `b . b = 1` and this simplifies to `(a . b) * b`. The scalar `a . b` is *how much of b there is in a*, and multiplying by b turns that scalar back into a vector.

Worked, with `a = [3, 4]` and `b = [1, 0]`:

```
a . b = 3
b . b = 1
proj_b(a) = 3 * [1, 0] = [3, 0]
the leftover: a − proj_b(a) = [0, 4]
```

The leftover, `[0, 4]`, is perpendicular to `b`: their dot product is `0*1 + 4*0 = 0`. That is the definition of **orthogonal** — a dot product of exactly zero, meaning the two directions carry no information about each other under this measure.

Where projection is already happening

Every row of a weight matrix is a direction, and multiplying by that row computes the scalar part of the projection: how much of that direction is present in the input. A linear layer with 768 inputs and 3,072 outputs is asking 3,072 projection questions at once. That is all it is doing.

In attention, the query and key matrices project each token's representation into a smaller space — often 64 dimensions per head — before the dot products are taken. The reason is that different heads can then pick different directions to compare along, so one head compares subject to verb and another compares a pronoun to its antecedent, without competing for the same numbers.

Removing a direction, and why it half works

Here is the operation that gets used for something more interesting. If you want the part of `a` that has nothing to do with `b`, subtract the projection:

```
a_perp = a − ( (a . b) / (b . b) ) ∗ b
```

This is the arithmetic behind a family of attempts to remove an unwanted attribute from a representation. Find a direction in embedding space that seems to encode, say, a gender association — often by taking the average difference between paired words — then project every embedding onto the space perpendicular to it. Afterwards, the dot product of every vector with that direction is zero. The attribute appears to be gone.

It is a genuinely useful technique and its limits are well documented. Later work found that after such a projection, the information is frequently still recoverable: a classifier trained on the "debiased" vectors can still predict the removed attribute well above chance, because the attribute was never confined to one direction. It was spread across many correlated directions, and removing one of them leaves the rest.

This is worth holding onto as a general lesson about linear methods. Subtracting one direction removes exactly one direction. If the thing you dis-

like lives in a subspace of forty directions, or is encoded non-linearly, the arithmetic will report success while the property survives. The maths did what you asked; the ask was too small.

Orthogonality as a design goal

Orthogonal directions are convenient because they do not interfere. Change the amount of one and the others are untouched. Several pieces of architecture lean on this:

- **Orthogonal initialisation** starts a weight matrix with mutually perpendicular rows, so the initial signal is neither amplified nor collapsed as it passes through.
- **Multi-head attention** works better when heads attend to different things, which in geometric terms means their projection subspaces are close to orthogonal. Nothing enforces it; training tends to arrive there because redundant heads gain nothing.
- **The residual stream** in a transformer is often described as a shared workspace in which different features occupy near-orthogonal directions, so that many can be written and read without erasing one another.

That last description is a useful model, not an established fact. Current interpretability research argues that models pack far more features than they have dimensions, using directions that are *almost* orthogonal rather than exactly so — a phenomenon called superposition. It explains a great deal and it is still an active research question rather than settled ground. Treat it as the best current story.

The rule to keep

The projection formula answers "how much of this direction is in that vector", and its complement answers "what is left once I take that direction out". Both are one dot product and one subtraction. What neither can do is guarantee that a concept you care about was ever a single direction in the first place.

BEFORE YOU MOVE ON

A team finds a "formality" direction in an embedding space and projects every stored vector onto the subspace perpendicular to it, so that all remaining vectors have exactly zero dot product with that direction. A classifier is then trained on the projected vectors and still predicts formality with 78% accuracy. What does this show?

A. The projection was computed incorrectly, since a correct projection leaves zero information about the removed direction.

B. The classifier is overfitting, and would fall to chance on a held-out set.

C. The concept was never confined to a single direction, so removing one direction left the correlated directions that also carry it.

D. Formality is a non-linear property, so no linear operation can affect it at all.

7 · 8 min

The same vector, written two different ways

THE ONE THING TO KEEP

A vector's numbers depend on the axes you chose to describe it with, which is why an individual dimension of an embedding usually means nothing on its own.

Numbers are a description, not the thing

Point at a spot on a table and say where it is. You might say "30 centimetres from the left edge, 20 from the front". Somebody standing on the other side of the table would describe the same spot with different numbers. The spot did not move. The description changed, because the axes changed.

A vector works the same way. When we write `[3, 4]` we mean 3 lots of one reference direction plus 4 lots of another. Those reference directions are called a **basis**. The default is the standard basis: `e1 = [1, 0]` and `e2 = [0,`

`1]`, so `[3, 4] = 3·e1 + 4·e2`. That is so automatic that most people never notice a choice was made.

Rewriting in another basis

Choose a different pair of directions, say `u1 = [1, 1]/√2` and `u2 = [−1, 1]/√2` — the same axes rotated 45 degrees, still unit length and still perpendicular. Since they are orthonormal, the new coordinates are just dot products:

```
a = [3, 4]
a . u1 = (3 + 4)/sqrt(2) = 7/1.414 = 4.95
a . u2 = (−3 + 4)/sqrt(2) = 1/1.414 = 0.707
```

So the same arrow is `[3, 4]` in one basis and `[4.95, 0.707]` in the other. Check that nothing physical changed: its length in the new basis is `sqrt(4.95² + 0.707²) = sqrt(24.5 + 0.5) = 5`, exactly the length it had before. Lengths, angles and dot products are all preserved by an orthonormal change of basis. Only the numbers move.

Why this matters for embeddings

People open an embedding, look at dimension 42, and try to work out what it means. Usually it means nothing.

Most training objectives are invariant to rotation of the representation space. If you rotate every embedding by the same orthogonal matrix and rotate the next layer's weights back by the inverse, every dot product, every distance and every output is identical, so the loss is identical. Training has no reason to prefer one rotation over another, so the axes it lands on are arbitrary. A single coordinate is a projection onto an axis nobody chose.

This is why the honest way to inspect an embedding space is through relationships — distances, neighbourhoods, directions between pairs — rather than through individual coordinates. The relationships survive a change of basis; the coordinates do not.

The important exception

There are situations where individual dimensions do mean something, and it is worth knowing which.

After PCA. Principal component analysis chooses a specific basis, ordered by how much variance each direction explains. Component 1 is not arbitrary: it is defined as the direction of greatest spread. Reading it is legitimate.

Where a non-linearity breaks the symmetry. A ReLU acts coordinate by coordinate — it zeroes negative entries in the basis it is given. That destroys rotation invariance for the layer it follows, which is why some interpretability work does find meaningful individual neurons in the hidden layers of a network with element-wise activations, while pure embedding tables show much less of it.

When a sparse autoencoder has been fitted. A recent line of work trains a wide, sparse layer whose job is to find a basis in which features *are* individually meaningful. Early results look encouraging and the approach is under active debate — how many features are found, whether they are the model's features or the autoencoder's, and how to evaluate any of it are all open. It is a promising research direction, not a solved tool.

The practical consequences

Three things follow directly.

1. **Do not compare dimension k of one model with dimension k of another.** Two models trained on the same data with different seeds will have unrelated bases. Their vectors are not interchangeable in any way, which is why you must re-embed your whole corpus when you change embedding model, rather than mixing old and new vectors in one index.

1. **Do not average embeddings from different models.** The arithmetic runs; the result is meaningless, because you are adding coordinates measured against different axes.

1. **Concatenating is different from averaging, and is sometimes fine.** Sticking a 768-dimension vector from one model next to a 384-di-

mension vector from another gives a 1,152-dimension vector whose dot product is the sum of two separate dot products. That is a defensible hybrid, provided you scale each half deliberately rather than by accident.

The rule to keep

Coordinates are relative to a basis. Distances, angles and neighbourhoods are not. Build everything you rely on out of the second kind.

BEFORE YOU MOVE ON

A team wants to save storage, so they keep only dimensions 0-383 of their 768-dimension embeddings, reasoning that half the information is better than none. Why is this worse than running PCA down to 384 dimensions?

A. Truncation loses exactly half the information while PCA loses none, since PCA is lossless.

B. The first 384 dimensions of an arbitrary basis carry no particular share of the variance, whereas PCA's first 384 components are defined as the directions carrying the most.

C. Truncation breaks the vectors' unit length, and PCA preserves it.

D. PCA is trained on the data and therefore adapts to it, while truncation is a fixed operation that cannot adapt.

8 · 10 min

Why your 3-D intuition is wrong in 768 dimensions

THE ONE THING TO KEEP

In high dimensions random vectors are nearly perpendicular and all distances converge, so any similarity threshold has to be measured for your actual model rather than reasoned about from a picture.

The picture you are drawing is misleading you

Every explanation of embeddings shows a 2-D or 3-D plot: dots, clusters, arrows. It is the only way to draw the thing. It is also wrong in ways that will cost you, because spaces with hundreds of dimensions behave nothing like the space you can see.

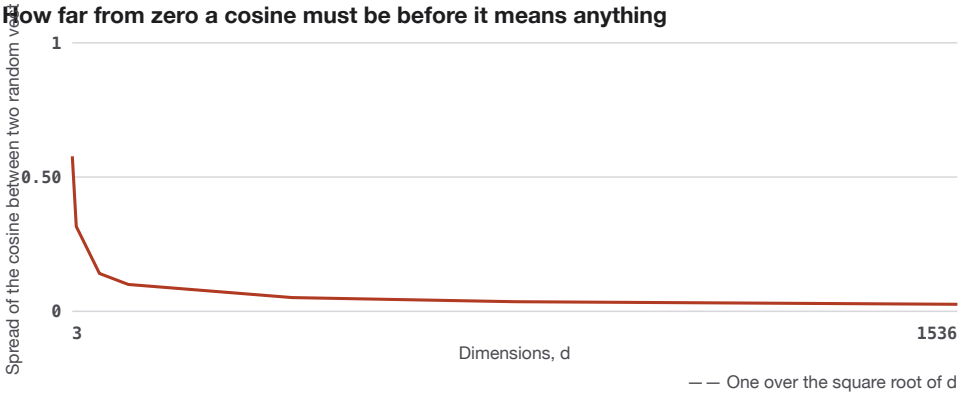
Here is the single most useful fact. Take two random vectors in `d` dimensions, each with entries drawn independently from a normal distribution, and compute the cosine of the angle between them. The expected value is 0 — they are perpendicular on average. That much is unsurprising. What surprises people is the spread: the standard deviation of that cosine is approximately `1/sqrt(d)`.

Run the numbers:

- In 3 dimensions: `1/sqrt(3) = 0.577`. Random vectors routinely land at cosine 0.5 or −0.5. Everything looks related to everything.
- In 100 dimensions: `1/sqrt(100) = 0.10`. A cosine of 0.3 is now three standard deviations out.
- In 768 dimensions: `1/sqrt(768) = 0.036`. A cosine of 0.11 is already three standard deviations from random.

So in a 768-dimensional space, two vectors that a 3-D intuition would call "barely related at 0.15" are in fact a four-sigma event. Almost everything is al-

most perpendicular to almost everything else. There is an enormous amount of room.



In three dimensions a cosine of 0.5 is ordinary. In 768 dimensions it is fourteen standard deviations out. Any threshold carried over from a picture, or from a blog post about a different model, is a number about the wrong space.

You can verify this in ten lines of free NumPy:

```
import numpy as np
for d in (3, 100, 768):
    a = np.random.randn(20000, d)
    a /= np.linalg.norm(a, axis=1, keepdims=True)
    b = np.random.randn(20000, d)
    b /= np.linalg.norm(b, axis=1, keepdims=True)
    cos = (a * b).sum(axis=1)
    print(d, round(cos.std(), 3), round(1/np.sqrt(d), 3))
```

The two printed numbers agree to two decimal places. This is not a rule of thumb; it is what the space is.

Distances stop discriminating

The second strange fact is called distance concentration. Sample many points at random in `d` dimensions and look at the nearest and the farthest from a query point. As `d` grows, the ratio

```
(farthest distance − nearest distance) / nearest distance
```

tends towards zero. In high enough dimensions everything is roughly the same distance away, and "nearest neighbour" stops being a meaningful description. Formally this holds for independently distributed coordinates, and it is why brute-force nearest-neighbour search on genuinely random high-dimensional data is close to useless.

A third, related fact: in high dimensions, nearly all of a ball's volume sits in a thin shell just under its surface. The fraction of the volume of a d-dimensional unit ball lying within radius 0.9 is `0.9^d`. For d = 3 that is 0.73; for d = 100 it is 0.000027. The interior is empty. Your mental image of a cluster as a filled blob is wrong; it is a shell.

So why does vector search work at all?

If distances concentrate, why does embedding search return sensible results every day?

Because real embeddings are not random. A trained model maps its inputs onto a much lower-dimensional structure inside the big space — a curved surface, usually called a manifold. The stated dimension is 768; the *effective* dimension, the number of directions along which the data actually varies, is typically far lower, often in the tens. Distance concentration is a statement about points spread through the whole space, and your points are not spread through the whole space.

This is exactly why approximate-nearest-neighbour indexes such as HNSW or IVF work. They exploit the fact that the data has local structure. On genuinely uniform random data they degrade towards brute force, which is one of the reasons a synthetic benchmark with random vectors tells you very little about how your index will behave on real ones.

What this changes in practice

Thresholds must be measured, never assumed. "Similarity above 0.8 means a match" is a claim about one model's output distribution. Compute the distribution of scores for your model on a few hundred labelled pairs and read the cut-off from it.

Beware anisotropy. Many trained embedding models push all their vectors into a narrow cone, so the average cosine between two unrelated sentences might be 0.6 rather than 0. The variance around that average is what carries the signal. If your scores all cluster between 0.55 and 0.75, that is normal, and subtracting the mean before comparing often sharpens results noticeably.

More dimensions are not free. Doubling dimensions doubles storage and query cost, and beyond the data's effective dimension it adds noise rather than signal. Measured retrieval quality often plateaus well before the model's full width, which is the reasoning behind models that let you truncate the vector to 256 or 512 dimensions with a small, measurable quality cost.

The rule to keep

`1/sqrt(d)` is the number to remember. It tells you how far from zero a cosine has to be before it means anything, and it explains why your intuition, trained in three dimensions, will always overestimate how related two things are.

BEFORE YOU MOVE ON

A developer builds a synthetic benchmark by generating one million random 768-dimensional vectors, indexes them with HNSW, and finds recall@10 is poor and the index barely faster than brute force. What has the benchmark actually demonstrated?

A. That HNSW is unsuitable for 768-dimensional data and a lower dimension should be used.

B. That the index is misconfigured, since a correctly built HNSW graph achieves high recall on any data.

C. That uniformly random vectors have no local structure for the graph to exploit and their distances concentrate, so the benchmark measures the worst case rather than the realistic one.

D. That one million vectors is too few to populate a 768-dimensional space, and more data would fix it.

9 · 9 min

Sparse and dense, and the memory each costs

THE ONE THING TO KEEP

Sparse representations record which items are present and dense ones record what they are like, and the arithmetic of storing each explains why serious search systems use both.

Two ways to write down a document

Suppose your vocabulary has 50,000 words and you want to represent the sentence "the train was late".

The sparse way. Make a vector with 50,000 slots, put a count in the slot for each word present, and zeroes everywhere else. Four non-zero entries out of 50,000. Stored naively that is 50,000 numbers; stored sensibly it is four `(index, value)` pairs, because the zeroes carry no information and need not exist.

The dense way. Run the sentence through an embedding model and get 768 numbers, almost none of them zero, none of them individually interpretable.

Neither is a better idea in general. They record different things. The sparse vector says *exactly which words are here*. The dense vector says *what this is roughly about*.

One-hot, the simplest sparse form

The atom of sparse representation is one-hot encoding: a vector with a single 1 and zeroes elsewhere. Word 8,417 in a 50,000-word vocabulary becomes a vector with 1 in position 8,417.

Its properties are worth stating because they explain what came next. Every one-hot vector has length 1. The dot product between any two different one-hot vectors is 0, so every word is exactly as unrelated to every other word —

`cat` and `dog` are as distant as `cat` and `parliament`. And the space needed grows with the vocabulary, one dimension per word.

The embedding table fixes all three at once. Multiply a one-hot vector by a 50,000 × 768 matrix and you select row 8,417 of that matrix. That is the entire mechanism: **an embedding lookup is a one-hot multiplication that somebody optimised into an array index.** The rows are learned, so related words end up with related rows, and the representation is 768 numbers instead of 50,000.

The memory arithmetic

Do the sums, because they decide architecture.

One million documents, dense 768-dimensional embeddings, 4-byte floats:

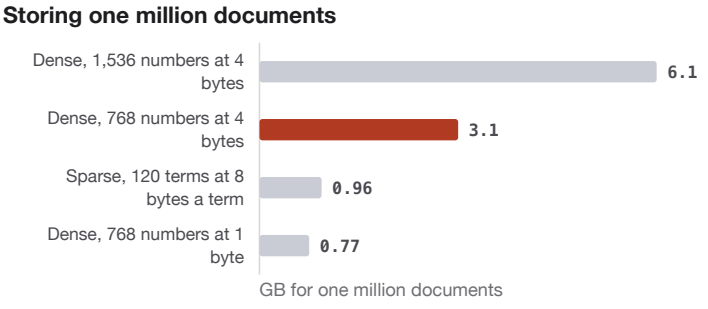
```
1,000,000 x 768 x 4 bytes = 3,072,000,000 bytes = 3.07 GB
```

That fits in the RAM of an ordinary laptop, which is why in-memory vector search is practical at this scale. At ten million documents it is 30.7 GB and you are buying a server or quantising. Store the same vectors as 1-byte integers instead of 4-byte floats and it falls to 768 MB, a point returned to in the module on quantisation.

The same million documents, sparse, averaging 120 distinct terms each, stored as an index plus a value at 8 bytes per pair:

```
1,000,000 x 120 x 8 bytes = 960,000,000 bytes = 0.96 GB
```

Less than the dense version, and it grows with document length rather than with vocabulary size. An inverted index — the structure behind every keyword search engine since the 1970s — makes this smaller still and queries very fast, because you only ever touch the postings lists for the words in the query.



Sparse storage grows with document length, dense storage with the width of the model. Multiply every figure by ten for ten million documents, and the first two lines stop fitting on a laptop.

TF-IDF: making sparse counts behave

Raw counts are a poor sparse representation because common words dominate. The classical repair is TF-IDF: multiply each term's count in a document by the inverse document frequency,

```
idf(t) = log( N / df(t) )
```

where `N` is the number of documents and `df(t)` how many contain the term. A word in every document gets `log(1) = 0` and vanishes. A word in one document in a thousand gets `log(1000) = 6.9` and dominates. BM25, the default in Lucene, Elasticsearch and SQLite's FTS5, is a refined version with saturation on term frequency and a correction for document length.

This is worth knowing because BM25 is a strong baseline that costs nothing to run and is genuinely hard to beat on queries containing rare exact terms — a part number, a surname, an error code, `NullPointerException`. A dense model has to have learned that token to represent it; a sparse index simply matches it.

Why real systems use both

The failure modes are complementary, which is the whole argument for hybrid search.

- Sparse fails on synonyms and paraphrase. A query for "cheap flights" does not match a document saying "budget airfare" if no word overlaps.

- Dense fails on rare exact strings. Ask for order `AX-99182` and the embedding has no idea; the token was probably never seen.

Hybrid search runs both and combines the rankings, typically with reciprocal rank fusion, which merges by rank position rather than by score and so avoids the problem that the two systems' scores are on incompatible scales. It reliably beats either alone on mixed query workloads.

The free path

None of this needs a paid service. `scikit-learn` gives you `TfidfVectorizer` in one line. SQLite ships FTS5 with BM25 built in and runs on a phone. `rank_bm25` is a small pure-Python implementation. For dense vectors, `sentence-transformers` runs a compact model on a CPU, and FAISS or `hnswlib` index them locally. A hybrid search system over a few hundred thousand documents runs on a laptop with no account anywhere.

The rule to keep

Sparse records identity, dense records similarity, and the memory arithmetic tells you which is affordable at your scale. If your queries contain names, codes or numbers, you need the sparse half no matter how good the embedding model gets.

BEFORE YOU MOVE ON

A support team replaces their keyword search with dense embedding search. Overall satisfaction improves, but agents complain that searching for a specific error code such as `ERR_SSL_PROTOCOL_ERROR` now returns nothing useful. What is the mechanism?

A. The embedding model has a token limit and the long code is being truncated before it is embedded.

B. Dense vectors are normalised, so the rare code contributes very little to the direction of the query vector and cannot dominate the match.

C. The code is a rare token the model has little or no learned representation for, so its embedding carries almost no distinguishing signal, whereas an inverted index matches it exactly.

D. Cosine similarity cannot represent exact matches, only approximate ones, so exact-string queries always fail.

CASE STUDY · MODULE 1

Six hundred circulars and a scheme code nobody could find

A district co-operative bank in Kolhapur, forty branches, building a search over its own loan circulars for the officers at the counter.

The bank had six hundred circulars going back eleven years. Every one of them told a loan officer what a scheme allowed, what it required and when it changed. They lived in a shared folder, named by date, and finding the right one took a phone call to head office. In the sowing season that call took two days.

Two people in the IT cell built a search. They split the circulars into about seven thousand passages, embedded each with a free 384-dimensional model that runs on a laptop, and stored the vectors. The arithmetic said the scale was never going to be the problem: seven thousand passages at 384 numbers of four bytes each is eleven megabytes, and a brute-force comparison against all of them takes under a millisecond. Nothing here needed a vector database, a server, or a subscription.

The first version worked well on questions and badly on facts. "What is the margin requirement for a dairy loan?" returned the right passage. "KCC-2019/07" returned nothing useful at all. The embedding model had never seen that string as a token, so it had no row to look up and no direction to point in; the circular that carried the code was ranked forty-first.

They also found their threshold was meaningless. They had copied a rule from a tutorial: accept a match above cosine 0.8. On their model almost nothing scored above 0.8, and the search returned an empty list for most queries. So they labelled two hundred query-passage pairs by hand, matching and not matching, and plotted the scores. Unrelated pairs averaged 0.61. Matching pairs averaged 0.78. The whole signal lived in a band about fifteen points wide, sitting well above zero, because the model pushed all its vectors into a narrow cone. The right cut-off for their model and their documents was 0.71, and it could not have been guessed.

That left the real decision. The dense search alone would ship in a week. It would answer questions about schemes and it would fail, quietly and confidently, on scheme codes, account formats, circular numbers and the surnames of the officers who signed them. A hybrid search, running a BM25 index alongside the embeddings and merging the two rankings by position, would take about three more weeks: building the index was an afternoon, but the merge, the evaluation set and the retraining of forty branch managers were not.

The cost of waiting was concrete. The sowing season started in six weeks and the counter queues would double. Every week without search was another week of two-day phone calls, and the branch managers had been promised something before the season, not after it.

The cost of shipping the dense version alone was harder to see and worse. A loan officer who searches a scheme code, gets a confident-looking passage from a superseded 2017 circular, and quotes it at the counter has been actively misled by the tool. Nothing in the interface would say the code had not matched; the score would look ordinary. The IT cell had seen that failure in testing and could not think of a way to warn a user about it, because a dense model does not know that it has never seen a token.

They put both options to the general manager with the numbers attached: eleven megabytes, one millisecond, a measured threshold of 0.71, and a list of twelve real queries from the branches on which the dense search returned the wrong circular.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The general manager took the three-week delay. The team shipped hybrid search two weeks before the season, running BM25 and the embeddings to-

gether and merging by rank position rather than by score, because the two scores were on scales that could not be compared. The threshold went in as 0.71, with a note in the code saying it had been measured on two hundred labelled pairs against that specific model and had to be re-measured if the model ever changed. On the twelve failing queries, hybrid search returned the right circular for eleven; the twelfth was a code that had been typed wrongly in the original document. The team also added a line to the results page saying which half of the system had found each result, which turned out to be the feature the branch managers mentioned most.

The dense search could not find "KCC-2019/07". Why does adding a keyword index fix that when a better embedding model would not?

A dense model represents a string only if its tokeniser produced tokens it has learned rows for. A rare code is split into fragments the model has almost no signal about, so its direction in the space is close to noise, and no amount of model quality changes that for a string it has never usefully seen. A sparse index does not represent the code at all; it matches it. The two systems fail in complementary ways, which is the whole argument for running both.

Why was a threshold of 0.8, copied from a tutorial, the wrong number here, and what would make 0.71 wrong too?

Cosine scores are a property of one model's output distribution, not of meaning. This model was anisotropic: it packed its vectors into a narrow cone, so even unrelated passages averaged 0.61 and the useful signal sat in a band above that. A threshold has to be read off a labelled sample from the actual model. The 0.71 would become wrong the moment they changed embedding model, or re-chunked the documents, since both change the distribution the number was read from.

The team merged the two rankings by position rather than by score. What goes wrong if you add a BM25 score to a cosine?

The two numbers are on incompatible scales. A cosine is bounded between minus one and one and, on this model, clustered around 0.6; a BM25 score is unbounded and depends on corpus statistics and document length. Adding or averaging them lets whichever happens to have the larger numbers dominate the ranking for reasons unrelated to relevance. Merging by rank position discards the magnitudes and keeps only the ordering each system is entitled to assert.

MODULE 1 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A first search system ranks documents by the raw dot product between the query vector and each document vector, with no normalisation anywhere. Long pages keep coming top. What is the mechanism?

A. A dot product is both lengths times the cosine of the angle, so a long vector scores high against everything

B. Long documents contain more of the query's words, so they genuinely are more relevant

C. The embedding model produces higher-quality vectors for long documents than for short ones

D. Dot products are undefined between vectors of different lengths, so the library pads the shorter one

2 A team moves from a 384-dimensional embedding model to a 1,536-dimensional one and keeps its cosine threshold of 0.35. What does the one-over-root-d rule say about that threshold?

A. Nothing changes, because a cosine is already normalised and is comparable across models

B. It now demands a far more unusual match: random pairs spread about 0.026, not 0.051

C. It is now far looser, because more dimensions give two vectors more chances to agree

D. It should be doubled, because each vector now holds four times as many numbers

3 Two error vectors over four items: A is (10, 0, 0, 0) and B is (3, 3, 3, 3). Which norm calls A the larger, and why does the answer matter for a loss function?

A. L1 calls A larger, because it adds absolute values without squaring them

B. Both call A larger, since 10 is the biggest single number in either vector

C. L2 calls A larger, because squaring makes one big deviation count for more than four small ones

D. Neither: the two vectors have the same size under every Lp norm

4 A team finds a direction in embedding space that encodes an unwanted attribute and projects every vector onto the space perpendicular to it. A classifier trained on the result still predicts the attribute well above chance. What went wrong?

A. The projection formula was applied with the wrong sign, so the direction was doubled rather than removed

B. The attribute was never confined to one direction, and subtracting one direction removes exactly one

C. Projection works only on unit vectors, and theirs had not been normalised first

D. The classifier is overfitting, and would fall to chance on a larger test set

5 Why can you not average a 768-number embedding from one model with a 768-number embedding from another model trained with a different seed?

A. The two models were trained on different data, so the averaged vector would be biased towards the larger corpus

B. Averaging is defined only for vectors of the same length, and these two differ in scale

C. Most training objectives are unchanged by a rotation, so each model lands on arbitrary axes and slot 42 means different things

D. They would need to be concatenated first and the result halved

6 A support search must handle both the sentence "my order has not arrived" and the order code AX-99182. Why does a dense embedding model handle the first and fail the second?

A. Dense models cap the number of characters they can represent, and the code exceeds that cap

B. The code contains digits, and embedding models discard numeric tokens during training

C. The code is too short, and an embedding needs a full sentence to produce a usable vector

D. A dense vector records what text is about, and a rare code splits into fragments the model has almost no signal for

MODULE 2

Matrices, and what a layer really does

Multiplying by hand, predicting shapes, and the three factorisations — rank, eigenvectors, singular values — that explain why a huge weight matrix can often be replaced by two small ones.

> BY THE END OF THIS MODULE YOU CAN
>
> Multiply two matrices by hand and predict the output shape of any chain of layers, explain what the rank of a weight matrix costs and buys, and use singular values to justify replacing a 4096x4096 matrix with two thin ones and to state what that replacement gives up

---

10 · 8 min

Doing a matrix multiplication yourself, once

> THE ONE THING TO KEEP
>
> Every entry of a matrix product is one dot product between a row of the left matrix and a column of the right, which is why the inner dimensions must match and why order cannot be swapped.

Do it once and you never fear it again

Matrix multiplication looks like a rule to memorise. It is one sentence: **entry (i, j) of the answer is the dot product of row i of the left matrix with column j of the right matrix.** Everything else follows from that, including the shape rule and the fact that order matters.

Here is a full worked example. Take

```
A = [ 1  2  3 ]        B = [ 7   8 ]
    [ 4  5  6 ]            [ 9  10 ]
                           [ 11 12 ]
```

A is 2×3, B is 3×2. Work out each of the four answer entries.

```
(1,1): row 1 of A . column 1 of B = 1*7 + 2*9 + 3*11 = 7 + 18 +
33  = 58
(1,2): row 1 of A . column 2 of B = 1*8 + 2*10 + 3*12 = 8 + 20
+ 36 = 64
(2,1): row 2 of A . column 1 of B = 4*7 + 5*9 + 6*11 = 28 + 45
+ 66 = 139
(2,2): row 2 of A . column 2 of B = 4*8 + 5*10 + 6*12 = 32 + 50
+ 72 = 154
```

So

```
AB = [ 58   64 ]
     [ 139 154 ]
```

Do that once, slowly, with a pen. It takes four minutes and it converts a rule into a thing you understand.

Every entry of AB is one dot product

	Col 1 of B: 7 9 11	Col 2 of B: 8 10 12
Row 1 of A: 1 2 3	**58** 7 + 18 + 33	**64** 8 + 20 + 36
Row 2 of A: 4 5 6	**139** 28 + 45 + 66	**154** 32 + 50 + 72

Four entries, four dot products of length three. The inner dimension has to be 3 on both sides because a dot product needs two lists of equal length, and that is the whole explanation for the shape error you will meet most often in your first month.

The shape rule, and why it is what it is

An (m × n) matrix times an (n × p) matrix gives an (m × p) matrix. Write the shapes next to each other:

```
(2 x 3) (3 x 2)  ->  (2 x 2)
     ^---^  the inner numbers must match; they vanish
   ^---------^     the outer numbers survive
```

The inner numbers must match because each answer entry is a dot product, and a dot product needs two lists of equal length. Row i of A has n entries; column j of B has n entries. If they differ, there is nothing to compute. That is the whole explanation for the error message you will see most often in your first month.

Order matters, and it is not a technicality

`AB` and `BA` are different operations, and frequently only one of them is even legal. With the A and B above, `BA` is (3×2)(2×3) = 3×3 — a different size of answer entirely. Even when both are square and both legal, the results usually differ:

```
[1 1] [1 0]   =  [1 0]        [1 0] [1 1]   =  [1 1]
[0 1] [1 1]      [1 1]        [1 1] [0 1]      [1 2]
```

Two legal products, two different answers. This matters in practice because it is why a stack of layers is a sequence, not a set: `W3 W2 W1 x` applies W1 first. Swapping two layers changes the function.

Reading it as columns instead of dots

There is a second reading, and it is the one that makes deep learning make sense. Instead of "dot products", think of a matrix times a vector as **a weighted sum of the matrix's columns**:

```
[ 1  2 ] [ 3 ]  =  3 * [1]  +  4 * [2]  =  [ 3 + 8  ]  =  [ 11
]
[ 4  5 ] [ 4 ]         [4]         [5]     [ 12 + 20]     [ 32
]
```

Every output of a linear layer is a mixture of the columns of the weight matrix, with the input supplying the mixing amounts. The set of everything reachable this way is called the column space, and it is exactly what the layer can produce. Anything outside it is unreachable at any input, which is a limitation you will meet again under the word *rank*.

Cost, in operations you can count

Multiplying an (m × n) by an (n × p) matrix requires `m * n * p` multiply-add operations, because there are `m * p` output entries and each costs `n` multiply-adds. Put numbers in it: a single 4096 × 4096 weight matrix applied to one 4096-vector costs

```
4096 * 4096 * 1 = 16.7 million multiply-adds
```

which is why people count a matrix multiplication as roughly `2 * m * n * p` floating-point operations — one multiply and one add each. A transformer layer does several of these per token, and a model has dozens of layers. That arithmetic, and nothing more mysterious, is where the electricity goes.

The two habits worth forming

Write the shapes. Before writing any model code, write the shape of every tensor as a comment: `(batch, seq, d_model)`. Most bugs in a first model are shape bugs, and most shape bugs are visible in the comments before the code runs.

Verify small. When you are unsure what an operation does, run it on a 2×3 and a 3×2 you can check by hand. In free NumPy:

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[7, 8], [9, 10], [11, 12]])
print(A @ B)          # the 58 / 64 / 139 / 154 above
print((A @ B).shape)  # (2, 2)
```

Three lines, no GPU, no account. Any claim in this module can be checked the same way, and checking it yourself is the difference between knowing the rule and believing it.

BEFORE YOU MOVE ON

A model has a weight matrix of shape (768, 3072) and an input batch of shape (32, 128, 768) meaning batch, sequence position, features. What is the output shape, and which dimension does the matrix consume?

A. (32, 128, 3072); the matrix consumes the 768 feature dimension, since that is the inner dimension that must match.

B. (32, 768, 3072); the matrix consumes the 128 sequence dimension, since matrix multiplication acts on the second-to-last axis.

C. (32, 128, 768); the matrix maps features to features and preserves the shape.

D. (3072, 128, 32); the output is the transpose of the input with the new feature count first.

11 · 8 min

A matrix is a stack of questions

THE ONE THING TO KEEP

A matrix is a stack of questions, each row a dot product; without a bend between them, stacked layers collapse into one.

Stop looking at the grid

A matrix is drawn as a grid of numbers, and that drawing is why it feels like homework. The grid is storage. It is not what the thing is.

A matrix is a function. You feed it a vector, it gives you back a vector. That is the useful definition, and it is the one that makes neural networks make sense.

Each row asks one question

Here is the mechanism, and it is entirely built from the last lesson. To apply a matrix to a vector, take each **row** of the matrix and dot-product it with your vector. Each row produces one number. Stack those numbers and that is your output vector.

So a matrix with 4 rows turns any input into 4 numbers. Each row is a question being asked of the input, and its answer is one number.

```
# rows ask: "how much X?", "how much Y?", "X and Y together?"
M = [[1, 0],
     [0, 1],
     [1, 1]]
v = [3, 5]

out = [sum(r*x for r, x in zip(row, v)) for row in M]
# [3, 5, 8]
```

That reframes every linear layer in every model you will ever use. `nn.Linear(768, 3072)` is a matrix with 3072 rows and 768 columns. It takes a 768-number description and asks 3072 questions of it, each question learned during training. The output is 3072 answers.

The shape now reads as meaning, not bookkeeping. Rows out, columns in. If the columns do not match the length of your input, the questions are asking about slots your vector does not have, which is exactly what a shape mismatch error is telling you.

What questions can be learned

The answer is: anything that is a weighted sum of the input.

A matrix can rotate a set of points. It can stretch one direction and squash another. It can project 768 numbers down to 2 so you can plot them. It can copy an input into three separate outputs, which is precisely what a transformer does to build queries, keys and values from one hidden state.

What it cannot do is bend. Straight lines stay straight, parallel lines stay parallel, and the origin stays put. A matrix cannot decide "only respond if this value is above 30". That is a bend, and no grid of numbers will produce one.

Why that limit forces activation functions

Apply matrix `A`, then apply matrix `B`. Ask a question of the answers to some questions.

The result is the same as applying one single matrix, `BA`. Not similar to it. Identical to it, for every possible input. Multiplying matrices together is exactly the operation of composing them into one.

So a hundred stacked linear layers can do precisely as much as one linear layer. Not less, but not one bit more. All those parameters, all that compute, and the model can still only draw straight lines.

That is the entire reason ReLU exists. Between the layers, do something a matrix cannot do:

```
def relu(x):
    return max(0.0, x)
```

Negative becomes zero, positive passes through. It is barely an operation. But it is a bend, and it cannot be absorbed into the neighbouring matrices. Now `B(relu(A(x)))` is genuinely a new function that no single matrix can reproduce, and stacking starts buying you something. Depth in neural networks is not matrices piled up. It is matrices with bends wedged between them, and the bends are what makes the pile worth having.

A practical read

When you next see a model summary listing shapes — `(768, 3072)`, `(3072, 768)`, `(768, 50257)` — you can read the pipeline directly. Expand 768 features into 3072 questions. Bend. Compress the 3072 answers back into 768. That final `(768, 50257)` is one row per token in the vocabulary, each row asking "how much does this hidden state look like my token?"

No grids required.

BEFORE YOU MOVE ON

Someone stacks two linear layers with no activation between them and finds the model is no better than a single linear layer. What is the real reason?

A. Two matrices applied in a row are the same as one matrix, so a single layer can already compute everything the pair can.

B. Gradients vanish through the second layer without a nonlinearity, so it never learns anything useful.

C. The extra layer does add capacity, but the doubled parameter count overfits and cancels the gain.

D. Both layers converge to the same weights, which makes the second one redundant.

12 · 9 min

Shapes, batches, and the broadcasting rules

THE ONE THING TO KEEP

Broadcasting silently stretches a dimension of size one to match its neighbour, which makes concise code possible and makes a whole family of bugs run without complaint.

The batch dimension is a convenience, not a concept

Real code never multiplies one vector at a time. It stacks 32 or 256 examples into a batch and processes them together, because a GPU is a machine for doing the same arithmetic to many things at once. So every tensor grows a leading dimension, and the shapes you read look like `(32, 128, 768)`: 32 examples, 128 token positions each, 768 features each.

The rule that keeps this simple is that matrix multiplication in NumPy and PyTorch acts on the **last two dimensions** and treats everything in front as batch. So `(32, 128, 768) @ (768, 3072)` gives `(32, 128, 3072)`: the same 768 × 3072 weight matrix applied to each of the 32 × 128 = 4,096 individual vectors, with no loop written anywhere.

Broadcasting, in three rules

Broadcasting is what lets you write `x + b` where `x` is `(32, 128, 3072)` and `b` is just `(3072,)`. The rules, applied right to left:

1. Line the shapes up from the right, padding the shorter one with 1s on the left.
2. Two dimensions are compatible if they are equal, or if one of them is 1.
3. A dimension of size 1 is stretched — conceptually repeated — to match the other.

Worked:

```
x   (32, 128, 3072)
b               (3072,)   ->  (1, 1, 3072) after padding
                              every axis compatible
result (32, 128, 3072)
```

The bias vector is added to every position of every example. No memory is actually copied; the library reads the same 3,072 numbers repeatedly. That is why broadcasting is fast as well as short.

The bug that costs a week

Now the failure. Broadcasting is *too* accommodating: it will happily produce a valid result from shapes you did not intend to combine.

```
import numpy as np
predictions = np.array([1.0, 2.0, 3.0, 4.0])       # shape (4,)
targets     = np.array([[1.1], [2.1], [2.9], [4.2]])  # shape
(4, 1)
error = predictions - targets
print(error.shape)     # (4, 4)  -- not (4,)
print(error.mean())    # a number, and completely meaningless
```

You wanted four errors. You got sixteen, because `(4,)` padded to `(1, 4)` and `(4, 1)` broadcast against each other into a full 4×4 grid of every prediction against every target. The mean is a real number. The loss decreases during training. The model learns something, slowly and wrongly, and nothing raises an exception.

This is the single most common silent bug in hand-written training code, and it comes from a column vector where a flat vector was expected — most often straight out of a pandas column or a `reshape(-1, 1)` somebody added to satisfy a different function.

The defence is one line:

```
assert predictions.shape == targets.shape, (predictions.shape,
targets.shape)
```

Put it before every loss computation. It costs nothing and it catches this class of bug entirely.

The shape operations, and what each really does

Four operations show up constantly, and confusing them causes the rest of the shape bugs.

- **`reshape`** reinterprets the same numbers in the same memory order under a new shape. `(2, 6)` to `(3, 4)` is legal because both hold 12 numbers. It never moves data between positions in the flat ordering.
- **`transpose`** genuinely reorders axes. `(32, 128, 768)` transposed to `(32, 768, 128)` puts different numbers in different places. Reshape and transpose are not interchangeable, and swapping them is the classic bug when splitting attention heads.
- **`squeeze`** removes dimensions of size 1; **`unsqueeze`** (or `None` indexing) inserts one. These exist mainly to make broadcasting do what you meant.
- **`view` versus `copy`.** Reshape usually returns a view sharing memory with the original, so writing to one changes the other. Transpose does too. This is fast and occasionally surprising.

Getting the head split right

The place this all comes together is multi-head attention. A tensor of shape `(batch, seq, d_model)` with `d_model = 768` and 12 heads has to become `(batch, heads, seq, d_head)` with `d_head = 64`. The correct sequence is reshape then transpose:

```
x = x.reshape(batch, seq, 12, 64)      # split the feature axis
x = x.transpose(0, 2, 1, 3)            # (batch, heads, seq,
d_head)
```

Reshaping straight to `(batch, 12, seq, 64)` is legal arithmetic and gives the wrong answer: it slices the sequence across heads instead of slicing the features. The model still trains. It just learns a worse function, and you have no error to chase.

The rule to keep

Every dimension of size 1 is an invitation for the library to stretch it. That is a feature when you meant it and a silent bug when you did not, so assert the shapes you expect rather than trusting that a result which computed is a result which is correct.

BEFORE YOU MOVE ON

Training loss decreases smoothly but final accuracy is far worse than a baseline. Inspecting the code shows `loss = ((pred - y) ** 2).mean()` where `pred` has shape `(64,)` and `y` has shape `(64, 1)`. What is happening?

A. The squared error is being computed on unnormalised targets, so the loss scale is wrong but the gradients still point the right way.

B. Broadcasting produces a 64x64 grid of every prediction against every target, so the loss averages 4,096 mostly irrelevant pairs and the gradient is dominated by comparisons that should not exist.

C. The mean is taken over the wrong axis, so only the first row contributes to the loss.

D. `(64,)` and `(64, 1)` are incompatible, so the operation silently returns zeros and the model learns nothing.

13 · 9 min

Undoing a matrix, and why you rarely should

THE ONE THING TO KEEP

The inverse exists only when a matrix loses no information, and even then solving a system directly is faster and far more numerically stable than forming the inverse.

The matrix that does nothing

The identity matrix has 1s down the diagonal and 0s everywhere else:

```
I = [ 1  0  0 ]
    [ 0  1  0 ]
    [ 0  0  1 ]
```

Multiply any vector by it and you get the vector back. Multiply any matrix by it and you get that matrix back. It is the matrix equivalent of the number 1, and it exists so that "undoing" has something to mean.

The inverse

The inverse of A, written `A^-1`, is the matrix that undoes A: `A^-1 A = I`. If A rotates, `A^-1` rotates back. If A stretches by 3, `A^-1` shrinks by 3.

For a 2×2 there is a formula worth seeing once, because it exposes the whole story:

```
A = [ a  b ]      A^-1 =    1     [  d  -b ]
    [ c  d ]              ad - bc [ -c   a ]
```

The quantity `ad - bc` is the determinant. When it is zero, the formula divides by zero and no inverse exists. Worked:

```
A = [ 2  1 ]   det = 2*3 - 1*1 = 5
    [ 1  3 ]
A^-1 = (1/5) [  3  -1 ]  =  [  0.6  -0.2 ]
             [ -1   2 ]     [ -0.2   0.4 ]
check: A A^-1 = [1 0; 0 1]   (multiply it out; it does)
```

When there is no inverse

A matrix has no inverse when it destroys information — when two different inputs produce the same output. Consider

```
A = [ 1  2 ]      A [1, 0] = [1, 2]     A [0, 1] = [2, 4]
    [ 2  4 ]
```

The second row is twice the first, so every output lands on a single line. Given an output you cannot recover which input produced it, because infinitely many did. Such a matrix is called singular, and its determinant is `1*4 - 2*2 = 0`.

This matters more than it first appears. A layer whose weight matrix is singular has thrown information away permanently, and no later layer can retrieve it. Real weight matrices are almost never exactly singular, but many are *close*, which is the practically important case.

Nearly singular: the condition number

The useful measure is the condition number, roughly the ratio of the largest to the smallest singular value. A well-conditioned matrix has a condition number in the tens. A badly conditioned one has it in the millions, and it means a tiny change in the input produces an enormous change in the output — or, in the direction that hurts, that a tiny rounding error in your data becomes an enormous error in the solution.

A rough rule: with 32-bit floats you have about 7 decimal digits of precision, so a condition number of `10^6` leaves you roughly one meaningful digit. That single fact explains most of the "why is my linear solve producing nonsense" problems in practice.

Why you should not compute the inverse

Here is a habit that separates people who have been burnt from people who have not. To solve `Ax = b`, do **not** compute `A^-1` and multiply.

```
import numpy as np
x = np.linalg.inv(A) @ b     # slower, less accurate, avoid
x = np.linalg.solve(A, b)    # do this instead
```

Three reasons. Forming the inverse costs roughly three times the work of solving directly. It is measurably less accurate, because it computes an intermediate object with its own rounding errors before using it. And it is unnecessary: `solve` uses LU decomposition, which never builds the inverse at all.

The rule generalises. In numerical code, the answer to "how do I undo this matrix" is almost always "solve the system", not "invert the matrix". Library authors know this, which is why `np.linalg.solve` exists and why the documentation for `np.linalg.inv` gently steers you away.

Where this shows up in machine learning

Explicit matrix inversion is rare in deep learning, which is worth saying plainly: nothing in transformer training inverts a matrix. It appears in three neighbouring places.

- **Linear regression's closed form** involves `(X^T X)^-1`, and this is exactly where the advice bites. When two features are highly correlated, `X^T X` is close to singular, the condition number explodes, and the fitted coefficients become huge with opposite signs. This is multicollinearity, and adding a ridge penalty — `(X^T X + λI)^-1` — fixes it precisely by pushing the smallest singular values away from zero.
- **Gaussian processes and Kalman filters** invert covariance matrices, and practitioners of both spend real effort on numerical conditioning, usually via Cholesky decomposition rather than an inverse.
- **Second-order optimisers** would like the inverse Hessian and cannot afford it; every practical method, from L-BFGS to Adam's diagonal approximation, is a way of avoiding that inversion.

The rule to keep

`solve`, never `inv`. And when a solve returns absurd numbers, check the condition number before you check anything else — `np.linalg.cond(A)` is one call, and it usually names the problem immediately.

BEFORE YOU MOVE ON

A linear regression fitted with the closed-form solution returns coefficients like +48,200 and -48,190 on two features that are almost perfectly correlated. What is the mechanism, and what does adding a small ridge penalty do?

A. The features were not standardised, so their scales differ by four orders of magnitude; ridge rescales them.

B. The model is overfitting the training data; ridge reduces capacity by removing features.

C. There are more features than examples, so the system is underdetermined; ridge selects one of the infinitely many solutions.

D. The correlated features make X-transpose-X nearly singular so its smallest singular value is close to zero; the ridge term adds lambda to every eigenvalue, pushing it away from zero and shrinking the solution.

14 · 8 min

The determinant, and what it says about collapse

THE ONE THING TO KEEP

The determinant is the factor by which a transformation multiplies volume, so a determinant of zero means dimensions were flattened away and nothing downstream can restore them.

One number for a whole transformation

The determinant of a square matrix is a single number, and it has a clean geometric meaning: **it is the factor by which the transformation multiplies area, or volume, or the higher-dimensional equivalent.**

Take the unit square with corners at (0,0), (1,0), (0,1), (1,1) — area 1. Apply

```
A = [ 3  0 ]
    [ 0  2 ]
```

It becomes a rectangle 3 wide and 2 tall, area 6. And `det(A) = 3*2 - 0*0 = 6`. The number is the area scaling, exactly.

Now a shear:

```
B = [ 1  1 ]     det = 1*1 - 1*0 = 1
    [ 0  1 ]
```

The square becomes a slanted parallelogram — different shape, same area. The determinant says 1, and it is right.

Zero means collapse

The case that matters is `det = 0`.

```
C = [ 1  2 ]     det = 1*4 − 2*2 = 0
    [ 2  4 ]
```

The unit square is squashed onto a line segment. A line has zero area, so the scaling factor is zero. That is the geometric version of the fact from the previous lesson: this matrix destroys a dimension, and destroyed dimensions cannot be recovered. Determinant zero, no inverse, information gone. They are three statements of one thing.

Negative determinants are also meaningful: they say the transformation flipped orientation, turning a left-handed arrangement into a right-handed one, like a mirror. `det = −2` means area doubled and the space was reflected.

Computing it, and why you usually should not by hand

For a 2×2, `ad − bc`. For a 3×3 there is a longer formula. For anything larger, do not use the recursive cofactor expansion you may have been taught — it costs on the order of `n!` operations, which for a 20×20 matrix is more arithmetic than the age of the universe permits. Real implementations use LU decomposition and cost about `n^3`, the same as a matrix multiplication.

And even then, the determinant itself is a poor thing to compute. Multiply a 100×100 matrix's entries by 2 and its determinant multiplies by `2^100`, about `1.3 × 10^30`. Determinants overflow and underflow floating-point range almost immediately at realistic sizes, which is why any library that needs one offers `slogdet` — the sign and the *logarithm* of the absolute determinant, computed as a sum of logs rather than a product.

```
import numpy as np
sign, logdet = np.linalg.slogdet(A)   # stable; det(A) = sign *
exp(logdet)
```

Where it appears in machine learning

The determinant is not a daily tool, but it sits under three things you may meet.

Normalising flows. These are generative models built from invertible transformations. To know the probability density after a transformation you must correct for how much the transformation squeezed or stretched the space, and that correction is exactly the log absolute determinant of the Jacobian. The entire architectural game in that field — coupling layers, autoregressive flows — is about designing transformations powerful enough to be useful whose Jacobian determinant is cheap to compute, usually by making the Jacobian triangular so the determinant is just the product of the diagonal.

Multivariate Gaussians. The density of a multivariate normal contains a `det(Σ)^(−1/2)` term. Fitting one requires the log determinant of the covariance, computed via Cholesky in every serious implementation.

Diagnosing collapse. If a covariance matrix has a determinant of effectively zero, your features are linearly dependent — one is a combination of the others — and any method that inverts that covariance will fail. Checking the log determinant, or better the singular values, tells you before the failure.

What it does not tell you

The determinant is one number summarising an entire transformation, so it hides almost everything. A matrix that stretches by 100 in one direction and squashes by 100 in another has determinant 1, exactly like the identity, while being wildly ill-conditioned and dangerous to work with. Two matrices with the same determinant can behave nothing alike.

This is the honest limit: the determinant answers "did total volume change" and nothing else. The question you usually want answered is "what happens in each direction", and for that you need the singular values, which are the subject of a later lesson in this module. The determinant is their product. Knowing a product tells you very little about its factors, and here that gap is the whole story.

The rule to keep

Determinant zero means a dimension was flattened. Determinant far from zero tells you almost nothing useful on its own, and if you find yourself needing its actual value, reach for `slogdet` rather than `det`.

BEFORE YOU MOVE ON

Two 512x512 weight matrices both have determinant 1.0. What can you conclude about how they behave?

A. Both preserve volume overall, and nothing more — one could be the identity while the other stretches enormously in some directions and crushes others.

B. Both are orthogonal rotations, since determinant 1 characterises rotation matrices.

C. Both are well-conditioned and safe to invert, since a determinant far from zero means the matrix is not near-singular.

D. Both leave the length of every input vector unchanged.

---

15 · 9 min

The best fit, and what "best" was defined to mean

THE ONE THING TO KEEP

Least squares finds the point in the space your model can reach that is closest to the data, and it does so by making the residual perpendicular to everything the model can represent.

The problem with no exact answer

You have 500 houses, four features each, and one price. You want `X w = y`, where X is 500×4 and w is the four coefficients you are looking for. There are

500 equations and 4 unknowns. Unless the data is astonishingly obliging, no `w` satisfies all 500. There is no exact answer.

So you settle for the closest one, and the definition of "closest" is a choice. Least squares chooses to minimise the sum of squared residuals:

```
minimise  ||X w − y||^2  =  sum over rows of (prediction − actual)^2
```

That is a decision, not a law. Choosing squares rather than absolute values means one house that is ₹50 lakh out counts as much as a hundred houses ₹5 lakh out, because $50^2 = 2{,}500$ and $100 \times 5^2 = 2{,}500$. Whether that is right depends on whether one big mistake really is as bad as a hundred small ones for your purpose. Frequently it is not, and that is the argument for absolute-error regression instead.

The geometry, which makes the formula obvious

Here is the picture worth carrying. `X w` — as w varies over all possibilities — traces out the column space of X: a 4-dimensional flat surface sitting inside 500-dimensional space. Your target `y` is a point in that 500-dimensional space, and almost certainly not on the surface.

The closest point on a flat surface to an outside point is found by dropping a perpendicular. So the best `Xw` is the **projection** of y onto the column space, and the residual `y − Xw` must be perpendicular to that surface — which means perpendicular to every column of X:

```
X^T (y − X w) = 0
```

Rearrange:

```
X^T X w = X^T y            <- the normal equations
w = (X^T X)^−1 X^T y       <- the closed form
```

That is the whole derivation, and it comes from one geometric fact: the error must be orthogonal to everything the model could have produced. The word "normal" in "normal equations" means perpendicular, not typical.

Worked, small

Fit `y = w x` through three points (1, 2), (2, 4.1), (3, 5.8), no intercept.

```
X = [1; 2; 3]        y = [2; 4.1; 5.8]
X^T X = 1 + 4 + 9 = 14
X^T y = 1*2 + 2*4.1 + 3*5.8 = 2 + 8.2 + 17.4 = 27.6
w = 27.6 / 14 = 1.971
```

Predictions: 1.971, 3.943, 5.914. Residuals: +0.029, +0.157, −0.114. Their dot product with the column `[1, 2, 3]` is `0.029 + 0.314 − 0.343 = 0`, up to rounding. The residual really is perpendicular to the column space. The geometry is not a metaphor.

What to run in practice

Do not code the closed form. Use `np.linalg.lstsq`, which solves it via QR decomposition or SVD, is numerically far better behaved, and works even when `X^T X` is singular:

```
import numpy as np
w, residuals, rank, sv = np.linalg.lstsq(X, y, rcond=None)
```

Note what it returns alongside the answer: the rank of X and its singular values. Those tell you whether your features were linearly dependent, which is the failure the closed form hides.

Reading R-squared without being fooled

The standard summary statistic is

```
R^2 = 1 − (sum of squared residuals) / (sum of squared devia-
tions from the mean of y)
```

It says what fraction of the variance in y your model accounts for, compared with predicting the mean every time. $R^2 = 0$ means you did no better than the mean; $R^2 = 1$ means perfect fit.

Two honest warnings. First, R^2 never decreases when you add a feature, even a column of random numbers, so comparing models by R^2 always favours the bigger one. Adjusted R^2 penalises this, imperfectly. Second, R^2 is computed on the data you fitted; the only version worth quoting is computed on data the model has not seen.

The assumptions you have quietly made

Least squares always returns an answer, so it is worth naming what has to be true for the answer to be meaningful.

- **Linearity.** You have assumed y is a straight-line function of the features. Fit a line through a curve and you get a line, with no complaint from the arithmetic.
- **Independent errors.** With time-series data, consecutive residuals are usually correlated, and the coefficient standard errors will be too small — your significance tests become overconfident.
- **Roughly constant error size.** If the errors grow with the prediction, as they do for prices and counts, the fit is dominated by the large end. Taking logs of the target often fixes this in one line.
- **No influential outliers.** Squaring makes a single wild point capable of dragging the whole fit. Always plot residuals against predictions; the plot shows all four of these problems and takes ten seconds.

The rule to keep

Least squares projects your data onto what your model can represent, and calls the perpendicular distance the error. If your model cannot represent the

truth, the projection is still computed, still optimal, and still wrong — optimality is relative to a model class you chose.

BEFORE YOU MOVE ON

A least-squares fit gives the residual vector r. Which statement about r is guaranteed by the way the solution was derived?

A. The residuals sum to exactly zero.

B. The dot product of r with every column of X is zero, because the solution is defined as the point whose residual is perpendicular to the column space.

C. The residuals are normally distributed, since least squares assumes normal errors.

D. The residuals are all smaller in magnitude than they would be under any other coefficient vector.

16 · 9 min

Rank, and why a huge matrix can be two small ones

THE ONE THING TO KEEP

Rank counts the genuinely independent directions in a matrix, and when it is far below the matrix's size the matrix can be factored into two thin ones at a fraction of the parameters.

Counting what is actually there

A matrix has a number of rows and a number of columns. It also has a **rank**: the number of linearly independent rows, which always equals the number of linearly independent columns. Rank counts how many genuinely different directions the matrix contains, as opposed to how many it is written with.

```
A = [ 1  2  3 ]      row 2 = 2 x row 1
    [ 2  4  6 ]      row 3 = 3 x row 1
    [ 3  6  9 ]      rank = 1
```

Three rows, one independent direction. This 3×3 matrix, nine numbers, carries the information of one 3-vector.

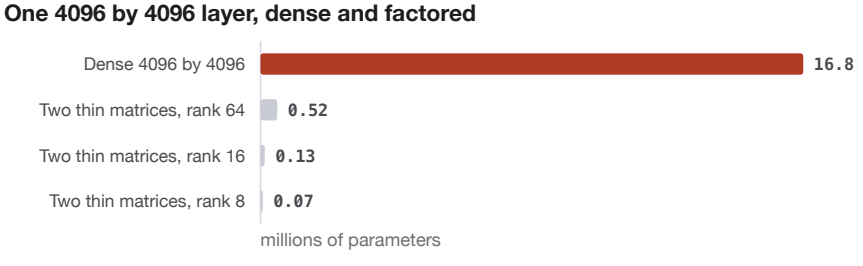
Because rank 1 means every row is a multiple of one row, such a matrix can be written as an outer product:

```
A = [1, 2, 3]^T  x  [1, 2, 3]      3 + 3 = 6 numbers instead of
9
```

That saving is trivial at 3×3. It is not trivial at 4096×4096.

The arithmetic that makes it matter

A dense 4096 × 4096 weight matrix holds 16,777,216 parameters. Suppose it has rank 8 — or is well approximated by a rank-8 matrix. Then it can be written as `A ≈ U V` where U is 4096 × 8 and V is 8 × 4096:



A rank-8 factorisation holds 65,536 numbers against 16,777,216: 256 times fewer, and the reason a fine-tune of a large model fits on one consumer card. It buys that by assuming the change the task needs is low rank, which is a claim about the task and not a theorem.

```
dense:    4096 * 4096 = 16,777,216 parameters
rank 8:   4096 * 8 + 8 * 4096 = 65,536 parameters
ratio:    256x fewer
```

That is the entire idea behind LoRA, the fine-tuning method used almost everywhere people adapt a large model on modest hardware. You freeze the

original weights and learn only a low-rank update `ΔW = BA` with a small inner dimension, typically 8 to 64. The forward pass becomes `W x + B(A x)`, and you train 65,536 numbers instead of 16.7 million.

The empirical claim behind it is that the *change* a fine-tune needs to make is much lower rank than the weights themselves. The original weights need all their capacity; adapting them to one new task apparently does not. That claim is supported by a lot of practice and it is not a theorem — it fails when the new task is far from anything the base model does, and the standard response is to raise the rank until it stops failing.

What rank tells you about a layer

The rank of a weight matrix bounds what the layer can do. A 768 → 768 layer whose weight matrix has rank 100 can only produce outputs lying in a 100-dimensional subspace, whatever the input. It has 590,000 parameters and 100 directions of expressive power.

This is why the low-rank bottleneck appears deliberately in architectures. An autoencoder squeezes through a narrow middle layer precisely to force a low-rank representation and make the network discard everything but the essentials. Attention projects to 64 dimensions per head for a related reason: comparisons happen in a small subspace, cheaply.

It is also why exact rank is the wrong thing to measure on real weights. A trained matrix is almost never exactly rank-deficient — floating-point noise sees to that. What you want is the *effective* rank: how many singular values are large enough to matter. `np.linalg.matrix_rank` uses a tolerance for exactly this reason, and looking at the singular values directly tells you more than the count does.

```
import numpy as np
s = np.linalg.svd(W, compute_uv=False)
print((s > s[0] * 0.01).sum(), "of", len(s), "singular values
above 1% of the largest")
```

Where low rank shows up besides LoRA

- **Recommendation.** A users × items rating matrix is enormous and mostly empty. Matrix factorisation approximates it as (users × k)(k × items) with k around 50 to 200. Those k dimensions are the "latent factors", and the whole approach exists because the true matrix is close to low rank: taste is not arbitrary.
- **Compressing a trained model.** Replace a big layer with two thin ones fitted to the original's behaviour. It works reasonably for layers whose singular values decay quickly and badly for layers whose do not, which is something you can check before you try.
- **PCA and embeddings.** Reducing 768 dimensions to 128 is a low-rank approximation of your data matrix, and the question of how much you lose is the question of how fast the singular values decay.

The limitation nobody mentions

Low-rank methods assume the important structure is *linear*. A matrix can be full rank and still be highly compressible in some non-linear sense, and conversely, data lying on a curved surface may need full rank to describe linearly while having very few real degrees of freedom. Rank measures linear redundancy only.

The practical version of this warning: when a low-rank fine-tune underperforms, the honest first hypothesis is that the required change genuinely was not low rank, not that you chose the learning rate badly. Raising the rank from 8 to 64 costs little and answers the question directly.

The rule to keep

Rank is the number of directions, not the number of numbers. When the second is far larger than the first, you are storing and computing redundancy, and a factorisation will get most of the behaviour for a fraction of the cost.

BEFORE YOU MOVE ON

A team applies LoRA with rank 4 to fine-tune a model for a task quite unlike anything in its pre-training, and quality is poor. Which diagnosis follows most directly from what rank means?

A. The base model's weights are too high-rank for a low-rank update to attach to, so full fine-tuning is the only option.

B. Rank 4 is a very small subspace for the weight change, and a distant task plausibly requires a change that does not fit in it, so raising the rank is the first thing to test.

C. The frozen base weights prevent the adapter from learning anything the base model cannot already do.

D. Rank 4 causes numerical instability because the factor matrices are ill-conditioned.

17 · 9 min

The directions a matrix does not turn

THE ONE THING TO KEEP

An eigenvector is a direction a matrix only stretches rather than rotates, and the largest eigenvalue governs what happens when the matrix is applied over and over.

A question with a surprisingly useful answer

A matrix generally turns vectors: put an arrow in, get an arrow out pointing somewhere else. But for most matrices there are a few special directions that come out pointing exactly where they went in, only longer or shorter. Those are the **eigenvectors**, and the stretch factor for each is its **eigenvalue**.

Written down:

```
A v = lambda v
```

Read it as: applying A to v does the same thing as multiplying v by a single number.

Worked, with a matrix simple enough to check:

```
A = [ 3  1 ]
    [ 0  2 ]

try v = [1, 0]:   A v = [3, 0] = 3 * [1, 0]     eigenvalue 3
try v = [1, -1]:  A v = [3 - 1, -2] = [2, -2] = 2 * [1, -1]
eigenvalue 2
try v = [1, 1]:   A v = [4, 2]                  not a multiple
of [1, 1]
```

Two eigenvectors, eigenvalues 3 and 2. Note that eigenvectors have no fixed length — `[2, 0]` works as well as `[1, 0]` — so implementations return unit-length ones by convention.

Why repeated application is the point

Eigenvectors matter because they tell you what happens when you apply a matrix many times, and many things in computing are exactly that.

Write any vector as a mixture of eigenvectors. Apply A once and each part scales by its eigenvalue. Apply it `n` times and each part scales by `λ^n`. With eigenvalues 3 and 2:

```
after 1 step:  3   and 2       ratio 1.5
after 10 steps: 59,049 and 1,024   ratio 58
after 30 steps: 2.06e14 and 1.07e9  ratio 192,000
```

The largest eigenvalue takes over completely. Whatever direction it belongs to is where any repeatedly-multiplied vector ends up pointing. That is the whole mechanism behind:

- **The power method**, which finds the top eigenvector by multiplying a random vector by A repeatedly and normalising. Five lines of code, no library.
- **PageRank**, which is the top eigenvector of a link matrix. Google's original algorithm was, in its arithmetic, this lesson.
- **Whether a recurrent network's memory decays or explodes.** If the largest eigenvalue of the recurrent weight matrix is below 1, signals shrink towards nothing over time steps; above 1, they blow up. That number is called the spectral radius, and the difficulty of keeping it near 1 is the reason LSTMs and later transformers exist.

```
import numpy as np
v = np.random.randn(A.shape[0])
for _ in range(50):
    v = A @ v
    v /= np.linalg.norm(v)
print(v, v @ A @ v)     # top eigenvector and eigenvalue
```

What eigenvalues mean when they are strange

Complex eigenvalues mean rotation. A pure 90-degree rotation matrix has no real eigenvector at all, which makes sense: it turns every direction, so no direction survives unturned. Its eigenvalues are `i` and `-i`.

Negative eigenvalues mean the direction is preserved but flipped.

Zero eigenvalues mean that direction is annihilated — the matrix is singular, and the count of zero eigenvalues is how many dimensions were destroyed.

Symmetric matrices behave beautifully. If `A = A^T`, all eigenvalues are real and the eigenvectors are mutually perpendicular. Covariance matrices, Hessians and graph Laplacians are all symmetric, which is why the tidy version of this theory covers most of the cases you meet. Use `np.linalg.eigh` for those, not `eig`: it is faster and returns sorted real results.

Where they are and are not used

Eigen-decomposition sits under PCA (eigenvectors of the covariance matrix), spectral clustering (eigenvectors of a graph Laplacian), and the analysis of optimisation landscapes (eigenvalues of the Hessian give the curvature in each direction, and their ratio is the condition number that decides how badly plain gradient descent will zigzag).

It is *not* used inside transformer training, and it is worth being clear about that so you do not go looking for it. There is no eigen-decomposition in a forward pass. Its role is analytical: understanding why training behaves as it does, and building the classical methods that surround the deep model.

The honest limitation

Eigen-decomposition applies only to square matrices, and not even to all of those — some square matrices cannot be diagonalised at all. Most of the matrices you actually care about, weight matrices between layers of different widths, are not square.

The generalisation that works for every matrix without exception is the singular value decomposition, which is the next lesson. If you only have room to keep one of the two, keep that one: SVD covers every matrix, is numerically well behaved, and reduces to the eigen-decomposition for symmetric matrices anyway.

The rule to keep

Eigenvectors answer "what does this matrix do if I apply it forever". When you see a stability question — does this recur, does this converge, does this blow up — the largest eigenvalue is usually the number that decides it.

BEFORE YOU MOVE ON

A recurrent layer's weight matrix has a largest-magnitude eigenvalue of 1.4. Over a 100-step sequence, what does that predict, and why?

A. The hidden state grows roughly as 1.4 to the power of the step count, so it explodes; each application multiplies the dominant component by 1.4 again.

B. The hidden state decays to zero, since eigenvalues above 1 damp the signal.

C. Nothing predictable, because eigenvalues describe a single application and say nothing about repeated ones.

D. The state converges to the top eigenvector with bounded magnitude, since normalisation keeps the length fixed.

18 · 10 min

Singular values, and the honest measure of importance

THE ONE THING TO KEEP

Every matrix without exception factors into a rotation, a set of axis stretches, and another rotation, and the sizes of those stretches tell you exactly how much you lose by throwing each one away.

The factorisation that always exists

Eigenvectors need a square matrix and still sometimes fail. The singular value decomposition has no such conditions. **Every** matrix — rectangular, singular, whatever — can be written as

```
A = U S V^T
```

where U and V are orthogonal (rotations, possibly with a reflection) and S is diagonal with non-negative entries down it. Those diagonal entries are the

singular values, conventionally listed largest first.

The geometric reading is clean and worth memorising: any linear transformation, however complicated it looks, is **rotate, stretch along the new axes, rotate again**. That is all a matrix can do. The singular values are the stretch factors.

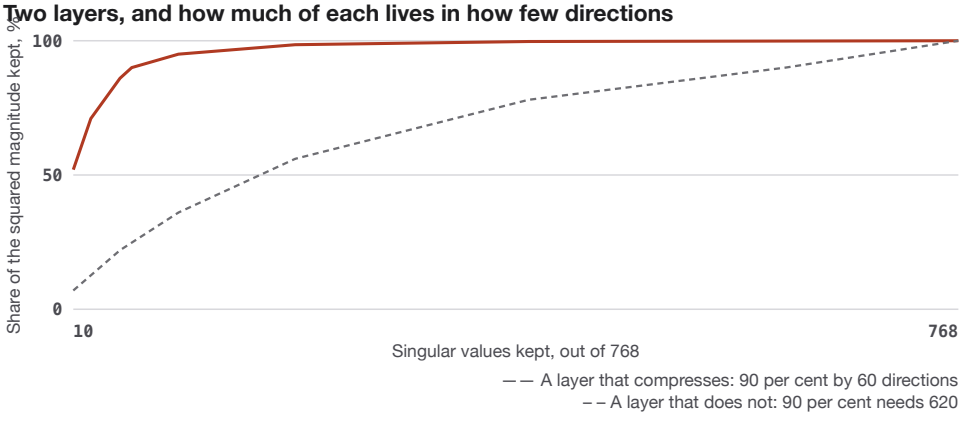
For an `m × n` matrix, there are `min(m, n)` singular values. The number of non-zero ones is the rank. The largest divided by the smallest is the condition number. Three concepts from earlier lessons all fall out of one decomposition.

Reading the decay

The practically useful thing is not the decomposition but the *list* of singular values, because it tells you how much of the matrix lives in how few directions.

```
import numpy as np
s = np.linalg.svd(W, compute_uv=False)
energy = np.cumsum(s**2) / np.sum(s**2)
print("dims for 90% of the energy:", np.searchsorted(energy,
0.90) + 1)
```

Squared singular values sum to the total squared magnitude of the matrix, so the cumulative fraction tells you what proportion you keep. If 90 per cent sits in the first 50 of 768 directions, the matrix is close to rank 50 and you can compress it hard. If you need 700 directions to reach 90 per cent, you cannot, and no amount of tuning will change that.



Four lines of NumPy produce this before you attempt any compression. The first layer can be replaced by two thin ones for a known error; the second cannot, and no amount of tuning changes that.

This is the check to run *before* attempting low-rank compression, not after it disappoints.

Truncation, and the guarantee behind it

Keep only the largest `k` singular values, zero the rest, and multiply back. The result is the best possible rank-`k` approximation of the original matrix, in the sense of squared error. This is the Eckart–Young theorem, and it is unusual in being both a real theorem and directly practical: no cleverer rank-k approximation exists, so if truncated SVD is not good enough, low-rank approximation is not good enough.

The error is exactly computable in advance:

```
squared error of rank-k truncation = sum of the squares of the
discarded singular values
```

You know what compression will cost before you compress. That is rare, and it is why SVD is the right first tool.

Worked, in storage terms. A 4096 × 4096 matrix truncated to rank 256:

```
original:  4096 * 4096       = 16,777,216 numbers
rank 256:  4096*256 + 256 + 256*4096 = 2,097,408 numbers
saving:    8x, and the error is the tail of the singular values
```

Where SVD earns its place

- **PCA is SVD.** Centre your data matrix and take its SVD; the right singular vectors are the principal components and the squared singular values are proportional to the variance explained. Any implementation of PCA you will use calls an SVD routine underneath, because doing it that way avoids forming the covariance matrix and is numerically better.
- **Latent semantic analysis**, the 1990s ancestor of embeddings, is a truncated SVD of a term–document matrix. The idea that meaning lives in a low-dimensional space of a big sparse count matrix is thirty years older than the transformer.
- **Least squares, robustly.** `np.linalg.lstsq` uses SVD, which is why it returns a sensible answer even when the closed form would divide by something close to zero. The pseudoinverse is defined by inverting only the non-negligible singular values.
- **Recommenders.** Matrix factorisation for ratings is SVD's idea, adapted because the actual matrix is mostly missing and plain SVD needs a complete one.
- **Diagnosing a trained model.** The singular value spectrum of a layer tells you whether that layer is using its full width. A rapidly decaying spectrum is a layer that could be smaller.

The cost, and how to avoid paying it

A full SVD of an `n × n` matrix costs about `O(n^3)`. For 4096 that is roughly 7 × 10^10 operations — seconds on a laptop, not free but affordable. For a 50,000 × 768 embedding matrix, a full SVD is wasteful when you only want the top 100 components.

The answer is a truncated or randomised SVD, which computes only the leading components and costs a fraction. `sklearn.decomposition.TruncatedSVD`

and `scipy.sparse.linalg.svds` both do this and both work on sparse matrices, which full SVD cannot. All free, all CPU.

The limitation to state plainly

SVD finds the best **linear** low-dimensional summary. If your data lies on a curved surface — a spiral, a sphere, most real embedding manifolds — the best linear approximation can be poor while a non-linear method does much better. That is why UMAP and t-SNE exist for visualisation and why autoencoders exist for compression.

There is a second, quieter limitation. SVD optimises squared reconstruction error, which is a statement about *fidelity to the numbers*, not about usefulness for your task. A direction with a small singular value can still be the one carrying the signal your classifier needs. Compress by SVD, then measure your actual task metric; do not assume that keeping 95 per cent of the energy keeps 95 per cent of the performance.

The rule to keep

Look at the singular values before you decide anything about a matrix's size. They tell you the rank, the conditioning, and exactly what compression will cost, and they are one function call away.

BEFORE YOU MOVE ON

Before compressing a 2048x2048 layer, an engineer computes its singular values and finds the cumulative squared energy reaches 90% only at component 1,900. What should they conclude?

A. The layer is well-conditioned, so truncation to rank 256 will be safe.

B. The singular values must be computed on the centred matrix, and the uncentred spectrum is misleading.

C. The layer's behaviour genuinely spans nearly all its directions, so no rank-k truncation will approximate it well, and Eckart-Young says no other rank-k method will either.

D. They should try a randomised SVD instead, which will find a better low-rank approximation than the full one.

CASE STUDY · MODULE 2

Rank eight, and the week it nearly cost

A two-person company in Pune adapting an open 1.3-billion-parameter translation model to Marathi legal notices, on one 16 GB graphics card.

The product was narrow and useful: municipal notices, tenancy orders and consumer-court summonses translated from English into Marathi in a register that lawyers would accept. A general model got the words right and the register wrong, rendering "the respondent is hereby directed" into everyday Marathi that read like a text message. Their customers were law clerks, and to a law clerk that is a wrong translation.

They had 9,400 aligned pairs, collected over a year, and one graphics card with 16 GB of memory.

The first question was whether they could train at all. The base model has 1.3 billion parameters. Full fine-tuning holds about sixteen bytes per parameter once the optimiser's two moments and the float32 master copy are counted, which is 20.8 GB before a single activation is stored. Their card could not hold it. Renting two data-centre cards for thirty hours would cost roughly what they spent on rent in a month, and would have to be repeated every time they changed anything.

The alternative was a low-rank adapter. Freeze the base model at two bytes a parameter, 2.6 GB, and train only two thin matrices inserted at each attention projection. At rank 8 across twenty-four layers and four projections of width 2,048, that is about 3.1 million trainable parameters: a quarter of one per cent of the model. Sixteen bytes applies only to those. The whole run fits with room to spare, and takes four hours rather than thirty.

Before committing they ran one check that took ten minutes. They took the singular values of a query projection matrix from the middle of the model and looked at where the squared magnitude accumulated. Ninety per cent of it sat in the first 180 of 2,048 directions. That did not prove a rank-8 adapter would work — the singular values describe the weights, not the change the task

needs — but it told them the matrices were nowhere near using their full width, which made the low-rank story plausible enough to try first.

The rank-8 adapter trained cleanly and the result was disappointing. Everyday sentences improved. The legal register did not: "hereby directed" still came out casual, and a set of twenty fixed formulae that appear in almost every notice were translated inconsistently, sometimes correctly and sometimes not, within the same document.

Here the decision had a cost on both sides, and it is the decision the module exists to prepare you for. The obvious hypothesis, and the one both of them wanted to believe, was that the learning rate was wrong. Adapters start from zero and need a larger rate than a full fine-tune; theirs might have been too small. Testing that meant a sweep, which meant a day, and if it was right they would have saved the memory.

The other hypothesis was less comfortable: that the change the task required simply was not rank 8. Legal register is not a small correction to general translation; it is a systematic shift in word choice across the whole vocabulary. Raising the rank to 64 costs eight times the adapter parameters, which is still only 25 million and still fits, and costs one more four-hour run.

One of them wanted the sweep, because a smaller adapter would keep their deployment cheap. The other wanted the rank raised, because the failure looked systematic rather than under-trained: everyday sentences had improved, which is not what an under-trained model does.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They raised the rank to 64 and kept the learning rate. The four-hour run fixed the register: the twenty fixed formulae came out consistently, and the everyday sentences held their earlier gains. They then went back and did the learn-

ing-rate sweep anyway, at rank 64, and found their original rate had been within a factor of two of the best one — so the day they did not spend on it would have been a day spent confirming that nothing was wrong. The deployed adapter is 50 MB against a 2.6 GB base, and they ship one adapter per customer segment. Their note in the repository reads: when a low-rank fine-tune underperforms, raise the rank before you tune anything, because raising the rank answers the question directly and tuning only ever tells you that tuning was not the problem.

Why did full fine-tuning need 20.8 GB when the model itself is only 2.6 GB in half precision?

Training holds far more than the weights. In the usual mixed-precision recipe with Adam it is about sixteen bytes per parameter: two for the half-precision working weights, four for the float32 master copy, two for the gradient, and four each for Adam's two moment estimates. Twelve of those sixteen bytes are optimiser and master state, which exist only for parameters you are training. That is exactly why freezing the base and training a small adapter collapses the figure: the sixteen bytes then apply to a quarter of one per cent of the model.

The singular-value check showed 90 per cent of the energy in 180 of 2,048 directions. Why was that not enough to justify rank 8?

The singular values describe the existing weight matrix, not the update the task needs. A matrix can be highly redundant while the change required to move it to a new register is spread across many directions. The check ruled out the worst case, a matrix already using its full width, and made the attempt reasonable. Only running it, and then running it at a higher rank, answered the actual question.

Everyday sentences improved while the legal register did not. Why does that pattern point at rank rather than at the learning rate?

An under-trained model, which is what too small a learning rate produces, improves little at everything. Here one kind of change took and another did not, which is the signature of a capacity limit rather than of insufficient optimisation: the adapter could represent the easy correction and could not represent the systematic one. That reading is a hypothesis, not a proof, but it is cheap to test, since raising the rank and rerunning costs one afternoon and settles it.

MODULE 2 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A network stacks a hundred linear layers with no activation function between them. What can it compute that a single linear layer cannot?

A. Nothing: the product of the hundred matrices is one matrix computing the same function

B. Considerably more, because each layer adds parameters and therefore capacity

C. The same functions, but with much better numerical conditioning

D. More, but only for inputs whose dimension exceeds the number of layers

2 Predictions have shape (4,) and targets have shape (4, 1). The code subtracts one from the other and takes the mean. What happens?

A. NumPy raises a shape error, because the two arrays cannot be subtracted

B. The shapes broadcast into a 4 by 4 grid of every prediction against every target, and the mean is meaningless

C. The targets are squeezed to shape (4,) automatically and the result is correct

D. The result has shape (4,) but every sign is reversed

3 A layer's square weight matrix has determinant zero. What follows for everything downstream of that layer?

A. Nothing follows: a determinant of zero only means the layer applies no overall scaling

B. The layer will output NaN, because the inverse divides by the determinant

C. At least one dimension was flattened, and no later layer can recover what was lost

D. The layer is well conditioned, since a small determinant means small, stable outputs

4 Replacing a 4096 by 4096 weight matrix with two matrices of shapes 4096 by 8 and 8 by 4096 cuts the parameters 256-fold. What does the replacement give up?

A. Numerical precision, since two multiplications round twice

B. Nothing measurable: any matrix can be written this way exactly

C. Speed, because two matrix multiplies always cost more than one

D. Expressive power: the outputs now lie in an 8-dimensional subspace, whatever the input

5 Before compressing a layer you find that 90 per cent of the squared magnitude of its weight matrix needs the first 620 of its 768 singular values. What should you conclude?

A. It compresses well, since 620 is fewer than 768

B. It is nearly rank one, since the energy accumulates smoothly

C. It cannot be compressed much by any low-rank method, and no tuning will change that

D. Its condition number is small, so the matrix is safe to invert

6 Least squares chooses the coefficients that make the residual perpendicular to every column of X. What does that geometric condition express?

A. That the residual is as small as it can be, since the closest point on a flat surface is the foot of a perpendicular

B. That the features are uncorrelated with one another, which the method requires

C. That the errors are normally distributed, which perpendicularity guarantees

D. That the model has enough parameters to fit every data point exactly

MODULE 3

Change, slopes and the walk downhill

Derivatives from first principles, the chain rule, backpropagation as bookkeeping, and how to read a loss curve for whether the problem is the learning rate, the curvature or the gradient itself.

> BY THE END OF THIS MODULE YOU CAN
>
> Compute a partial derivative and a chain-rule derivative by hand, trace a gradient backwards through a two-layer network and say where the memory goes, and diagnose from a loss curve and a gradient-norm plot whether the learning rate, the curvature or a vanishing gradient is the cause

---

19 · 8 min

Slope, before anyone says the word derivative

> THE ONE THING TO KEEP
>
> A derivative is the slope between two points as you shrink the gap between them, and you can compute a usable one with two evaluations and a small number.

Two points and a division

Forget calculus for a moment. You have a function — anything that takes a number and returns a number — and you want to know whether it goes up or down near some point, and how steeply.

Pick your point, take a small step, and divide:

```
slope = ( f(x + h) − f(x) ) / h
```

That is the average rate of change over the step. Now shrink `h`. Take `f(x) = x^2` at `x = 3`:

```
h = 1:      (16 − 9) / 1        = 7
h = 0.1:    (9.61 − 9) / 0.1    = 6.1
h = 0.01:   (9.0601 − 9) / 0.01 = 6.01
h = 0.001:  (9.006001 − 9)/.001 = 6.001
```

The numbers are converging on 6. That limit is the derivative of `x²` at 3, and the general formula `2x` gives `2 × 3 = 6`. No mystery: a derivative is the slope between two points, with the gap taken to zero.

Why it matters that it converges

The reason a derivative is useful is that it turns a question about a whole function into a single number at one point. Ask "if I nudge this weight up a little, does the loss go up or down, and how fast?" and the derivative answers it. Every training step is that question, asked millions of times.

The sign carries the direction and the magnitude carries the urgency. Derivative +6 means going right increases the value at six times the rate you go. Derivative 0 means flat, at least locally — a peak, a trough or a plateau.

Do not shrink h too far

Here is the practical wrinkle that a maths course will not tell you and a debugging session will. The formula gets more accurate as `h` shrinks, right up until floating-point arithmetic ruins it.

`f(x + h)` and `f(x)` are nearly equal for tiny `h`, so subtracting them cancels the leading digits and leaves you dividing noise by a very small number. With 64-bit floats the two effects balance around `h ≈ 1e−8`; go smaller and the answer gets *worse*.

```
def f(x): return x**2
x = 3.0
for h in (1e-4, 1e-8, 1e-12, 1e-16):
    print(h, (f(x+h) - f(x)) / h)
# 1e-04  6.000100000012054
# 1e-08  5.999999963900386
# 1e-12  6.000533403494046      <- getting worse
# 1e-16  0.0                    <- x + h rounds back to x
```

The last line is the whole lesson in one number. `3.0 + 1e-16` is exactly `3.0` in double precision, the numerator is zero, and the derivative comes back as zero. This is called catastrophic cancellation and it is why nobody computes gradients this way in production.

The version that is actually used for checking

A better finite difference uses a step in both directions:

```
central difference = ( f(x + h) - f(x - h) ) / (2h)
```

Its error shrinks with h^2 rather than h, so at `h = 1e-5` it is typically accurate to seven or eight digits. This is the standard gradient check: compute the analytic gradient your code produces, compute the central difference, and compare. If they disagree by more than about `1e-5` relative, your backward pass has a bug.

It is worth knowing this exists because a wrong gradient does not crash. It trains — badly, mysteriously, in a way that looks like a hyperparameter problem for a week.

Derivatives you should be able to write down

Five cover most of what you meet, and it is worth being able to produce them without looking:

- `f(x) = c` (a constant): derivative 0. A flat line has no slope.

- `f(x) = x^n`: derivative `n x^(n−1)`. So `x²` gives `2x`, `x³` gives `3x²`, and `x` gives 1.
- `f(x) = e^x`: derivative `e^x`. It is its own derivative, which is why `e` shows up everywhere in this subject.
- `f(x) = ln(x)`: derivative `1/x`. Steep near zero, nearly flat for large x.
- `f(x) = 1/(1 + e^(−x))` (the sigmoid): derivative `f(x) (1 − f(x))`. Worth remembering as written, because it says the slope is computable from the output alone.

That last one has a consequence you will meet in this module. The sigmoid's output is between 0 and 1, so `f(1 − f)` is at most 0.25, at `f = 0.5`, and approaches zero at both ends. A sigmoid that has saturated has almost no gradient, and stacking several multiplies those small numbers together.

The two rules for combinations

- **Sum:** the derivative of `f + g` is the derivative of f plus the derivative of g. Slopes add.
- **Constant multiple:** the derivative of `3f` is 3 times the derivative of f.

Together with the list above these let you differentiate most simple expressions. `f(x) = 3x² + 5x − 7` has derivative `6x + 5`; the constant vanishes because a constant has no slope.

The one remaining rule, for functions inside functions, is the chain rule, and it gets its own lesson because it is the entire mechanism of training a neural network.

The rule to keep

A derivative is a slope, computed by shrinking a gap. You can always check one numerically with a central difference at `h = 1e−5`, and doing so is the fastest way to find out whether a hand-written backward pass is right.

BEFORE YOU MOVE ON

An engineer verifies a hand-written gradient by finite differences and, wanting maximum accuracy, uses h = 1e-15 with 64-bit floats. The check reports gradients of exactly zero everywhere and they conclude the backward pass is broken. What has actually happened?

A. The step is so small that f(x+h) rounds to f(x) in floating point, so the numerator is zero regardless of the true slope.

B. h = 1e-15 is below the smallest representable double, so it is treated as zero.

C. The forward pass must be returning constants, since a correct function would show some change at any step size.

D. Finite differences only work for convex functions, and the loss surface is not convex.

20 · 9 min

Slopes, and why training is walking downhill

THE ONE THING TO KEEP

A derivative says how the loss moves when you nudge one number; training just steps against it, over and over.

The only question a derivative answers

If I nudge this number a tiny bit, how much does that number move, and in which direction?

That is a derivative. Not a limit, not a rule about `x^n` becoming `nx^(n−1)`. A sensitivity.

Make it concrete. You are training a model. The loss right now is 2.6400. You increase one weight by 0.001 and recompute. The loss becomes 2.6431. It went up by 0.0031 when the weight went up by 0.001.

```
slope = 0.0031 / 0.001 = 3.1
```

The derivative of the loss with respect to that weight is 3.1. Read it in units: loss per unit of weight. Positive means increasing the weight increases the loss, so to reduce the loss you should decrease the weight.

That is the entire content of the idea. Everything else is machinery for computing it fast.

The gradient is just all of them at once

A model has more than one weight. GPT-2 small has 124 million. Do the same nudge test for each one — hold everything else still, wiggle this one, see what the loss does — and you get 124 million slopes.

Pack them into a list. That list is the gradient. It is a vector with one slot per parameter, and slot 7 means "how sensitive is the loss to parameter 7, right now, right here".

Nobody computes these by nudging. Backpropagation gets all of them in roughly the cost of one forward pass, using the chain rule. That is an engineering triumph and it does not change what the numbers mean.

Why walk instead of solve

In school you found a minimum by setting the derivative to zero and solving. Why not do that here?

Because you cannot. The loss of a neural network is a function of hundreds of millions of parameters, wrapped in nonlinearities, averaged over data. There is no formula to set to zero and rearrange. No closed form exists, and nobody expects one.

So you do the only thing left. Stand where you are, look at which way is downhill, take a step, look again.

```
w = 0.4200
grad = 3.1        # d(loss)/dw at this point
lr = 3e-4         # learning rate: how big a step

w = w - lr * grad # 0.41907
```

That is gradient descent, complete. The minus sign is the whole idea: the gradient points uphill, so you move against it. Then you do it again. Training a model is that line, executed a few hundred thousand times, on every parameter at once.

The learning rate, and why it is fiddly

A derivative is honest only very near where you measured it. It tells you the slope under your feet, not the shape of the valley.

Step too small and training crawls; you burn a week of GPU time getting nowhere. Step too big and you leap past the bottom and land higher up the far slope, then leap back further, and the loss goes to `nan` in about forty steps. Typical values for transformers land near `3e-4` for pretraining and `1e-5` or lower for fine-tuning, and those numbers are folklore refined by thousands of failed runs, not derived from anything.

What the picture gets wrong

Every tutorial draws a smooth bowl with a ball rolling to the bottom. That drawing is doing you a small amount of harm.

The real loss surface has flat plateaus, narrow ravines where the gradient points across the ravine instead of along it, and vast numbers of points where the slope is near zero without being any kind of bottom. On top of that, you never see the true gradient. You see an estimate from one batch of examples, which is noisy — the next lesson but one is about exactly how noisy.

And the goal is not the lowest point. A model that reached the true minimum of its training loss has memorised the training set and will be worse on anything new. You are walking downhill on a surface you cannot see, using a noisy compass, and you want to stop somewhere reasonable rather than at the

bottom. It works far better than it has any right to, and no one fully agrees on why.

BEFORE YOU MOVE ON

Partway through training, one weight has a gradient far larger than any other weight's. What does that actually tell you?

A. That weight is a long way from its best value and has far to travel.

B. The loss changes steeply if you nudge that weight right here, and nothing about how far away the minimum is.

C. That weight is the most important parameter in the network.

D. The learning rate is set too high for this model.

---

21 · 8 min

Many knobs, one derivative each

> THE ONE THING TO KEEP
>
> A partial derivative is the slope in one variable with all others frozen, and the gradient collects them into a vector that points in the direction of steepest increase.

Freeze everything else

A loss depends on millions of weights, not one. The extension is simpler than it sounds: to find the partial derivative with respect to one variable, treat every other variable as a constant and differentiate as usual.

Take `f(x, y) = x^2 + 3xy + y^3`.

For the partial with respect to x, pretend y is a fixed number:

```
d/dx of x^2   = 2x
d/dx of 3xy   = 3y        (y is a constant, so this is 3y times
x)
d/dx of y^3   = 0         (no x in it, so it is a constant)

df/dx = 2x + 3y
```

Now the same for y, pretending x is fixed:

```
df/dy = 0 + 3x + 3y^2 = 3x + 3y^2
```

Evaluate both at the point (2, 1):

```
df/dx = 2(2) + 3(1) = 7
df/dy = 3(2) + 3(1) = 9
```

So at (2, 1), nudging x up increases f at rate 7, and nudging y up increases it at rate 9.

The gradient is the collection

Stack the partials into a vector and you have the gradient, written `∇f`:

```
grad f at (2, 1) = [7, 9]
```

This vector has two properties that carry the whole of gradient-based training.

It points in the direction of steepest increase. Not "a direction that increases f" — the steepest one, out of all possible directions. Move against it and you descend as fast as locally possible.

Its length says how steep. `||[7, 9]|| = sqrt(49 + 81) = 11.4`. A gradient norm of 11.4 means the function is changing quickly here. A norm near zero means you are somewhere flat.

That second quantity is the one to watch during training. `grad_norm` in a training log is exactly this, computed over every parameter at once, and its behaviour tells you more about what is going wrong than the loss does.

Reading the direction claim carefully

Why is the gradient the *steepest* direction rather than merely an uphill one? Because the rate of change in any unit direction `u` is the dot product `∇f · u`, and from the module on vectors, a dot product with a fixed vector is largest when `u` points the same way as that vector. The gradient is not defined as steepest ascent; it turns out to be it.

The consequence used everywhere: to go down fastest, step along `−∇f`. That is gradient descent, and the whole of it is

```
w  <-  w − learning_rate ∗ grad
```

repeated. With the gradient `[7, 9]` and a learning rate of 0.1, you move from (2, 1) to `(2 − 0.7, 1 − 0.9) = (1.3, 0.1)`.

Steepest is local, not global

The word to hold onto is *locally*. The gradient describes the surface at the exact point you are standing on and says nothing about what is over the hill. In a long, narrow valley — the common case in real loss landscapes — the steepest direction points at the near wall, not along the valley floor. You cross the valley, overshoot slightly, cross back, and progress along the floor is slow.

That zigzag is the standard picture of ill-conditioning. It is caused by curvature differing wildly between directions, and it is the problem that momentum, Adam and learning-rate schedules all exist to reduce. Nothing is broken when it happens; the gradient is doing exactly what it promised, which is only ever a local promise.

The scale of it in a real model

For a model with 7 billion parameters, the gradient is a vector with 7 billion entries — one partial derivative per weight, recomputed on every batch. Stored as 32-bit floats that is 28 GB, which is why the memory arithmetic of training is dominated by gradients and optimiser state rather than by the weights themselves.

Nobody computes those partials one at a time. Automatic differentiation gets all of them in roughly the cost of one extra forward pass, which is the subject of the backpropagation lesson two along from here. It is worth pausing on how surprising that is: seven billion derivatives for the price of about two function evaluations.

Checking your intuition on a real surface

```
import numpy as np
def f(v): x, y = v; return x**2 + 3*x*y + y**3
def grad(v): x, y = v; return np.array([2*x + 3*y, 3*x +
3*y**2])

v = np.array([2.0, 1.0])
for step in range(5):
    g = grad(v)
    v = v - 0.05 * g
    print(step, v.round(3), f(v).round(4),
np.linalg.norm(g).round(3))
```

Watch the third column fall and the fourth shrink. When the gradient norm approaches zero you have arrived somewhere flat — which may be a minimum, and may not be.

The rule to keep

One partial derivative per parameter, collected into a vector that points uphill fastest. Its length is the number to watch: a gradient norm that collapses towards zero or explodes towards thousands is telling you about your training long before the loss curve does.

BEFORE YOU MOVE ON

During training, the loss is barely moving but the logged gradient norm is large and roughly constant. Which explanation fits the mechanism best?

A. The model has converged to a minimum, where the gradient is naturally flat.

B. Steps are bouncing across a narrow, steeply-curved valley rather than travelling along it, so each step is large but the net progress is small.

C. The gradient is being computed on the wrong loss, since a large gradient always produces a large loss change.

D. The learning rate is too small, so the large gradient is being multiplied down to nothing.

22 · 8 min

The chain rule, which is the whole trick

THE ONE THING TO KEEP

When functions are nested, their rates of change multiply, which is why a network of many layers is differentiable at all and why signals can shrink or blow up as they pass back.

Rates multiply

If a car's fuel use depends on speed, and speed depends on the accelerator position, how does fuel use depend on the accelerator? Multiply the two rates. That is the chain rule, and it is the only calculus fact you genuinely have to internalise for machine learning.

Formally, if `y = f(u)` and `u = g(x)`:

```
dy/dx = (dy/du) * (du/dx)
```

Worked. Let `y = (3x + 1)^2`. Set `u = 3x + 1`, so `y = u^2`.

```
dy/du = 2u = 2(3x + 1)
du/dx = 3
dy/dx = 2(3x + 1) * 3 = 6(3x + 1) = 18x + 6
```

Check it by expanding first: `y = 9x² + 6x + 1`, derivative `18x + 6`. Same answer. The chain rule saved you the expansion, and in a network the expansion is not available at any price.

Longer chains

Nesting three deep just multiplies three factors:

```
y = f(g(h(x)))
dy/dx = f'(g(h(x))) * g'(h(x)) * h'(x)
```

A neural network with 40 layers is a function nested 40 deep — plus an activation function inside each layer, so more like 80. The derivative of the loss with respect to a first-layer weight is a product of around 80 terms.

That single sentence explains two of the most important phenomena in deep learning.

Why gradients vanish, in arithmetic

Suppose each of those factors is around 0.5 — entirely plausible for a sigmoid, whose derivative maxes out at 0.25.

```
0.5^10 = 0.00098
0.5^20 = 0.00000095
0.5^40 = 9.1e-13
```

By layer 40 the gradient reaching the first layer is a millionth of a millionth of what reaches the last. In 32-bit floating point it is indistinguishable from zero. The early layers receive no signal and do not learn. This is the vanishing gradi-

ent problem, and it is not a bug in anyone's code — it is a product of small numbers, behaving exactly as multiplication does.

Now suppose each factor is 1.5:

```
1.5^10 = 57.7
1.5^20 = 3,325
1.5^40 = 11,057,332
```

That is the exploding gradient, which shows up as a loss that suddenly becomes `NaN`.

The architectural responses all attack the factors directly. ReLU has derivative exactly 1 for positive inputs, so it contributes a factor of 1 rather than 0.25. A residual connection computes `x + F(x)`, whose derivative is `1 + F'(x)` — a path through which the gradient passes with factor 1 no matter what the layer does. Normalisation layers keep activations in a range where derivatives are not tiny. Every one of these is a fix for a product of eighty numbers.

The multivariable version

When a variable influences the output through several paths, the chain rule sums over the paths as well as multiplying along them:

```
dL/dx = sum over paths of (product of derivatives along that
path)
```

This is what automatic differentiation implements. The computation is a graph; each edge has a local derivative; the derivative of the output with respect to any input is the sum over all routes of the product along each route. Backpropagation is an efficient ordering of that sum, not a different idea.

Worked on a tiny network

One neuron, sigmoid activation, squared loss. Forward:

```
z = w x + b
a = sigmoid(z)
L = (a − y)^2
```

Backward, one factor at a time:

```
dL/da = 2(a − y)
da/dz = a(1 − a)
dz/dw = x

dL/dw = 2(a − y) ∗ a(1 − a) ∗ x
```

Put numbers in: `x = 2`, `w = 0.5`, `b = 0`, `y = 1`.

```
z = 1.0
a = 1/(1 + e^−1) = 0.7311
L = (0.7311 − 1)^2 = 0.0723
dL/dw = 2(−0.2689) ∗ (0.7311)(0.2689) ∗ 2 = −0.2115
```

Negative, so increasing w decreases the loss — which is right, since a needs to rise towards 1. Notice the middle factor, `a(1 − a) = 0.197`. If `a` had been 0.99, that factor would be 0.0099 and the gradient would be twenty times smaller. A confident sigmoid learns slowly even when it is confidently wrong, which is the specific reason cross-entropy loss replaced squared error for classification: the `a(1 − a)` term cancels out of the algebra.

The rule to keep

Nested functions multiply their rates. Chains of many small numbers vanish and chains of many large ones explode, and almost every architectural trick in deep learning is an attempt to keep those factors near one.

BEFORE YOU MOVE ON

Why does a residual connection, computing x + F(x) instead of F(x), help gradients reach early layers?

A. It halves the number of layers the gradient must pass through, since the skip path bypasses one layer each time.
B. It normalises the activations, keeping them in a range where derivatives are large.
C. Its derivative is 1 + F'(x), so there is a path contributing a factor of exactly 1 at each block, and a product of ones does not shrink.
D. It makes the loss surface convex, so gradient descent converges from any starting point.

23 · 10 min

Backpropagation, and where the memory goes

THE ONE THING TO KEEP

Backpropagation computes every gradient in one backward sweep by reusing stored forward values, which is why training memory is dominated by activations rather than by weights.

The problem it solves

You have a network with 7 billion parameters and you need the derivative of the loss with respect to each one. The naive approach is to nudge each parameter, rerun the network, and see how the loss changed. That is 7 billion forward passes per training step. At a tenth of a second each, one step would take twenty-two years.

Backpropagation gets all 7 billion in roughly the cost of two forward passes. The trick is that most of the arithmetic is shared, and doing it backwards is what lets you share it.

The idea in one paragraph

Going forward, you compute and keep the intermediate values. Going backward, you carry a single quantity — the derivative of the loss with respect to the current layer's output — and at each layer you do two things with it: use it to compute this layer's weight gradients, and convert it into the same quantity for the layer below. One sweep, one pass, every gradient.

Worked, on a two-layer network

Small enough to check with a pen. One input, two layers, squared loss.

```
Forward:
  z1 = w1 * x           (cache x)
  a1 = relu(z1)         (cache z1)
  z2 = w2 * a1          (cache a1)
  L  = (z2 - y)^2
```

Numbers: `x = 2`, `w1 = 0.5`, `w2 = 3`, `y = 1`.

```
z1 = 1.0
a1 = relu(1.0) = 1.0
z2 = 3.0
L  = (3 - 1)^2 = 4
```

Now backwards. Start with the derivative of the loss with respect to its own input.

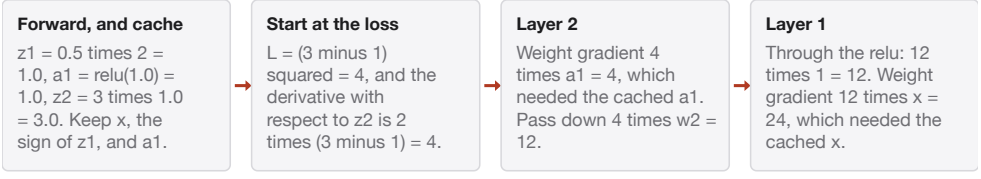
```
dL/dz2 = 2(z2 - y) = 2(3 - 1) = 4

layer 2: dL/dw2 = dL/dz2 * a1 = 4 * 1.0 = 4
         dL/da1 = dL/dz2 * w2 = 4 * 3   = 12       <- pass this
down

layer 1: dL/dz1 = dL/da1 * relu'(z1) = 12 * 1 = 12   (relu' is
1 since z1 > 0)
         dL/dw1 = dL/dz1 * x = 12 * 2 = 24
```

Two weight gradients, 4 and 24, obtained in one backward sweep. Notice that computing `dL/dw2` needed `a1`, and computing `dL/dw1` needed `x` and the sign of `z1`. **Every one of those is a forward value that had to be kept.** That is where the memory goes.

One backward sweep over a two-layer network



Two weight gradients, 4 and 24, from one sweep. Notice which forward values had to survive to make it possible: a1 for the second layer and x for the first. That cache, held from the forward pass until the backward pass reaches it, is where training memory goes.

The memory arithmetic

Here is the practical consequence, and it is the reason training a model needs so much more memory than running it.

For inference, you need the weights and one layer's activations at a time; earlier activations can be discarded as soon as they have been used. For training, every activation needed by the backward pass must survive from the forward pass until the backward pass reaches it. So peak memory scales with **depth × batch size × activation size**, on top of the weights.

Work a rough example. A 7-billion-parameter model, 32 layers, batch of 8 sequences of 2,048 tokens, hidden width 4,096, activations in 16-bit:

```
one layer's activations = 8 * 2048 * 4096 * 2 bytes = 134 MB
32 layers, several tensors kept per layer, say 4:
   32 * 4 * 134 MB = 17 GB of activations
```

against 14 GB for the weights themselves in 16-bit. Activations are the larger half, and they scale with batch size while the weights do not. That is why "reduce the batch size" is the first answer to an out-of-memory error, and why it works when nothing about the model has changed.

Gradient checkpointing, and the trade you are making

The standard remedy is gradient checkpointing: keep only a few activations — say every fourth layer — and recompute the rest during the backward pass. Memory falls by roughly the square root of the depth; compute rises by about 30 per cent, since you do part of the forward pass twice.

It is worth understanding as a pure trade. You are choosing to spend time to save memory, and it is the right trade whenever the alternative is not being able to run at all. One flag in most frameworks.

Two things worth knowing about the real implementation

The graph is built as you go. In PyTorch, every operation on a tensor with `requires_grad=True` records itself. Calling `.backward()` walks that record in reverse. This is why a Python `if` inside a model is fine — the graph records whichever branch actually ran.

Gradients accumulate rather than replace. Calling `.backward()` twice adds the second set to the first, which is a feature — it is how gradient accumulation lets you simulate a large batch on a small card — and a trap, because forgetting `optimizer.zero_grad()` silently sums every batch you have ever seen and your model diverges for no visible reason.

The asymmetry worth knowing

Backpropagation is *reverse-mode* automatic differentiation, and it is cheap when there are many inputs and one output. That is exactly the shape of a loss function: billions of parameters in, one scalar out.

Forward-mode differentiation is the reverse trade and is cheap when there is one input and many outputs. If you ever need the derivative of many outputs with respect to a single parameter, forward mode is the efficient choice, and every serious framework provides it. The reason you rarely hear about it is simply that loss functions return one number.

The rule to keep

Backpropagation is bookkeeping: cache on the way forward, reuse on the way back. The cache is why training memory is dominated by activations, and knowing that turns most out-of-memory errors into a choice between batch size and checkpointing rather than a mystery.

BEFORE YOU MOVE ON

A model trains at batch size 8 but hits out-of-memory at batch size 16, even though the weights occupy well under half the card. What does this tell you about where the memory is going, and why does the model's size stay constant?

A. The optimiser states grow with batch size, since Adam keeps two moments per example.

B. The gradients grow with batch size, since one gradient is computed per example.

C. Cached forward activations dominate and scale linearly with batch size, while weights, gradients and optimiser states do not.

D. The weights are duplicated per batch element so that each example can be processed independently.

24 · 9 min

Convexity, and why nobody promises anything about deep networks

THE ONE THING TO KEEP

A convex problem has one minimum that gradient descent is guaranteed to find, deep networks are not convex, and what saves them in practice is that most flat points in high dimensions are saddles rather than traps.

The shape that comes with a guarantee

A function is convex if the straight line between any two points on its graph lies on or above the graph — a bowl, with no dents. The reason anyone cares is a guarantee: **a convex function has exactly one minimum region, and gradient descent with a sensible step size will find it, from any starting point.** No luck involved, no initialisation to tune.

Convex problems include linear regression with squared loss, logistic regression, support vector machines, and lasso and ridge regression. This is why those methods are so reliable: run the fit twice on the same data and you get the same answer, because there is only one answer to get.

Test for convexity in one variable by looking at the second derivative. `f(x) = x²` has `f'' = 2 > 0` everywhere: convex. `f(x) = x³` has `f'' = 6x`, negative for negative x: not convex. In many variables the equivalent is that the Hessian, the matrix of second derivatives, has no negative eigenvalues.

Neural networks are not convex, and it is not close

Add one hidden layer with a non-linear activation and convexity is gone. It is easy to see why: swap two hidden units, along with their weights, and you get a network computing exactly the same function with different parameters. Two distinct points, identical loss. A convex function cannot have two separate equally-low points with a rise between them, and a network with `n` hid-

den units in a layer has `n!` such copies of every solution. A layer of 100 units has more equivalent minima than there are atoms in the observable universe.

So there are no convergence guarantees for deep learning. None. Everything about why training works is empirical, and it is worth saying that plainly rather than implying there is a proof somewhere you have not read.

Why it works anyway: saddles, not traps

The intuition people carry from two-dimensional pictures — gradient descent trapped in a small local dip, missing the deep valley next door — turns out to be mostly wrong in high dimensions, and the reason is a counting argument.

At any flat point, the surface curves up or down in each of the directions given by the Hessian's eigenvectors. For that point to be a local minimum, it must curve *up* in every single one. In a million-dimensional space, that requires a million independent conditions to hold at once. A point where some directions curve up and others curve down is a **saddle**, and gradient descent escapes a saddle: it simply follows a downward direction out.

Both theoretical analysis of random high-dimensional surfaces and empirical measurement of trained networks support this picture: the flat points encountered during training are overwhelmingly saddles, and the local minima that exist tend to have losses close to one another. Being stuck is far less common than being slow.

Treat this as the current best explanation rather than a settled fact. It is an active area, and the honest statement is that deep networks train reliably, that we have a plausible account of why, and that the account is not a proof.

Sharp and flat, and why it might matter

A related and genuinely contested question is whether the *kind* of minimum matters. The proposal is that wide, flat minima generalise better than narrow, sharp ones, because a flat basin means small perturbations to the weights barely change the loss, and the test distribution is a small perturbation of the training one.

There is supporting evidence, there is a well-known counter-argument that sharpness can be manufactured or removed by rescaling weights without changing the function at all, and there are optimisers built on the idea, such as sharpness-aware minimisation, which do measurably help on some benchmarks. It is a live disagreement. Anyone presenting flat-minima theory as established is going beyond the evidence.

What follows for you in practice

- **Initialisation matters** in a way it does not for convex problems. Different seeds give different final models, with genuinely different behaviour on edge cases even at identical accuracy.
- **Reporting one run is reporting one draw.** If two configurations differ by half a point, run each three times before believing the difference. Seed variance on a small dataset is routinely larger than the effect people publish.
- **Do not go hunting for the global minimum.** It almost certainly corresponds to memorising the training set. What you want is a good enough minimum that generalises, which is why regularisation and early stopping exist.
- **Reach for convex methods when they fit.** If logistic regression solves your problem, its reproducibility and its interpretable coefficients are real advantages, not a consolation prize.

The rule to keep

Convexity buys a guarantee, and deep learning gave it up in exchange for expressive power. What replaced the guarantee is the empirical observation that high-dimensional loss surfaces have far more saddles than traps, which is a good reason for confidence and not a substitute for running your experiment more than once.

BEFORE YOU MOVE ON

Why are true local minima rare relative to saddle points in a loss surface with millions of parameters?

A. Because modern architectures use ReLU, which makes the loss piecewise linear and eliminates local minima.

B. Because the loss is bounded below by zero, so any flat point must be at or near the global minimum.

C. Because being a local minimum requires the surface to curve upward in every one of millions of directions simultaneously, whereas curving down in even one direction makes it an escapable saddle.

D. Because with enough parameters the loss surface becomes convex, so only one minimum exists.

25 · 10 min

Curvature, and what the optimisers are actually doing

THE ONE THING TO KEEP

The ratio between the steepest and shallowest curvature decides how badly plain gradient descent zigzags, and momentum and Adam are two cheap ways of compensating without ever computing that curvature.

The second derivative, in one line

The first derivative says which way is downhill. The second says how the slope itself is changing — whether the surface curves sharply or gently. In many dimensions the second derivatives form a matrix called the Hessian, one entry for every pair of parameters.

You will never compute a Hessian for a real model. For 7 billion parameters it would have 49 × 10^18 entries. But you need one number from it, and it is the

one that explains most optimisation trouble.

The number that explains the zigzag

The Hessian's eigenvalues are the curvatures along its eigen-directions. The ratio of the largest to the smallest is the condition number, and it decides how gradient descent behaves.

Take a simple quadratic bowl with curvature 100 along one axis and 1 along another — condition number 100. Gradient descent must use a learning rate small enough not to diverge along the steep axis, roughly `2/100 = 0.02`. But along the shallow axis, a step of 0.02 makes progress at rate `0.02 × 1`, so covering that direction takes about 100 times as many steps as the steep one.

That is the zigzag: fast oscillation across the narrow direction, crawling progress along the long one. The number of steps needed scales roughly with the condition number, and real loss surfaces have condition numbers in the thousands.

Momentum: remembering where you were going

The fix costs one extra buffer. Instead of stepping along the current gradient, keep a running average of recent gradients and step along that.

```
v  <-  beta * v + grad
w  <-  w - learning_rate * v
```

with `beta` typically 0.9.

Why it works is worth seeing concretely. Along the narrow direction the gradient alternates sign each step, so successive terms in the running average cancel. Along the long direction the gradient points the same way every step, so they accumulate. With `beta = 0.9` the accumulated step approaches `1/(1 - 0.9) = 10` times a single gradient. The consistent direction gets amplified tenfold and the oscillating one gets damped — precisely the correction the geometry needed.

Adam: a different step size per parameter

Adam adds a second idea. Track not just the average gradient but the average *squared* gradient, per parameter, and divide by its square root:

```
m <- beta1 * m + (1 - beta1) * grad          (average gradient)
v <- beta2 * v + (1 - beta2) * grad^2        (average squared
gradient)
m_hat = m / (1 - beta1^t)                    (bias correction)
v_hat = v / (1 - beta2^t)
w <- w - lr * m_hat / (sqrt(v_hat) + eps)
```

Defaults: `beta1 = 0.9`, `beta2 = 0.999`, `eps = 1e-8`.

The division is the substance. A parameter whose gradients are consistently large gets a large `v`, so its effective step is scaled down; a parameter with small gradients gets scaled up. Each parameter ends up with its own step size, adapted automatically. That is a crude, cheap, diagonal-only substitute for using the curvature.

Bias correction is not decoration. Both `m` and `v` start at zero, so early estimates are biased towards zero. At step 1 with `beta2 = 0.999`, the raw `v` is 0.001 times the true squared gradient; dividing by `1 - 0.999^1 = 0.001` restores it. Without this the first few hundred steps take wildly wrong step sizes, which shows up as an unstable start and is the reason warmup helps.

What it costs

Adam stores two extra numbers per parameter. For 7 billion parameters in 32-bit:

```
weights   :  7e9 * 4  = 28 GB
gradients :  7e9 * 4  = 28 GB
Adam m    :  7e9 * 4  = 28 GB
Adam v    :  7e9 * 4  = 28 GB
                        112 GB, before a single activation
```

This is the arithmetic behind the familiar claim that full fine-tuning of a 7B model needs far more memory than you would guess from its size, and behind the popularity of memory-efficient variants: 8-bit Adam quantises the two states, and SGD with momentum keeps only one.

Reading a loss curve for which problem you have

Three patterns, three causes:

- **Loss spikes to NaN early.** The learning rate exceeds the stability limit for the steepest curvature. Lower it, or add warmup so the first steps are small.
- **Loss decreases very slowly and smoothly.** The learning rate is too small, or the problem is badly conditioned. Try raising the rate first; it is one experiment.
- **Loss falls then plateaus while the gradient norm stays high.** Oscillation across a narrow valley. A learning-rate decay usually breaks it, which is the main reason cosine schedules are the default.

Log the gradient norm alongside the loss. Two curves distinguish these cases in seconds; one curve does not.

The honest position on optimisers

AdamW — Adam with weight decay applied separately from the gradient — is the default for transformer training and has been for years. The claim that a newer optimiser beats it is made regularly and survives independent replication rarely, in part because comparisons are extremely sensitive to how much tuning each method received. If you are choosing an optimiser, the evidence supports AdamW plus a careful learning rate over almost anything else, and the time is better spent on the learning rate than on the optimiser.

The rule to keep

Curvature differing across directions is why plain descent is slow. Momentum averages away the oscillation, Adam rescales each parameter by its own gradi-

ent history, and both are approximations chosen because the exact correction is unaffordable.

BEFORE YOU MOVE ON

Adam's update divides by the square root of a running average of squared gradients. What does this achieve that momentum alone does not?

A. It removes noise from the gradient estimate, giving a cleaner descent direction.

B. It gives each parameter its own effective step size, shrinking steps for parameters with consistently large gradients and enlarging them for parameters with small ones.

C. It guarantees the loss decreases at every step, since the step size adapts to the local curvature.

D. It prevents the gradient from vanishing by rescaling small gradients up to a fixed magnitude.

---

26 · 9 min

Reading a training failure from the numbers

THE ONE THING TO KEEP

Almost every training failure announces itself in the gradient norm, the activation statistics or the loss curve before it announces itself as a bad model, and each pattern has a specific mechanism behind it.

Four failures and their signatures

Training goes wrong in a small number of recognisable ways. Each has a mechanism from the earlier lessons in this module, and each has a signature you can see if you log the right two numbers.

Log these from the first run, always:

```
grad_norm = sum(p.grad.pow(2).sum() for p in model.parameters()
if p.grad is not None).sqrt()
```

plus the loss, and if you can, the mean absolute activation at a couple of layers. Three numbers, and they separate the four cases below.

Failure 1: the loss becomes NaN

Signature. Loss is fine, then infinite or NaN, usually within the first few hundred steps, often abruptly.

Mechanism. Either the gradient exploded — the chain-rule product of factors above 1 — or something in the forward pass overflowed, most often a `log(0)` or an `exp` of a large number. In 16-bit floats the maximum representable value is 65,504, which is not a large number at all: a sum of a few hundred squared activations can reach it.

What to do. Gradient clipping, which rescales the whole gradient vector when its norm exceeds a threshold, typically 1.0:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

This preserves direction and caps magnitude. Add warmup so the first steps are small. If you are training in 16-bit, use bf16 rather than fp16 where the hardware allows: bf16 has the same exponent range as fp32 and simply cannot overflow where fp32 would not.

Failure 2: the loss does not move at all

Signature. A flat line from step one. Gradient norm very small, or exactly zero.

Mechanism. Several candidates, distinguishable by the numbers. If the gradient norm is exactly zero, something is detached from the graph or every ReLU is dead. If it is very small but non-zero, you are in the vanishing-gradient case: a deep stack of saturating activations multiplying small factors together.

Dying ReLU deserves its own note because it is common and invisible. A ReLU unit whose input is negative for every example in the data outputs zero always, and its gradient is zero always, so it never recovers. A large learning rate early in training can kill a large fraction of units in one step. The symptom is a network with far less capacity than its parameter count suggests, and the diagnostic is to count how many units are zero across a batch. Leaky ReLU and GELU exist partly to remove this failure, since neither has an exactly-zero-gradient region.

Failure 3: the loss decreases then explodes

Signature. Healthy descent for a while, then a sharp spike, sometimes recovering, sometimes not.

Mechanism. The optimiser has walked into a region of much sharper curvature, where the learning rate that was fine is now above the stability threshold. This is common in language-model training and is the main reason large runs use a decaying schedule: the same learning rate that is right at step 1,000 is too large at step 100,000.

What to do. Clip gradients, decay the learning rate, and if a spike destroys a long run, restart from the most recent checkpoint with a lower rate and skip the batch that caused it — a genuinely standard practice in large-scale training, not a hack.

Failure 4: training loss falls, validation loss rises

Signature. The two curves separate and the gap widens.

Mechanism. Not an optimisation failure at all. The model is fitting the training set, including its noise. This is the only one of the four that gradient statistics will not diagnose, because nothing is going wrong with the descent — it is succeeding at the wrong objective.

What to do. Early stopping, more data, augmentation, regularisation, a smaller model. Note the order: more data is the most reliable and least often available.

Four training failures, and the mechanism behind each

What the two logged numbers show	What is actually happening
Loss fine, then NaN within a few hundred steps	A gradient exploded, or an exp or a log overflowed
Loss flat from step one, gradient norm at or near zero	Dead ReLUs, or a tensor detached from the graph
Loss falls for a while, then spikes sharply	The optimiser reached sharper curvature at the old rate
Training loss falls while validation loss rises	Not an optimisation failure at all: the model is overfitting

Only the last is invisible in the gradient norm, because nothing is going wrong with the descent: it is succeeding at the wrong objective. The other three are three different problems with three different fixes, and the loss curve on its own cannot tell them apart.

The normalisation layers, in one paragraph

Layer normalisation subtracts the mean and divides by the standard deviation across each token's feature vector, then applies a learned scale and shift. Its effect on this module's subject is direct: it keeps the input to each layer in a range where activation derivatives are not tiny, so the chain-rule factors stay near 1. It also makes the loss surface measurably better conditioned, which is why models with normalisation tolerate much higher learning rates than models without.

RMSNorm, used in most recent large models, drops the mean subtraction and divides by the root-mean-square only. It is slightly cheaper and works about as well, which is a fair summary of the evidence.

What to do first, in order

1. **Overfit ten examples.** Before anything else, check the model can drive the loss on ten samples to near zero. If it cannot, there is a bug, and no amount of tuning will help.
2. **Check the data pipeline.** A shuffled label column produces a model that trains perfectly and predicts noise, and it is more common than any of the failures above.
3. **Then tune the learning rate**, which is worth more than every other hyperparameter combined.
4. **Then everything else.**

Most people do these in reverse order, and lose a week.

The rule to keep

Log the gradient norm from the first run. A norm of zero, a norm in the thousands and a norm that is fine while the loss diverges are three different problems with three different fixes, and the loss curve alone cannot tell them apart.

BEFORE YOU MOVE ON

A network trains normally for 2,000 steps, then the loss spikes and never recovers. Gradient norms were around 0.8 throughout, then hit 400 at the spike. Which action addresses the mechanism most directly?

A. Reduce the batch size, since large batches produce noisier gradients that can cause spikes.

B. Switch from ReLU to a leaky variant, since dead units are causing the instability.

C. Clip the gradient norm and decay the learning rate, since the run entered a sharper region where the previously safe step size now overshoots.

D. Increase the learning rate to escape the bad region the model has entered.

27 · 9 min

The one hyperparameter worth your afternoon

THE ONE THING TO KEEP

The learning rate is bounded above by the sharpest curvature and below by your patience, and a single sweep across a few orders of magnitude locates the usable band faster than any amount of guessing.

Why this one and not the others

If you can tune exactly one thing, tune the learning rate. It has a larger effect on final quality than the optimiser choice, the initialisation scheme, the batch size or the architecture width, and it is the only hyperparameter whose wrong value can turn a working model into `NaN` in forty steps.

The reason follows from the previous two lessons. The step you take is `learning_rate × gradient`. Too large and you overshoot along the sharpest-curvature direction and diverge. Too small and you crawl along the shallowest one. The usable band sits between those, it is usually about one order of magnitude wide, and it moves when you change almost anything else about the model.

The range test: twenty minutes, one plot

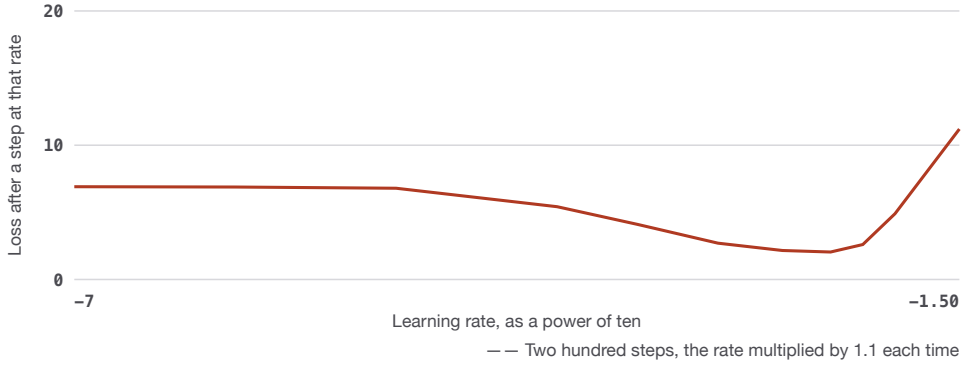
There is a cheap way to find the band, and it is worth doing before every serious run.

Start with an absurdly small learning rate, around `1e−7`. Train for a few hundred steps, multiplying the learning rate by about 1.1 after each one, and record the loss at each rate. Plot loss against learning rate on a log axis.

```
lrs, losses = [], []
lr = 1e-7
for batch in itertools.islice(loader, 200):
    for g in opt.param_groups: g['lr'] = lr
    loss = step(batch)
    lrs.append(lr); losses.append(loss)
    lr *= 1.1
```

The plot has a consistent shape. Flat on the left, where the rate is too small for anything to happen. Then a descent, where the loss falls fastest. Then a sharp rise, where the rate crosses the stability threshold and the run begins to break.

The range test, in twenty minutes and one plot



The usable band is about one order of magnitude wide. Take a rate ten times below where the loss starts to rise, near ten to the minus three here, rather than the exact bottom: a run that survives two hundred steps at the edge will often fail at twenty thousand.

The rate you want is roughly **one order of magnitude below where the loss starts rising** — near the steepest part of the descent, not at its bottom. Picking the exact minimum of the curve puts you at the edge of instability, and a run that survives 200 steps there will frequently fail at 20,000.

Typical values, and why they are so different

The numbers vary by orders of magnitude across settings, and it helps to know the neighbourhood before you start:

- Training a transformer from scratch with AdamW: `1e−4` to `1e−3`.

- Full fine-tuning of a pre-trained model: `1e−5` to `5e−5`. Ten to a hundred times smaller, because you are adjusting a good solution rather than searching for one, and a large step destroys what pre-training built.
- LoRA fine-tuning: `1e−4` to `1e−3`. Larger again, because the adapter starts from zero and has its own scale.
- SGD with momentum on a vision model: `0.01` to `0.1`. Much larger than any Adam figure, because SGD does not divide by the gradient magnitude and Adam effectively does.

If you find yourself needing a rate far outside these bands, the usual cause is a scaling problem in the data or the initialisation rather than a genuinely unusual model.

Warmup, and the reason it exists

Almost every large training run begins with a linear warmup: the learning rate rises from near zero to its target over the first few hundred or few thousand steps.

The mechanism is the one from the Adam lesson. Adam's second-moment estimate `v` is built from very few samples at the start, so the per-parameter step sizes are based on almost no evidence and can be badly wrong. Bias correction fixes the systematic part but not the variance. Warmup simply avoids taking large steps while the estimates are still noise.

There is a second reason for pre-trained models: the first few batches produce large gradients because the new task's head is random, and a full-size step at that moment can wreck weights that took a great deal of compute to produce.

Decay, and what schedule to use

The rate should fall over training, because the region you are optimising in gets sharper as the loss gets lower. Two schedules cover almost everything:

- **Cosine decay** from the peak to near zero over the planned number of steps. This is the default for language-model training and works well. Its

one demand is that you must know the total number of steps in advance, because the shape depends on it.

- **Step decay**, dividing by 10 at fixed points. Older, still fine, and easier to adjust mid-run.

A detail that costs people real runs: if you use cosine decay and then decide to train for longer, you cannot simply continue. The schedule has already annealed to near zero, and extending it means the extra steps do almost nothing. Decide the length first.

Batch size interacts with it

Increase the batch size and the gradient estimate becomes less noisy, so a larger step is safe. Two rough rules circulate: scale the learning rate linearly with batch size (common for SGD on vision tasks), or with the square root of the batch size (often better for Adam). Neither is exact and both break down at very large batches.

The practical version: if you change the batch size by more than a factor of two, re-run the range test. Carrying a learning rate across a batch-size change is one of the most common reasons a configuration that worked for somebody else does not work for you.

The rule to keep

Sweep the learning rate before you tune anything else, choose a value about ten times below where the loss starts to rise, warm up into it and decay away from it. Everything else in the training configuration matters less than getting this one number into the right decade.

BEFORE YOU MOVE ON

A team copies a training configuration that worked at batch size 32 and runs it at batch size 512, changing nothing else. Training is stable but converges to a noticeably worse result. What is the most likely mechanism?

A. The larger batch overfits faster because each step sees more data, so early stopping should trigger sooner.

B. The learning rate is now too small relative to the reduced gradient noise, so the same number of steps covers much less ground and the run under-trains.

C. The larger batch exceeds the model's capacity, so gradients from different examples cancel out.

D. Batch normalisation statistics become unreliable at large batch sizes, degrading the model.

CASE STUDY · MODULE 3

Twenty minutes of a shared GPU night

An agricultural university in Ludhiana, a research assistant training a wheat-disease classifier from 11,000 photographs taken on farmers' phones.

The department had one graphics card and six research students. The rota gave each of them one night a week, ten hours, from eight in the evening. Miss the slot and the next one is seven days away, which in a project with a June deadline is a real loss.

The assistant had 11,000 photographs across seven categories — yellow rust, brown rust, loose smut, karnal bunt, aphid damage, nitrogen deficiency, and healthy — collected by extension officers over two seasons. The plan for the night was to fine-tune an image model that had already been pre-trained on general photographs.

The first attempt died in forty steps. The loss fell from 1.94 to 1.6, then printed as NaN and stayed there. The learning rate was 3e-4, taken from a widely shared post about training transformers.

He restarted with the rate at 1e-4. The loss went to NaN in about ninety steps. He restarted at 5e-5, and it survived, and after two hours the loss was 1.71 and falling so slowly that it would clearly not reach anything useful by morning.

At that point three hours of a ten-hour slot were gone and the decision arrived. He could start a full run at 5e-5 and let it go overnight, and have something to show his supervisor in the morning even if it was poor. Or he could spend twenty minutes on a learning-rate range test — start absurdly low, multiply the rate by 1.1 after each of two hundred steps, and record the loss — which would leave under seven hours for the run and might still produce nothing.

His supervisor's meeting was at eleven the next morning. A run that finished and was mediocre could be discussed. A slot spent on diagnostics with no model at the end could not, and the next slot was a week away.

He also had two numbers he had not looked at. He had never logged the gradient norm, so he could not tell whether the NaN was an explosion or an overflow somewhere in the forward pass. And he had not counted dead units, so he did not know what the two failed runs had done to the network before he restarted them.

He spent five minutes adding both. The gradient norm at 3e-4 rose from 2.1 to 340 over the forty steps before the NaN, which is an explosion and not an overflow. And when he loaded the checkpoint from the failed run and counted, 41 per cent of the units in the third block were producing zero for every image in a batch: a large early step had pushed their inputs negative for the whole dataset, and a unit in that state has no gradient and never comes back.

That second number changed the argument. It meant the run at 5e-5, started from a model already damaged in the first minutes, was not a slow model but a smaller one than its parameter count suggested. Continuing it overnight would produce a number that told him nothing about his data.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

He ran the range test. It took eighteen minutes and showed the loss falling steepest around 5e-5 and rising sharply past 3e-4, which put the usable band an order of magnitude below where he had started; he chose 2e-5, with five hundred steps of warmup so that the first steps could not kill units before the optimiser's estimates had settled. He also added gradient clipping at norm 1.0. The overnight run finished at a validation accuracy of 0.86 on a held-out set of 1,400 photographs, against 0.31 for always predicting the commonest class. He kept the gradient-norm plot in his thesis appendix. His note to the other five students, pinned above the machine, was two lines: log the gradient norm from the first run, and the learning rate a blog post gives you is for training from scratch, not for fine-tuning, which wants ten to a hundred times less.

The gradient norm rose to 340 before the loss became NaN. Why does that distinguish an exploding gradient from an overflow in the forward pass?

An overflow in the forward pass, such as an exp of a large number or a log of zero, produces a non-finite loss directly, and the gradient computed from it is non-finite immediately rather than large and growing. A rising norm over dozens of steps is the chain-rule product of factors above one compounding as the optimiser takes steps that are too large for the local curvature. The two have different fixes: clipping and a lower rate for the first, a numerically stable formulation for the second.

Why does a large learning rate early in training kill ReLU units permanently, and why does warmup help?

A ReLU unit whose input is negative for every example in the data outputs zero always, so its gradient is zero always, and no later step can revive it. One large step can push a whole block of units into that state at once. Warmup keeps the first steps small while the optimiser's estimates are still built from very few samples, so no single early step is large enough to do it.

He had a working configuration at 5e-5 and threw away three hours of the slot to test something else. Was that defensible?

Yes, because the surviving run was starting from a network with 41 per cent of one block dead, so its overnight result would have measured a damaged model rather than his data. A number produced by a run you know to be compromised is worse than no number, since it invites a week of conclusions about the dataset. The range test cost eighteen minutes against a slot of ten hours and told him where the usable band was, which no amount of restarting at guessed values would have done.

MODULE 3 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A 40-layer network uses sigmoid activations, whose derivative peaks at 0.25. Why do the earliest layers stop learning?

A. The first layers have fewer parameters, so they saturate sooner

B. The gradient reaching them is a product of eighty factors below one, which underflows to nothing

C. Sigmoid outputs are always positive, so their gradients cancel one another out

D. The optimiser updates layers in order and runs out of step budget before reaching them

2 Training a model needs far more memory than running it, even at a batch size of one. Why?

A. Gradients are stored in double precision, which doubles the size of every tensor

B. Training loads the whole dataset into memory, while inference loads one example

C. Every layer's forward values must survive until the backward pass reaches them

D. The optimiser keeps a copy of the model for each layer it has updated so far

3 The loss is a flat line from step one and the gradient norm is exactly zero. Which explanation fits?

A. The learning rate is too high, and a lower rate will fix it
B. Something is detached from the graph, or every unit in a layer is dead
C. The model is overfitting, and the training loss has already converged
D. The batch size is too small, so the gradient estimate is too noisy to be useful

4 Why does gradient descent on a million-parameter network get stuck far less often than a two-dimensional picture suggests?

A. High-dimensional loss surfaces are convex, so there is only one minimum to find
B. The optimiser injects noise deliberately, which shakes it out of every trap
C. A flat point is a minimum only if it curves upward in every one of a million directions, and almost all are saddles
D. Modern optimisers compute the Hessian and step around the traps

5 Momentum with a factor of 0.9 speeds up descent in a long, narrow valley. What does the running average do to the two directions?

A. It raises the effective learning rate uniformly, which helps every direction equally
B. Across the valley the gradient alternates sign and cancels; along it the gradient repeats and builds to about ten times one step
C. It divides each parameter's step by the size of that parameter's own gradient history
D. It smooths the loss curve, which makes the plot easier to read and changes nothing else

6 A range test shows the loss falling steepest around 3e-4 and rising sharply past 3e-3. Which rate should you take, and why not the bottom of the curve?

A. 3e-3, the last rate before the rise, to make the fastest possible progress

B. 1e-6, well below the band, because a smaller rate is always the safer choice

C. The rate hardly matters once warmup and gradient clipping are in place

D. 3e-4, about ten times below where the loss rises, since the edge is stable for 200 steps and not for 20,000

MODULE 4

Probability you can rely on

Counting, conditioning, Bayes with real base rates, the distributions that describe real processes, and the specific ways an average misleads when the tail is heavy.

> BY THE END OF THIS MODULE YOU CAN
>
> Compute a posterior probability from a base rate and a classifier's error rates and explain why a 95%-accurate filter can still be wrong most times it fires, choose a distribution that matches a stated generating process, and predict from the shape of a tail when a mean is the wrong summary

---

28 · 8 min

Counting, and the size of the spaces involved

> THE ONE THING TO KEEP
>
> Probability is counting favourable cases over possible ones, and in language and search the possible ones outnumber anything that could ever be enumerated.

Probability starts as division

The oldest definition of a probability is a fraction: the number of outcomes you care about, divided by the number of outcomes there are, when all outcomes are equally likely. Rolling a six on a fair die is 1/6. Drawing a heart from a shuffled pack is 13/52.

The definition is only useful if you can count both numbers, so counting comes first.

The three counting rules

Multiplication. If one choice has `m` options and an independent second choice has `n`, together they have `m × n`. A four-digit PIN has `10 × 10 × 10 × 10 = 10,000` possibilities. Add a fifth digit and it becomes 100,000; each extra position multiplies rather than adds, which is why password length beats password complexity.

Permutations — order matters. The number of ways to arrange `k` items chosen from `n`:

```
P(n, k) = n! / (n − k)!
```

The number of ways to rank the top 3 of 10 candidates is `10 × 9 × 8 = 720`.

Combinations — order does not matter. The number of ways to choose `k` from `n` disregarding order:

```
C(n, k) = n! / (k! (n − k)!)
```

Choosing any 3 of 10 candidates is `720 / 6 = 120`, because each set of three was counted `3! = 6` times in the permutation count.

Worked example with a use: how many ways can you pick 2 features from 20 to test for an interaction? `C(20, 2) = 190`. Test all of them at the usual 5 per cent significance level and you should expect about 9 or 10 to look significant purely by chance. That arithmetic is a whole lesson later in this course, and it starts here.

The number that should change how you think about language

Now apply multiplication to text. A model with a 50,000-token vocabulary generating a 20-token sentence has

```
50,000^20 = about 10^94
```

possible outputs. For scale, there are roughly `10^80` atoms in the observable universe.

Two consequences follow immediately, and both matter.

No model has memorised the space. It could not have. The training data contains perhaps `10^13` tokens, which is a vanishing fraction of `10^94`. Whatever a language model is doing, exhaustive lookup is not available to it, and any explanation of the form "it just retrieves what it saw" fails on arithmetic alone. What it does instead is compress the structure of the space into weights, which is why it can produce fluent sentences nobody has ever written.

Exhaustive search is impossible for generation. You cannot score all continuations and choose the best. Decoding must be greedy or sampled, one token at a time, which is why the sequence a model produces is generally *not* the highest-probability sequence available. Beam search widens the search a little — keeping perhaps 4 or 8 candidates — and even that is a rounding error against `10^94`.

The birthday problem, because your intuition is wrong

Twenty-three people in a room. What is the chance two share a birthday? Most people guess a few per cent. The answer is 50.7 per cent.

The mechanism is that you are not comparing one person against the others; you are comparing every *pair*. With 23 people there are `C(23, 2) = 253` pairs, and 253 chances at 1-in-365 each adds up quickly. Compute it as the complement:

```
P(no match) = (365/365)(364/365)(363/365) ... (343/365) = 0.493
P(at least one match) = 1 − 0.493 = 0.507
```

This is not a party trick. It is the reason hash collisions appear far earlier than the hash space suggests — a 64-bit hash starts colliding around `2^32` items,

not `2^64` — and the reason near-duplicate documents in a corpus are far more common than a per-document duplicate rate implies. Whenever your quantity of interest is about pairs, the count grows with the square.

Counting with the computer

The formulas are in the standard library, and you should not implement factorials yourself.

```
from math import comb, perm, factorial
print(comb(20, 2))            # 190
print(perm(10, 3))            # 720
print(comb(50, 25))           # 126,410,606,437,752
```

That last number is worth staring at. Choosing half of a set of 50 has more than 126 trillion possibilities, which is why "just try all the subsets of features" is never a plan.

The limit of the counting definition

Everything above assumes equally likely outcomes, and almost nothing in machine learning has them. The tokens a model might emit are not equally likely; the documents in a corpus are not equally likely queries. The counting definition is where probability starts historically and where it stops being sufficient almost immediately.

What replaces it is the idea already introduced in this course: a probability as a stated degree of belief, judged by calibration over many cases. Counting remains useful for sizing a space and for knowing when exhaustive anything is off the table. It is a tool for the denominators, not for the beliefs.

The rule to keep

Multiply for independent choices, use combinations when order does not matter, and check the size of the space before proposing to search it. When the count involves pairs, expect coincidences far sooner than the raw space size suggests.

BEFORE YOU MOVE ON

A deduplication pipeline hashes 5 billion documents with a 64-bit hash and treats a hash match as a duplicate. Roughly when should collisions between genuinely different documents start appearing, and why?

A. Around 2^64 documents, since that is the number of distinct hash values available.
B. Never in practice, since 64 bits gives 1.8e19 values and 5 billion is a tiny fraction of it.
C. Around 2^32, roughly 4.3 billion documents, because the number of pairs grows with the square of the count and collisions are a statement about pairs.
D. Only if the hash function is poorly distributed; a good hash avoids collisions entirely below its capacity.

29 · 7 min

Probability is a degree of belief

THE ONE THING TO KEEP

A probability states a degree of belief; you judge it by calibration across many cases, never by one outcome.

Two ways to read the same number

School taught you the first one. Probability is a long-run frequency: roll a die 6000 times, expect roughly 1000 threes, so P(three) is 1/6. Something repeatable, counted.

That reading cannot handle most of what you care about. What is the probability that the rupee weakens against the dollar this quarter? That this patient has dengue rather than chikungunya? That the next word after "I went to the" is "market"? None of these repeat. There is one quarter, one patient, one sentence.

The second reading covers them. A probability is a stated degree of belief, given what you know. 0 means ruled out, 1 means certain, and everything real lives between. The number is not a property of the world. It is a property of your information about the world.

This is the reading a model uses, and once you have it, model outputs stop being mysterious.

What a model is actually saying

A language model does not output words. It outputs a belief spread across every token in its vocabulary, conditioned on everything it has seen so far.

```
P(next = "market"  | "I went to the") = 0.21
P(next = "office"  | "I went to the") = 0.14
P(next = "hospital"| "I went to the") = 0.06
... 50,000 more, all adding to 1.00
```

Two rules constrain that list. Every number is at least zero, and they add to exactly one across the full set of options. Those two rules are why softmax exists: raw model scores can be negative or huge, and softmax exponentiates them (making everything positive) then divides by the total (making them sum to one). It converts scores into a statement of belief.

The vertical bar means "given". Everything a model reports is conditional. Change the context and the belief changes, which is why the same question with a different preamble gets a different answer. That is not the model being unstable. That is the model correctly having different beliefs given different information.

Calibration is the actual test

If a probability is a belief, how do you check it?

Not on one case. If a doctor in Manila says 30% chance of dengue and it turns out to be dengue, she was not wrong. 30% things happen 30% of the time. That is what the number claims.

You check it in bulk. Collect every case where the doctor said 30%, and see whether about 3 in 10 turned out that way. Then do the same for her 70% cases and her 90% cases. If each bucket lands near its label, she is calibrated, and her numbers are worth something. If everything she calls 90% happens half the time, she is overconfident, and you should discount her.

That is the only honest test, and it applies to models exactly as written. Take 10,000 predictions the model gave 0.8 to, and count how many came true. Near 8,000 is calibrated. Near 5,000 means the model's confidence is decoration.

The uncomfortable part

Base models tend to be reasonably calibrated. Instruction-tuned and RLHF-tuned models tend to be worse — the tuning rewards answers that sound helpful and assured, and hedging reads as unhelpful, so confidence gets inflated without accuracy following it up. A model that says "I am 95% sure" in prose is producing text that scored well with raters, which is a different thing from a belief you can bet on. The token probabilities underneath are more trustworthy than the sentence, though not by as much as you would like.

So: a confident model is not a correct model. When someone shows you a system that assigns probabilities, the question is never whether the last prediction was right. It is whether anyone has plotted the buckets.

BEFORE YOU MOVE ON

A weather service said 90% chance of rain in Bogotá on Tuesday. Tuesday was dry. Was the forecast wrong?

A. Yes. 90% was a claim about that specific day, and the day was dry.

B. Yes, and it fits the well-documented pattern of overconfident forecasting systems.

C. You cannot tell from one day. Gather every day it said 90% and check that rain came on roughly nine in ten.

D. The question does not apply. Tuesday happens once, so there is no long run and no way to score the forecast at all.

30 · 8 min

Probability once you know something

THE ONE THING TO KEEP

Conditioning shrinks the space of possibilities to the cases consistent with what you know, and forgetting to shrink the right one is the source of most probability errors.

The bar means "given"

`P(A | B)` is the probability of A once you know B is true. It is not a new kind of probability; it is the same fraction computed over a smaller universe. Formally:

```
P(A | B) = P(A and B) / P(B)
```

Read the denominator as the shrinking. You have thrown away every case where B is false, so you divide by how much of the world is left.

Worked, with a table you can count

A hundred support tickets. Sixty are billing, forty are technical. Of the billing tickets, 45 are resolved on the first reply; of the technical tickets, 20 are.

```
P(billing) = 60/100 = 0.60
P(first-reply resolved) = (45 + 20)/100 = 0.65
P(billing and resolved) = 45/100 = 0.45

P(resolved | billing) = 45/60 = 0.75
P(billing | resolved) = 45/65 = 0.69
```

Note that `P(resolved | billing) = 0.75` and `P(billing | resolved) = 0.69` are different numbers. They answer different questions, and swapping

them is the most common error in applied probability. It has a name — the prosecutor's fallacy — and it appears constantly in reasoning about model outputs: "the model flags 95 per cent of fraud" is `P(flagged | fraud)`, and it tells you almost nothing about `P(fraud | flagged)`, which is the number you act on.

The multiplication rule, and how a language model works

Rearranging the definition gives

```
P(A and B) = P(B) * P(A | B)
```

which extends to any number of events by chaining:

```
P(A, B, C) = P(A) * P(B | A) * P(C | A, B)
```

This chain is the entire probabilistic content of a language model. The probability of a sentence is

```
P(w1, w2, ..., wn) = P(w1) * P(w2 | w1) * P(w3 | w1, w2) * ...
```

Every forward pass computes one factor: the distribution over the next token given everything before it. Generation multiplies them out by sampling one at a time. That is why context matters mechanically rather than metaphorically — the conditioning set is literally the input to the function.

It is also why probabilities of long sequences are computed as sums of logs. Multiplying 500 numbers each below 1 underflows to zero in any floating-point format, a point returned to in the module on logs.

Independence, defined properly

A and B are independent when knowing one tells you nothing about the other:

```
P(A | B) = P(A)      equivalently      P(A and B) = P(A) * P(B)
```

Independence is a strong claim and it is usually false in the data you have. Two support tickets from the same customer on the same day are not independent. Two frames of a video are not independent. Two sentences from the same document are not independent.

This matters more than it sounds, because nearly every statistical formula you will use — standard errors, confidence intervals, significance tests — assumes independent samples. When your samples are clustered, the effective number of independent observations is far below the row count, and every interval you compute is too narrow. A test set of 10,000 sentences drawn from 50 documents does not give you 10,000 independent observations; it gives you something closer to 50.

The base rate, which is the part people drop

Here is the pattern to watch for. Somebody reports "the classifier is 95 per cent accurate on fraudulent transactions". You want to know whether to act on a flag. The missing number is how common fraud is at all.

If fraud is 0.1 per cent of transactions, then out of 100,000 transactions there are 100 fraudulent ones. The classifier catches 95 of them. But it also fires on some fraction of the 99,900 legitimate ones — at even a 1 per cent false-positive rate, that is 999 false alarms. So of 1,094 flags, 95 are real: about 9 per cent.

A 95-per-cent-accurate detector that is wrong 91 per cent of the times it fires. Nothing is broken. The base rate did that, and the next lesson makes the arithmetic general.

The rule to keep

`P(A | B)` and `P(B | A)` are different numbers, and the one you are quoted is usually not the one you need. Ask what the denominator is, every time.

BEFORE YOU MOVE ON

A model card reports "recall of 0.95 on toxic comments". A moderation lead concludes that 95% of the comments the model flags are toxic. What has been confused, and what extra number is needed?

A. Recall has been confused with precision; the missing number is the base rate of toxic comments, since that determines how many false positives accompany the true ones.

B. Recall has been confused with accuracy; the missing number is the overall error rate on the full test set.

C. Nothing is confused; recall and the proportion of correct flags are the same quantity computed from the same confusion matrix.

D. The threshold has not been stated, so recall is undefined; the missing number is the decision threshold.

31 · 10 min

Bayes, done slowly, with real numbers

THE ONE THING TO KEEP

Bayes converts a detector's error rates into the probability that a fired alarm is real, and when the thing being detected is rare, most alarms are false however good the detector is.

The formula, and then the arithmetic that matters

```
P(H | E) = P(E | H) * P(H) / P(E)
```

Four parts, each with a name. `P(H)` is the **prior**: how likely the hypothesis was before the evidence. `P(E | H)` is the **likelihood**: how likely the evidence is if the hypothesis holds. `P(E)` is the **evidence**: how likely the evidence is at all. `P(H | E)` is the **posterior**: what you believe now.

The formula is worth knowing. The arithmetic below is worth more, and you can do it without the formula.

Do it by counting, always

Bayes calculations go wrong when done symbolically and go right when done by imagining a concrete population. Here is a moderation classifier.

A platform's classifier flags policy-violating posts. It is good: it catches 98 per cent of genuine violations, and it wrongly flags 3 per cent of acceptable posts. Violations are 1 per cent of all posts.

A post is flagged. What is the chance it really violates policy?

Take 100,000 posts.

```
Genuine violations:        1,000
  correctly flagged:         980   (98% of 1,000)
  missed:                     20

Acceptable posts:         99,000
  wrongly flagged:         2,970   (3% of 99,000)
  correctly passed:       96,030

Total flagged: 980 + 2,970 = 3,950
Of those, genuine: 980

P(violation | flagged) = 980 / 3,950 = 0.248
```

Twenty-five per cent. Three out of four flags from a 98-per-cent-accurate classifier are wrong. Nobody made a mistake; there are simply ninety-nine times as many acceptable posts, so even a small error rate on the large group swamps the correct detections on the small one.

A 98 per cent classifier on 100,000 posts, 1 per cent of which violate policy

	Flagged	Not flagged
enuinely violating	**980** caught	**20** missed
Acceptable	**2,970** false alarms	**96,030** correctly passed

Three thousand nine hundred and fifty flags, of which 980 are real: precision 24.8 per cent. Halving the false-positive rate to 1.5 per cent lifts precision to 40; lifting recall from 98 to 99 moves it by under a point, because recall acts on the small group and false positives on the large one.

Why this is the most important calculation in applied AI

Every deployment where a model looks for something rare has this structure. Fraud, disease, security intrusions, defects, policy violations, exam cheating, sanctions matches. In each case the base rate is small and the false positives outnumber the true ones.

Three practical implications follow.

Human review capacity must be sized from the flag count, not the violation count. In the example above, 3,950 flags need reviewing per 100,000 posts, and 2,970 of those reviews will find nothing. If you budgeted reviewers on the basis of 1,000 violations, you are short by a factor of four.

The false-positive rate is the number to optimise. Halve it, from 3 per cent to 1.5 per cent, and the arithmetic becomes `980 / (980 + 1,485) = 0.40`. Precision jumps from 25 to 40 per cent from a change that barely moves overall accuracy, because it acts on the large group. Improving recall from 98 to 99 per cent moves precision by less than a percentage point.

Do not tell users a flag means guilt. At 25 per cent precision, treating a flag as a finding is wrong three times in four. It is a reason to look, not a conclusion, and the interface should say so.

Evidence accumulates by multiplying odds

The tidier way to combine several pieces of evidence is in odds form. Odds are `p / (1 − p)`, and Bayes becomes:

```
posterior odds = prior odds x likelihood ratio
```

where the likelihood ratio is `P(E | H) / P(E | not H)`. For the classifier above, `0.98 / 0.03 = 32.7`.

```
prior odds     = 0.01 / 0.99 = 0.0101
posterior odds = 0.0101 x 32.7 = 0.330
posterior prob = 0.330 / 1.330 = 0.248
```

Same answer, and now you can chain. A second independent signal with a likelihood ratio of 10 multiplies again: `0.330 × 10 = 3.30`, giving a posterior of 0.77. Multiplying odds is why the log-odds form appears throughout machine learning — in log space, evidence simply adds.

The word doing the work is *independent*. Two signals derived from the same text are usually correlated, and multiplying their likelihood ratios as if they were independent overstates the evidence, often badly. This is exactly the assumption naive Bayes makes, and it is why naive Bayes produces well-ordered but badly calibrated probabilities: the ranking survives the wrong assumption, the numbers do not.

Where the prior comes from, honestly

The obvious objection is that the prior is a guess. Sometimes it is. Three honest responses:

- **Often it is measurable.** The rate of fraud in your transaction log is a fact you can compute, not an opinion.
- **Sometimes the conclusion is robust to it.** Try 0.5 per cent and 2 per cent as well as 1; if the decision is the same across that range, the prior's uncertainty does not matter here.

- **Sometimes it is not, and you should say so.** Reporting "precision is between 15 and 40 per cent depending on an assumed base rate we have not measured" is more useful than a single number with false precision.

The alternative to a stated prior is not neutrality. It is an unstated prior, usually the assumption that the thing is common, which is the error that produced the 95-per-cent guess in the first place.

The rule to keep

Draw the population. Ten thousand or a hundred thousand cases, four boxes, count them. The formula is easy to misapply and the counting is almost impossible to get wrong.

BEFORE YOU MOVE ON

A rare-disease screening test has 99% sensitivity and 99% specificity, and the disease affects 1 in 10,000 people. A hospital proposes screening everyone. What does the arithmetic say about a positive result?

A. About 99% of positives are genuine, since both error rates are 1%.

B. About 1% of positives are genuine: in a million people there are 100 cases giving 99 true positives, against roughly 9,999 false positives from the healthy.

C. About 50% of positives are genuine, since sensitivity and specificity are equal.

D. The question cannot be answered without knowing how many people are screened.

32 · 9 min

Correlation, dependence, and the gap between them

THE ONE THING TO KEEP

Correlation measures straight-line association only, so a correlation of zero can sit on top of a perfect relationship, and a strong correlation can sit on top of no relationship at all.

What correlation actually computes

The Pearson correlation coefficient is the covariance of two variables divided by the product of their standard deviations:

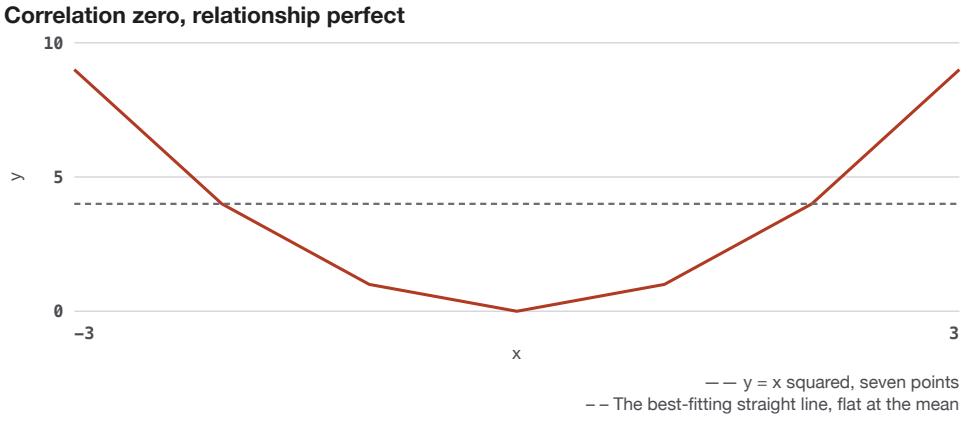
```
r = cov(X, Y) / (sd(X) * sd(Y))
```

It runs from −1 to +1. It answers exactly one question: **how well does a straight line describe the relationship?** Not whether there is a relationship. Whether a straight line captures it.

Zero correlation, perfect relationship

Take `y = x²` with x spread symmetrically around zero: `x = −3, −2, −1, 0, 1, 2, 3`, so `y = 9, 4, 1, 0, 1, 4, 9`.

The correlation is exactly **0**. Knowing x tells you y precisely — there is no uncertainty at all — and Pearson reports no association, because the best-fitting straight line through those points is flat. The upward half and the downward half cancel.



Pearson r is exactly 0.00 on these seven points, and knowing x tells you y with no uncertainty at all. The best straight line through them is flat at y = 4, and a feature-selection step that ranks by correlation throws this feature away along with every other U-shaped one.

This is not a contrived exception. U-shaped relationships are everywhere: performance against arousal, engagement against notification frequency, quality against temperature in a language model. A feature-selection step that ranks features by correlation with the target will drop every one of them.

Strong correlation, no relationship

The reverse failure is more famous and more dangerous. Correlations appear between quantities with no mechanism connecting them, and they appear more often than intuition suggests because people look at many pairs.

Three mechanisms produce them:

- **A common cause.** Ice-cream sales and drowning deaths correlate, both driven by hot weather. Neither causes the other.
- **Selection.** Among people admitted to a competitive university, test scores and interview scores may correlate negatively, because admission required at least one to be high. The correlation is created by the filtering, not by the world. This is called collider bias, and it appears whenever you analyse a sample that was selected on an outcome.
- **Chance across many comparisons.** Test 100 pairs of unrelated variables at the 5 per cent level and about 5 will look significant. This is the

arithmetic from the counting lesson, and it is why exploratory correlation matrices produce so many findings that do not replicate.

Independence is stronger than uncorrelated

Two variables are independent when knowing one changes nothing about the distribution of the other. Uncorrelated means only that the linear component of the association is zero. Independence implies zero correlation; zero correlation does not imply independence, as the $y = x^2$ example shows.

The one place they coincide is jointly normal variables, where zero correlation does mean independence. That special case is why the two ideas get conflated: a great deal of classical statistics assumes normality, and under normality the distinction disappears. Your data is generally not jointly normal.

Measures that catch what correlation misses

- **Spearman correlation** is Pearson applied to the ranks. It catches any monotone relationship, including curved ones, and it is robust to outliers. If you compute one correlation, compute this one alongside it.
- **Mutual information** measures any dependence at all, linear or not, and is zero if and only if the variables are independent. It is harder to estimate from a sample and needs binning or a density estimate, but it is the honest general answer, and it gets its own lesson in the module on information.
- **Plot the thing.** A scatter plot takes five seconds and shows the U-shape, the outlier and the two clusters that every summary statistic hides. Anscombe's quartet — four datasets with identical means, variances and correlations and visibly different shapes — exists precisely to make this point.

Correlated features inside a model

Two consequences matter in practice.

Linear model coefficients become unstable. If two features are 0.98 correlated, the fit can put a large positive weight on one and a large negative weight on the other, with the pair nearly cancelling. Small changes to the data

swing them wildly. The predictions may still be fine; the coefficients are not interpretable, and this is exactly the multicollinearity condition-number problem from the module on matrices.

Feature importance gets split arbitrarily. With two nearly identical features, a tree model uses one about half the time and the other the rest, so each shows roughly half the importance it deserves. Removing either changes nothing, which makes each look useless in an ablation. Always check the correlation structure before believing an importance ranking.

The one sentence about causation

Correlation is evidence about association, and association is consistent with causation in either direction, a common cause, selection, or chance. Establishing a causal direction needs something correlation cannot supply: an intervention, a randomised experiment, or a defensible assumption about the structure that generated the data. This is not a technicality to be waved away with a disclaimer. It is the reason A/B tests exist, and why a model trained on observational data can be an excellent predictor and a terrible guide to what to change.

The rule to keep

Correlation measures straight lines. Plot the data, compute Spearman as well as Pearson, and treat a zero as "no linear component found" rather than "no relationship".

BEFORE YOU MOVE ON

A feature-selection step keeps only features whose absolute Pearson correlation with the target exceeds 0.1, and drops a feature that a domain expert insists is important. What is the most likely explanation the mechanism supports?

A. The feature is redundant with another kept feature, so its independent correlation is suppressed.

B. The feature needs scaling, since Pearson correlation depends on the units of both variables.

C. The relationship is non-monotone — U-shaped, for instance — so the best-fitting straight line is flat and Pearson reports near zero despite a strong dependence.

D. The feature has too many missing values, which drives the correlation towards zero.

33 · 9 min

When you cannot solve it, simulate it

THE ONE THING TO KEEP

Any probability you can describe as a procedure can be estimated by running the procedure many times, and the error of that estimate falls with the square root of the number of runs.

The escape hatch

Analytic probability runs out quickly. The distribution of the maximum of three correlated variables, the chance that a queue exceeds capacity in the next hour, the spread of an A/B test result under a realistic model of user behaviour — all of these are awkward or impossible to write down in closed form.

All of them are easy to simulate. If you can describe how the randomness works as a procedure, you can run the procedure a hundred thousand times and count. This is called Monte Carlo, and it is the most practically useful idea in applied probability.

The pattern, in six lines

```
import numpy as np
rng = np.random.default_rng(0)

trials = 200_000
result = rng.normal(0, 1, trials) + rng.normal(0, 1, trials) >
1.5
print(result.mean())      # 0.1444
```

That estimates the probability that the sum of two standard normals exceeds 1.5. The exact answer is available here — the sum is normal with standard deviation √2, giving 0.1444 — which is exactly why it is a good first example: the simulation agrees, so you can trust the method on the problems where the exact answer is not available.

How accurate is the estimate

The estimate is a proportion from `n` independent trials, so its standard error is

```
se = sqrt( p (1 − p) / n )
```

With `p ≈ 0.14` and `n = 200,000`:

```
se = sqrt(0.14 * 0.86 / 200000) = 0.00078
```

So the answer is 0.144 give or take about 0.0008 — three reliable digits. To get one more digit you need a hundred times more trials, because the error falls with `√n`. That relationship is the whole cost model of simulation: cheap to be roughly right, expensive to be very precise.

A problem worth simulating: the queue

Requests arrive at a service at 100 per second on average, and each takes 8 milliseconds. Average utilisation is 80 per cent, which sounds comfortable. How often does a request wait more than 100 ms?

The analytic answer needs queueing theory. The simulation needs a loop.

```
import numpy as np
rng = np.random.default_rng(1)
n = 200_000
gaps = rng.exponential(1/100, n)          # arrivals, seconds
service = rng.exponential(0.008, n)       # service times, sec-
onds

arrival = np.cumsum(gaps)
finish = np.zeros(n)
prev = 0.0
for i in range(n):                        # a queue is inher-
ently sequential
    start = max(arrival[i], prev)
    prev = finish[i] = start + service[i]
wait = finish - arrival - service
print((wait > 0.1).mean(), np.percentile(wait, 99))
```

You will find several per cent of requests waiting over 100 ms at 80 per cent utilisation, and a 99th percentile far above the 8 ms average. That gap between the mean and the tail is the single most important fact about capacity planning, and it emerges from twelve lines you can run on a laptop.

Two things that ruin a simulation

Correlations you did not model. If arrivals are bursty rather than independent — and real traffic always is — the simulation above understates the tail substantially. A simulation is only as good as the generating story you gave it, and a wrong story produces confident wrong numbers with no warning.

A bad random source, or a shared one. Use `np.random.default_rng(seed)` rather than the legacy global `np.random.seed`. The modern generator has better statistical properties, and passing

an explicit generator object avoids the situation where two parts of your code silently share and reorder each other's random stream. When simulations must be reproducible, seed explicitly and record the seed with the results.

Sampling from distributions you have

Every library gives you the standard ones directly: `rng.normal`, `rng.binomial`, `rng.poisson`, `rng.exponential`, `rng.choice`. For a distribution you have only as a set of weights, `rng.choice(values, p=weights)` handles it, and that is exactly what a language model's sampler does at each step — draw one token from a categorical distribution over 50,000 options.

For resampling from data you already have, rather than a distribution you assumed, the operation is `rng.choice(data, size=len(data), replace=True)`. That is the bootstrap, and it gets a full lesson in the statistics module.

Where simulation beats a formula even when a formula exists

Three cases:

- **Communication.** "In 200,000 simulated weeks, we breached capacity in 3 per cent of them" is understood by everyone in the room. A closed-form tail probability is not.
- **Composition.** Formulas do not compose; simulations do. Add a retry policy, a cache, a second server, and the formula must be re-derived while the simulation gains four lines.
- **Checking your algebra.** If you derived a result analytically, simulate it. Ten minutes, and it catches the sign error that would otherwise ship.

The rule to keep

If you can write down the procedure, you can estimate the probability. Accuracy improves with the square root of the trial count, so budget accordingly, and remember that the simulation tests your assumptions rather than the world.

BEFORE YOU MOVE ON

A Monte Carlo estimate over 10,000 trials gives a probability of 0.020. A colleague wants the estimate accurate to plus or minus 0.0002. Roughly how many trials are needed, and why?

A. About 100,000, since accuracy improves linearly with the number of trials.

B. About 20,000, since doubling the trials halves the error.

C. About 500,000 trials, since the standard error at 10,000 is roughly 0.0014 and reducing it sevenfold requires about fifty times the trials.

D. No number of trials will achieve it, since Monte Carlo error has a floor set by the random generator.

34 · 8 min

Distributions, and why the normal one keeps showing up

THE ONE THING TO KEEP

The normal shape comes from many small independent effects adding up, and most real quantities are not built that way.

A distribution is belief spread over options

One probability answers one question. A distribution answers all of them at once: for every possible value, how much belief goes there.

When the options are countable, that is a list. A language model's next-token output is a distribution over 50,000 tokens, each with its share, all adding to 1. You have already met it.

When the values are continuous — a delivery time, a price in naira, a person's height — you cannot list them, because there are infinitely many. Instead you

describe the shape: where belief piles up, how wide the spread is, how heavy the ends are. A histogram is that shape, drawn from data.

The normal one

The bell curve gets more airtime than every other distribution combined, and mostly for one solid reason.

When a quantity is built by **adding up many small, independent effects**, none of which dominates, the total tends toward a normal shape. That is the central limit theorem, and it is close to a miracle: it does not care what the individual effects look like. Add enough of them and you get a bell.

Adult height is roughly normal because it is thousands of genetic and nutritional contributions summed. Measurement error is roughly normal because it is many tiny independent disturbances summed. Average anything over a decent sample, and the average is roughly normal even when the raw values are not.

A normal distribution needs exactly two numbers: the mean (where the peak sits) and the standard deviation (how wide it is). From those, the rough guides everyone uses: about 68% of values fall within one standard deviation of the mean, about 95% within two.

Where it appears in a model

Weight initialisation. Before training, every weight is drawn from a narrow normal centred on zero, with a standard deviation scaled to the layer's input size — commonly `sqrt(2 / fan_in)`. That scaling is not decoration. Too wide and activations explode through the layers; too narrow and the signal fades to nothing by layer 20. The distribution you start from decides whether the network trains at all.

Diffusion models. The forward process adds normal noise to an image, step by step, until it is indistinguishable from static. Generation runs it backwards: start from a fresh draw of pure normal noise and remove a little at a time. The whole method is built on that distribution specifically, because normal noise added to normal noise stays normal, which makes the maths tractable.

Dropout, sampling, augmentation. Wherever a model needs randomness with no preferred direction, this is the default draw.

Where it does not apply, which is most places

Here is the part that gets skipped, and it causes more damage than everything above combined.

Word frequencies are not normal. They follow Zipf's law: the most common word appears about twice as often as the second, three times as often as the third. "The" makes up around 5% of English text on its own. Plot it and you get a cliff, not a bell — which is exactly why tokenizers and vocabulary cutoffs are designed the way they are.

Income is not normal. It is right-skewed with a long tail, which is why the mean income in any city sits well above what most people earn, and why the median is the honest number to quote.

API latency is not normal. Requests cannot take negative time, most are fast, and a few hit a retry or a cold start and take 40 times the median. Server latency, file sizes, city populations, revenue per customer, time between failures — all skewed, all with tails much heavier than a bell allows.

The test is the sentence from earlier: **is this quantity made of many small independent things added together?** Height, yes. Latency, no — it comes from queueing and multiplication, which produce a different shape entirely. When the answer is no, borrowed normal-shaped rules such as "two standard deviations covers 95%" quietly stop being true, and no error tells you.

BEFORE YOU MOVE ON

A team assumes API latency is normal, sets an alert at the mean plus two standard deviations, and expects to page on about 2 to 3% of requests. In production the alert fires on 8%. What is the best explanation?

A. The sample used to compute the mean and standard deviation was too small, so both estimates are off.

B. Latency is not many small independent effects added together — queueing and retries make it right-skewed with a heavy tail, so the two-sigma guide does not hold.

C. Two standard deviations leaves 5% outside, not 2.5%, so the expected rate was wrong from the start.

D. Latency is normal, but its mean drifts through the day, so a fixed threshold misfires.

35 · 10 min

Six distributions, and the process each one describes

THE ONE THING TO KEEP

Each standard distribution corresponds to a specific generating story, so choosing one is a claim about how your data was produced rather than a matter of which curve fits best.

Pick by the process, not by the shape

The usual way distributions are taught is as a gallery of curves. That is backwards. Each one is the answer to a specific question about how something is generated, and if you know the process you know the distribution. Fitting curves by eye and choosing the best-looking one is how people end up modelling counts with a normal and being surprised by negative predictions.

Six cover almost everything you will meet.

Bernoulli: one yes-or-no event

One trial, probability `p` of success. Mean `p`, variance `p(1 − p)`.

Every binary label is a Bernoulli. Every click, conversion, correct answer, spam flag. The variance formula is worth noticing: it is largest at `p = 0.5`, where it is 0.25, and falls to zero at both extremes. A coin is maximally uncertain; an event that almost never happens is nearly predictable.

Binomial: how many successes in n trials

`n` independent Bernoulli trials with the same `p`. Mean `np`, variance `np(1 − p)`.

This is the distribution of your accuracy count on a test set. Score 87 out of 100 and the observed proportion is 0.87 with a standard error of

```
sqrt(0.87 * 0.13 / 100) = 0.034
```

That is 3.4 percentage points, which is why an 87 per cent and an 84 per cent on a 100-item test set are not distinguishable. The binomial is where the honest error bar on every accuracy figure comes from.

Its assumption is that trials are independent and share the same `p`. Test items from the same document break the first; a test set mixing easy and hard cases breaks the second, in the direction of making the true variance larger than the formula says.

Categorical: one draw from k options

The generalisation of Bernoulli to `k` outcomes with probabilities summing to 1. This is precisely what a language model's softmax output is: a categorical distribution over the vocabulary, from which one token is drawn. Nothing more exotic is happening at the sampling step.

Poisson: counts of rare events in a window

Events happening independently at a constant average rate `λ` per interval. Mean `λ`, and — the distinguishing feature — **variance also `λ`**.

Support tickets per hour, errors per thousand requests, arrivals per minute. The mean-equals-variance property gives you a free diagnostic: compute both on your count data. If the variance is much larger than the mean, the events are not independent — they arrive in bursts — and the Poisson assumption is wrong. This is called overdispersion, it is extremely common in real systems, and the standard replacement is the negative binomial.

Ignoring overdispersion produces confidence intervals that are far too narrow, which is how monitoring systems end up with alert thresholds that fire constantly.

Exponential: waiting time between Poisson events

If events are Poisson with rate `λ`, the gap between consecutive events is exponential with mean `1/λ`.

It has one strange and consequential property: it is **memoryless**. If you have waited 10 minutes for the next event, the distribution of the remaining wait is identical to what it was at the start. Nothing accumulates.

This is why the exponential is a reasonable model for the gaps between independent arrivals and a poor model for anything with wear, ageing or a schedule. Time until a machine fails is not memoryless; the machine gets older.

Log-normal: the product of many small effects

If a quantity is the *product* of many independent factors rather than their sum, its logarithm is normal and the quantity itself is log-normal. Right-skewed, always positive, with a long upper tail.

This describes an enormous share of real measurements: incomes, file sizes, city populations, request latencies, session durations, revenue per customer. The normal distribution arises from adding many small effects; the log-nor-

mal arises from multiplying them, and multiplication is at least as common in the world.

The practical signal that you are looking at one: the mean is well above the median, and taking logs makes the histogram symmetric. When that happens, model the log. A regression on log-revenue is usually better behaved in every way than a regression on revenue.

Name the process, and the distribution follows

How the number was generated	The distribution that describes it
One yes-or-no event	Bernoulli: mean p, variance p times one minus p
A count of successes in n independent tries	Binomial: where an accuracy's error bar comes from
One draw from k options	Categorical: a language model's softmax output
Rare events at a steady rate in a fixed window	Poisson: check that the variance equals the mean
The gap between independent events, nothing ageing	Exponential: memoryless, so waiting resets nothing
Many independent factors multiplied together	Log-normal: right-skewed, mean well above median

Choosing a distribution is a claim about how the data were produced, not a question of which curve fits best. When counts show a variance far above their mean, the Poisson story is wrong, and every interval built on it is far too narrow.

Which one, in one line each

- Single yes/no: **Bernoulli**.
- Count of yes out of n: **binomial**.
- One of k options: **categorical**.
- Count of rare events in a window: **Poisson**, and check the variance against the mean.
- Gap between independent events: **exponential**, if nothing ages.
- Positive, right-skewed, produced by multiplication: **log-normal**.
- Sum of many small independent effects: normal, which the earlier lesson covers.

The honest caveat

Every one of these carries assumptions — independence, constant rate, identical distribution — that real data violates. Choosing a distribution is choosing a story, and the useful question is not "which fits best" but "which story is closest to how this actually happens, and how does it fail". Plot your data against the fitted distribution before relying on either. The mismatch in the tail is usually the interesting part, and it is invisible in a summary statistic.

The rule to keep

Name the generating process first, and the distribution follows. If you cannot state the process, you are curve-fitting, and the resulting probabilities in the tail — which is where the decisions are — will be wrong in ways the fit statistic will not show you.

BEFORE YOU MOVE ON

A team models hourly error counts as Poisson to set an alerting threshold. Over 500 hours the mean is 12 and the variance is 140. What does that say, and what happens if they proceed?

A. The data is consistent with Poisson, since 12 and 140 are both plausible values for a count distribution.

B. The errors are overdispersed, so they arrive in bursts rather than independently, and a Poisson threshold will be far too tight and fire constantly.

C. The variance exceeds the mean, so the counts are underdispersed and the threshold will be too loose.

D. The sample of 500 hours is too small to estimate a variance, so the discrepancy is sampling noise.

36 · 8 min

Expectation and variance in plain terms

THE ONE THING TO KEEP

An expectation is a weighted average that may never occur, and its noise falls with the square root of the sample size.

Expectation is a weighted average

Every possible value, multiplied by how likely it is, all added up.

A street vendor in Jakarta sells 40 bowls on a good day (60% of days) and 15 on a bad one (40%).

```
expected sales = 0.6 * 40 + 0.4 * 15 = 24 + 6 = 30 bowls
```

Thirty is the expectation. Now notice what it is not: he never sells 30 bowls. Not once, ever. The expectation is not a value you expect to see. It is what the average settles near over many days.

This trips people constantly. The expected value of a die roll is 3.5. Average household size in a country might be 4.2 people. Neither is a possible outcome. An expectation is a summary of a distribution, and summaries are allowed to be things that never happen.

Every loss you have seen is an expectation

The loss you actually want to minimise is the average over all data your model will ever meet. You cannot compute that — you do not have all the data. You compute it over your training set, which is itself a stand-in. And in practice you compute it over one batch of 32 or 512 examples, which is a stand-in for the stand-in.

So the gradient you step with is an estimate. It is roughly right, and it wobbles. That wobble is not a flaw to be engineered away; it is the "stochastic" in stochastic gradient descent, and a small amount of it helps by shaking the model out of bad spots.

Variance is how much things wobble

Variance measures spread: how far values sit from their mean, on average, with the distances squared so they do not cancel out. Standard deviation is the square root of that, which puts it back into the original units — bowls, rupees, seconds — so it is the one worth reading.

The fact that matters most in machine learning is this one: **when you average n independent things, the spread of that average shrinks by the square root of n.**

Square root, not n. This shows up everywhere and disappoints everyone.

Go from batch size 32 to batch size 128 — four times the compute per step — and the gradient noise does not drop to a quarter. It halves. To halve it again you need batch size 512. The returns are brutal, and this is exactly why very large batches stop helping long before they stop costing.

The number that should change how you read results

You evaluate two models on a 500-question benchmark. Model A scores 71.2%. Model B scores 71.8%. B wins, ship it.

No. Work out how noisy a 500-question score is. For a proportion near 0.7 with n = 500, the standard deviation of the measurement is:

```
sqrt(0.7 * 0.3 / 500) = sqrt(0.00042) ≈ 0.0205
```

About 2 percentage points. Run the same model on a different random 500 questions and you would routinely see 69% or 73%. The 0.6 point gap between A and B is a quarter of one standard deviation. It is noise. You have measured nothing.

To resolve a 0.6 point difference reliably you need on the order of 20,000 questions, or a paired test that compares the two models question by question, which cancels out much of the shared difficulty and is far more efficient. Neither is what most benchmark tables do.

This is why leaderboards where the top eight entries sit within a point of each other are ranking noise, and why a model that gains half a point on a small evaluation has demonstrated nothing at all. Once you know the square root rule, you cannot unsee it — and you will start asking for error bars on results that never had any.

BEFORE YOU MOVE ON

You raise the batch size from 32 to 128, hoping for a less noisy gradient. What happens to the noise?

A. It stays the same. A gradient is an average, and averaging more examples does not change what the average is.

B. It falls to a quarter, since the error shrinks in proportion to the number of examples.

C. It roughly halves, because the spread of an average shrinks with the square root of the count.

D. It depends on the learning rate, since that is what controls how far each step goes.

37 · 9 min

Covariance, and the shape of a cloud of points

THE ONE THING TO KEEP

A covariance matrix describes the shape and orientation of a cloud of data, and its eigenvectors are the directions along which that cloud is longest.

From one variable to two

Variance measures how much one quantity wobbles. Covariance measures whether two wobble together:

```
cov(X, Y) = average of (x − mean_x)(y − mean_y)
```

Positive when they tend to be above or below their means at the same time, negative when one is high while the other is low, near zero when there is no linear pattern.

Its units are the product of the two variables' units, which makes the raw number impossible to interpret — a covariance of 4,200 between height in centimetres and weight in kilograms tells you nothing on its own. Divide by both standard deviations and you get correlation, which is unitless and comparable. Covariance is the quantity the maths uses; correlation is the quantity people read.

The matrix

With `d` features you get a `d × d` matrix: variances down the diagonal, covariances off it. It is symmetric, since `cov(X, Y) = cov(Y, X)`, and that symmetry is what makes everything downstream well behaved.

```
import numpy as np
X = np.random.randn(1000, 3)
X[:, 1] += 0.8 * X[:, 0]              # make features 0 and 1
dependent
C = np.cov(X, rowvar=False)
print(C.round(2))
```

The matrix is a complete description of the shape of the data cloud, provided the cloud is elliptical. It says how wide it is in each feature direction, and how tilted.

Eigenvectors of the covariance are the axes of the cloud

Here is where the linear algebra module pays off. Because the covariance matrix is symmetric, its eigenvectors are perpendicular and its eigenvalues are real and non-negative. Those eigenvectors are the principal axes of the data cloud, and each eigenvalue is the variance along its axis.

So a cloud of points shaped like a tilted cigar has one large eigenvalue, along the length of the cigar, and small ones across it. Sorting the eigenvalues and keeping the largest few is exactly principal component analysis, which gets a worked lesson later. The point here is that PCA is not a separate technique with its own theory. It is reading the eigenvectors of a covariance matrix.

The multivariate normal

The multivariate normal is the natural extension of the bell curve to several dimensions, and it is fully specified by two things: a mean vector, saying where the cloud sits, and a covariance matrix, saying its shape and orientation. Nothing else.

Three cases of the covariance matrix are worth being able to picture:

- **Identity matrix.** A perfectly round cloud, all directions equal, no correlation.
- **Diagonal but unequal.** An axis-aligned ellipse, stretched more in some feature directions than others. This is what "diagonal covariance" means

when you see it as an assumption in a model, and it is a claim that the features are uncorrelated.

- **Full matrix with off-diagonal terms.** A tilted ellipse. The tilt is the correlation.

This is why a full covariance is expensive: it has `d(d+1)/2` free parameters. For 768 dimensions that is 295,296 numbers to estimate, and estimating them well needs far more than 768 data points. That parameter count, not any deep principle, is why so many methods assume a diagonal covariance — it is the only version you can estimate from a realistic sample.

Whitening

If you multiply your data by the inverse square root of the covariance matrix, the result has an identity covariance: round, unit-scale, uncorrelated. This is called whitening, and it is worth knowing because it explains several things at once.

Standardising each column — subtract mean, divide by standard deviation — is the cheap diagonal version of whitening. It fixes the scale differences but leaves the tilt. Full whitening removes the tilt as well, and it is what makes the loss surface of a linear model well conditioned, so gradient descent stops zigzagging. The connection between the optimisation lesson and this one is direct: badly correlated features produce a badly conditioned problem, and whitening is the fix that removes the cause.

The catch is that inverting a covariance matrix estimated from too little data is exactly the ill-conditioned operation from the matrix module. Whitening a 768-dimensional dataset with 500 samples will amplify noise dramatically, because the smallest estimated eigenvalues are essentially random and you are dividing by their square roots.

Where you will actually meet it

- **PCA and dimensionality reduction**, as above.
- **Mahalanobis distance**, which measures distance in units of the data's own spread rather than raw units. It is the sensible distance to use for out-

lier detection when features have different scales and are correlated.

- **Gaussian mixture models**, which fit several multivariate normals and where the choice between full, diagonal and spherical covariance is the main capacity knob.
- **Embedding analysis.** The covariance of an embedding set tells you whether the model is using its dimensions evenly. A spectrum dominated by a few large eigenvalues means the embeddings occupy a narrow cone, which is the anisotropy mentioned in the module on vectors, and it is measured exactly this way.

The rule to keep

The covariance matrix is the shape of the cloud, its eigenvectors are the cloud's axes, and estimating a full one needs far more data than most people have. When in doubt, standardise, plot the first two principal components, and look.

BEFORE YOU MOVE ON

A team wants to whiten 768-dimensional embeddings using a covariance matrix estimated from 400 samples. What goes wrong?

A. Nothing; 400 samples is sufficient because covariance estimation converges quickly with dimension.

B. With fewer samples than dimensions the estimated covariance is singular, and whitening divides by the square roots of eigenvalues that are essentially noise, amplifying it enormously.

C. Whitening requires normally distributed data, and embeddings are not normal, so the operation is undefined.

D. The embeddings must be normalised to unit length first, otherwise the covariance is dominated by magnitude.

38 · 9 min

Heavy tails, and the average that describes nobody

THE ONE THING TO KEEP

When a distribution has a heavy tail the mean is dragged by rare extremes and describes almost no one, so percentiles rather than averages are what you report and act on.

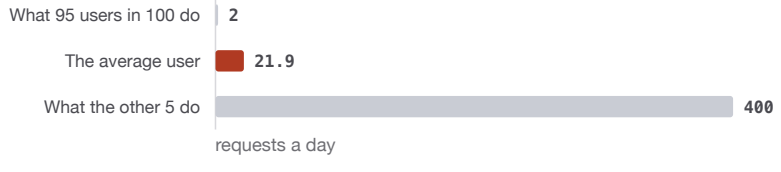
The average user does not exist

Take a hundred users of a service. Ninety-five make 2 requests a day. Five make 400.

```
mean = (95 * 2 + 5 * 400) / 100 = (190 + 2000)/100 = 21.9
median = 2
```

The average user makes 21.9 requests a day. No user makes 21.9 requests a day. Ninety-five per cent make 2 and five per cent make 400, and the mean sits in an empty region between them, describing nobody.

A hundred users of one service



The mean sits in an empty region between the two groups and describes nobody at all. The median is 2. Compare the two before quoting either: a mean more than about one and a half times the median means a few extreme values are carrying the number.

This is not a contrived example. It is the ordinary shape of usage data, spend per customer, file sizes, session lengths, tokens per request and requests per API key. When somebody says "average users do X", the first question is

whether the distribution has a tail, because if it does, the sentence is about a fiction.

What a heavy tail means

A distribution has a heavy tail when extreme values are far more likely than a normal distribution would allow. Under a normal, a value five standard deviations out happens roughly once in 3.5 million. Under a heavy-tailed distribution, it happens often enough to matter every week.

The sharpest version is the power law, where the probability of a value above `x` falls as `x^(−α)`. City sizes, word frequencies, file sizes, wealth and network degrees all approximately follow one. When `α ≤ 2` the variance is infinite — not large, infinite — and when `α ≤ 1` the mean is infinite too. Computing a sample mean from such data gives you a number, and that number does not converge as you collect more data. It keeps drifting upward as bigger examples arrive.

That is worth sitting with. Some quantities have no meaningful average, and the sample average of them is not an estimate of anything.

The latency case, which you will meet

Response times are always right-skewed. A service with a 40 ms mean routinely has a 400 ms 99th percentile, because a small fraction of requests hit a cold cache, a garbage collection pause, or a slow dependency.

The consequence people miss: **a page that makes 20 backend calls will show its user the slowest of the 20.** If each call independently exceeds 400 ms one per cent of the time,

```
P(all 20 fast) = 0.99^20 = 0.818
P(at least one slow) = 18.2%
```

So a "1 per cent tail" at the service level is an 18 per cent tail at the page level. This is why serious latency work targets the 99th and 99.9th percentiles rather

than the mean, and why an average latency in a dashboard is close to useless as a user-experience measure.

Cost, where the same arithmetic bites

Token usage per request is heavy-tailed. Most requests are short; a few paste in an entire document. Budgeting on the mean request size understates cost, because the mean is itself unstable — one very large request moves it, and next month's mean will differ.

The defensible approach is to budget from percentiles and from the total, not from the average: measure total tokens per day directly, and separately cap the per-request maximum. A hard input cap converts an unbounded tail into a bounded one, which is the only thing that makes the cost predictable at all.

How to tell if you have one

Three checks, all cheap:

1. **Compare mean and median.** If the mean is far above the median, the tail is on the right. A ratio above about 1.5 is worth investigating.
2. **Plot the histogram of the logs.** If the log-histogram looks roughly symmetric, you have something log-normal-ish and should work in log space.
3. **Plot the survival function on log-log axes.** Plot the fraction of values above `x` against `x`, both on log scales. A power law appears as a straight line. This is the standard diagnostic, and it takes four lines.

```
import numpy as np
x = np.sort(data)[::-1]
frac = np.arange(1, len(x)+1) / len(x)
# plot log(x) against log(frac); a straight line indicates a
power law
```

Be careful with the third: many distributions look straight-ish over one or two decades, and claims of power laws in the literature have a long history of be-

ing overturned by more careful fitting. Report it as "consistent with" rather than "is".

What to do instead of the mean

- **Report percentiles.** Median, 90th, 99th. Three numbers describe a skewed distribution far better than a mean and a standard deviation, which describe it wrongly.
- **Work in log space.** For log-normal-ish quantities, take logs, do the arithmetic, and transform back. The geometric mean is often the right central summary.
- **Trim or winsorise, and say so.** Dropping the top 1 per cent stabilises an estimate at the cost of ignoring exactly the cases that may matter most. It is a legitimate choice that must be disclosed, because the trimmed mean of a heavy-tailed quantity is a different quantity.
- **Separate the populations.** Sometimes the tail is not a tail but a second group — bots, batch jobs, one enterprise customer. Splitting them gives two distributions that each behave, and is more informative than any robust statistic applied to the mixture.

The rule to keep

Before quoting a mean, compare it with the median. If they differ substantially, the mean is being carried by a few extreme values and every plan built on it — capacity, cost, expected quality — is built on a number that describes none of your cases.

BEFORE YOU MOVE ON

A dashboard shows mean API latency of 45 ms and the team considers performance fine. Users report the product feels slow. Which explanation follows most directly from the distribution's shape?

A. The mean is computed over the wrong time window, and a shorter window would reveal the problem.

B. The latency distribution is right-skewed, so a small fraction of very slow requests barely moves the mean while being exactly what users notice, especially when one page makes many calls.

C. The mean excludes failed requests, so the real average is higher.

D. Users are comparing against competitors rather than an absolute standard, so the measurement is fine.

CASE STUDY · MODULE 4

One thousand seven hundred flags and two radiographers

A municipal screening campaign in Nashik, twenty thousand chest X-rays over ten days, and a vendor's automated triage tool.

The campaign was planned around a mobile van, three technicians and two radiographers. Twenty thousand people would be screened in ten working days. Every image the tool flagged would be read by a radiographer before anybody was contacted.

The vendor's numbers were good and honestly stated. The tool flags 96 per cent of the images a radiologist would call abnormal, and it wrongly flags 8 per cent of the images a radiologist would call normal. Those figures came from a study of thirty thousand images and there was no reason to doubt them.

The health officer running the campaign did the arithmetic before the contract was signed, and she did it by counting a population rather than by using a formula.

In the group being screened, the expected rate of abnormal images was about 0.8 per cent. Out of twenty thousand people that is 160 abnormal images and 19,840 normal ones. The tool would flag 154 of the 160 and miss 6. It would also flag 8 per cent of 19,840, which is 1,587. Total flags: 1,741. Of those, 154 are real. Precision is 8.8 per cent.

Two radiographers reading forty confirmatory images a day each can get through eighty a day, or eight hundred over the campaign. Seventeen hundred flags is twenty-two days of reading for a ten-day campaign. The plan did not fit, and nothing about the vendor's numbers was wrong; the base rate did that.

There were three ways out and each cost something.

Raise the threshold. Tuning the tool so that it wrongly flags 3 per cent rather than 8 brings false alarms down to 595. Recall falls too, to about 88 per cent, so the tool now flags 141 of the 160 and misses 19. Total flags 736, precision 19

per cent, nine days of reading. The plan fits. Thirteen more people are missed by the tool than before.

Add readers. Two more radiographers for ten days would clear seventeen hundred flags. There were not two more radiographers in the district who could be released for ten days, and hiring from outside had a cost the campaign budget did not carry.

Screen fewer people, more selectively. Restricting the campaign to wards with a higher prior rate would raise the base rate and therefore the precision, because precision depends on the ratio of the two groups. Doubling the rate to 1.6 per cent takes precision from 8.8 to about 16 per cent for the same tool. It also means not screening the people who were excluded.

A fourth option was raised and rejected quickly: contact people directly on a flag, with a radiographer reading only the ones who came in. At 8.8 per cent precision that meant telling roughly sixteen hundred people that an automated system had found something in their chest X-ray when it had not. The officer would not do it, and the arithmetic is why: a flag at that precision is a reason to look, not a finding, and an interface or a phone script that presents it as a finding is wrong nine times in ten.

The last complication was that the 0.8 per cent was itself an estimate, taken from the previous year's campaign in a neighbouring district. She ran the numbers again at 0.5 and at 1.5 per cent. At 0.5 the precision falls to 5.7 and the flag count rises; at 1.5 it rises to 15.5. The decision came out the same across that range, which is the useful thing to know about a prior you are not sure of.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

She took the threshold change and wrote the reasoning into the campaign order, in the form of a table: the tool at the higher threshold is expected to flag about 736 images, of which about 141 are real and 595 are not, and to miss about 19. The order also required that the count of flags be checked against 736 at the end of day two, because if the real base rate was far from 0.8 per cent the plan would need to change while there was still time. On day two the count came to 71 flags against an expected 74, so the plan held. Two of the nineteen expected misses were later found by the routine referrals that ran alongside the campaign, which is roughly what the campaign's designers had assumed such referrals would catch.

The vendor's tool catches 96 per cent of abnormal images. Why is only 8.8 per cent of what it flags actually abnormal?

Because the two error rates act on groups of wildly different sizes. There are 160 abnormal images and 19,840 normal ones. A 96 per cent catch rate on the small group gives 154 true flags; an 8 per cent error rate on the large group gives 1,587 false ones. The 8 per cent is applied to a group 124 times larger, so it dominates the total. The catch rate and the precision answer different questions, and the one you act on is the second.

Improving recall from 96 to 99 per cent would help less than halving the false-positive rate. Why?

Recall acts only on the 160 abnormal images: going from 96 to 99 per cent adds five true flags. Halving the false-positive rate from 8 to 4 per cent removes 794 false ones. The denominator of precision is dominated by the false alarms, so the lever that moves precision is the error rate on the large group. This is the general shape of every rare-event deployment, and it is why the false-positive rate is usually the number worth optimising.

She recomputed everything at base rates of 0.5 and 1.5 per cent. What did that test tell her that the single number could not?

It tested whether the decision was robust to the prior she was least sure of. The precision figure moved from 5.7 to 15.5 per cent across that range, which is a large relative change, but the conclusion — that seventeen hundred flags will not fit two radiographers and ten days — held throughout. When a decision is the same across the plausible range of a prior, the prior's uncertainty does not need resolving. When it is not, that is the number to go and measure.

MODULE 4 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A detector catches 95 per cent of fraud and wrongly flags 1 per cent of legitimate transactions. Fraud is 0.1 per cent of all transactions. Roughly what share of its flags are real?

A. About 95 per cent, which is its catch rate on fraud
B. About 9 per cent
C. About 50 per cent, since the two error rates roughly balance
D. About 99 per cent, because the false-positive rate is only 1 per cent

2 A feature-selection step drops every feature whose Pearson correlation with the target is near zero. Which real relationship does it throw away?

A. Any relationship in which the feature has a larger variance than the target
B. Any relationship in which the two variables are genuinely independent
C. Any U-shaped relationship, where the rising and falling halves cancel out
D. Any relationship involving a categorical feature, which Pearson cannot handle

3 You model support tickets per hour as Poisson. The sample mean is 12 and the sample variance is 71. What does that tell you?

A. The Poisson fits well, since both numbers are of the same order
B. The rate is rising, and a trend should be fitted before anything else
C. The sample is too small, and more data will bring the two numbers together
D. Tickets are not arriving independently: they burst, and Poisson intervals will be far too narrow

4 A backend call exceeds 400 ms one time in a hundred. A page makes twenty such calls and the user waits for the slowest. How often is the page slow?

A. One time in a hundred, since the calls are independent of one another

B. One time in two thousand, because the small chances multiply together

C. About eighteen times in a hundred

D. One time in five, because twenty calls at one per cent each is twenty per cent

5 Going from batch size 32 to batch size 128 costs four times the compute per step. By how much does the gradient noise fall?

A. To a quarter, in proportion to the batch size

B. To a half, because the spread of an average falls with the square root of n

C. Not at all: batch size affects speed rather than noise

D. To a sixteenth, because variance is a squared quantity

6 A vendor reports that its model flags 95 per cent of fraudulent transactions. Which quantity is that, and which do you need?

A. It is the probability of fraud given a flag, which is what you need

B. It is the overall accuracy, and precision and recall follow from it directly

C. It is the probability of a flag given fraud; you need the probability of fraud given a flag

D. The two are the same number whenever the classifier is well calibrated

MODULE 5

Logarithms, information and the shape of a loss

Logs and exponentials as tools rather than tables, surprise measured in bits, entropy and cross-entropy computed by hand, why a loss function is a likelihood in disguise, and the softmax arithmetic that every classifier ends with.

BY THE END OF THIS MODULE YOU CAN

Compute a cross-entropy and a KL divergence by hand from two small distributions and say which part of a training loss is irreducible, derive the squared-error and cross-entropy losses from a stated noise assumption, compute a softmax and its gradient for a given set of logits, and explain by the overflow limit of float32 why every library subtracts the maximum before exponentiating

39 · 8 min

What a log is, and why the axis got logged

THE ONE THING TO KEEP

A logarithm counts how many multiplications reach a number, which turns products into sums and equal ratios into equal spacings, and that is why probabilities are added in log space and why any axis spanning several orders of magnitude should be logged.

A log counts multiplications

A logarithm answers one question: how many times must I multiply the base by itself to reach this number? `log2(1024) = 10` because ten doublings reach 1,024. `log10(1,000,000) = 6` because six tens do. That is the whole definition. Everything else is consequence.

Three bases matter in this field.

- **Base 2** counts doublings and gives answers in bits. `log2(32000) = 14.97`, so choosing one token from a 32,000-word vocabulary is about fifteen yes-or-no questions.
- **Base 10** counts orders of magnitude. A learning rate of 0.001 sits at `log10 = −3`.
- **Base e**, written `ln`, where `e = 2.71828`. It is the one calculus prefers, because the slope of `e^x` is `e^x`. Every loss you will see from PyTorch is in this base, in units called nats.

You convert between them by one multiplication: `ln x = 2.303 × log10 x` and `log2 x = 1.443 × ln x`. Remember `ln 2 = 0.693` and `ln 10 = 2.303`; they come up constantly.

The four rules, and the one that matters

```
log(a × b) = log a + log b
log(a / b) = log a − log b
log(a^k)   = k × log a
log_b(x)   = ln x / ln b
```

The first rule is the reason logs exist in machine learning. A model that assigns probability 0.9 to each of 200 tokens in a sentence assigns the sentence `0.9^200 = 7 × 10^−10`. Multiply another hundred such sentences together and the product is below anything a computer can store. Take logs first and the product becomes a sum: `200 × ln 0.9 = −21.07`. Sums of moderate numbers stay moderate. Module 8 returns to exactly how small a number can get before it becomes zero.

Why the axis got logged

Take a learning-rate sweep over 0.00001, 0.0001, 0.001, 0.01 and 0.1. On a linear axis the first four values share the leftmost pixel and the fifth sits alone at the right. On a log axis they are evenly spaced, because equal spacing on a log axis means equal *ratio*, not equal difference. Each grid line is ten times the last.

The same applies to loss curves. A loss that falls from 8 to 4 in the first hundred steps and from 2.1 to 2.0 in the last ten thousand is invisible in its late phase on a linear plot. On a log y-axis the late phase is still readable, and something else appears: a curve that is a straight line on log-log axes is a power law, and one that is straight on a log-linear axis is an exponential. The last lesson of this module teaches you to read the slope. For now, the habit: if the quantity spans more than two orders of magnitude, log the axis.

```
import matplotlib.pyplot as plt
plt.plot(steps, loss)
plt.yscale("log")   # one line; nothing else changes
```

Two things a log axis cannot show

Zero. `log(0)` is minus infinity. A curve that reaches zero loss leaves the bottom of a log plot rather than touching an axis, and a bar chart with a zero bar has no bar. If your data contain zeros, plotting `log(x + 1)` is the usual compromise; say so on the axis label, because it distorts the small values.

Negatives. The log of a negative number is not a real number. Losses are non-negative, so this rarely bites there, but a log axis on a plot of gradients or weight changes silently drops every negative point.

A related trap is the `epsilon` people add to avoid the zero: `log(p + 1e-9)`. It works, but `log(1e-9) = -20.7`, so a probability the model put at exactly zero is scored as a surprise of 20.7 nats rather than infinity. That is a choice, and it changes the average. Pick the epsilon consciously and keep it the same across every run you intend to compare.

Everyday logs you already read

Decibels are `10 × log10` of a power ratio, so 30 dB is a thousandfold. Earthquake magnitude is base 10, so a magnitude 7 releases about 32 times the energy of a 6. pH is a negative log. The screen brightness slider on most phones is logarithmic because perception is. You have been reading log scales for years; the only new thing is doing so on purpose.

Tools

Python gives you `math.log` (base e), `math.log2`, `math.log10` and `math.log(x, base)`. NumPy applies them to whole arrays. A phone calculator has `log` and `ln`. Nothing here needs a GPU or a paid licence.

One exercise, done once, fixes the idea: compute `ln(32000)` on a calculator, divide by `ln 2`, and confirm you get 14.97. Then compute a model's reported loss of 2.0 nats in bits: `2.0 × 1.443 = 2.89`. That model needs just under three bits per token, against fifteen for a uniform guess. The next lessons say what those bits mean.

BEFORE YOU MOVE ON

A training loss is plotted with a logarithmic y-axis and the curve is a straight line sloping downward. What does that shape tell you?

A. The loss is falling by the same amount every step, so it will reach zero where the line meets the axis.

B. The loss shrinks by the same factor every fixed number of steps.

C. The loss follows a power law in the step count.

D. The plot is broken, because a log axis compresses everything into a curve.

40 · 9 min

Exponentials, decay, and the memory of a moving average

THE ONE THING TO KEEP

An exponential moving average with factor β remembers roughly 1/(1−β) steps and weights the past geometrically, which is where Adam's 0.9 and 0.999 come from, why it needs a bias correction at the start, and why a smoothed loss curve shows a spike late and small.

Repeated multiplication has a shape

Multiply 0.9 by itself and watch what happens.

```
0.9^1  = 0.900
0.9^5  = 0.590
0.9^10 = 0.349
0.9^22 = 0.098
0.9^44 = 0.010
```

Every ten steps the value falls by roughly the same factor, about 0.35. That is exponential decay, and it has a **half-life**: the number of steps to halve. For a

factor `β` the half-life is `ln 0.5 / ln β`. For 0.9 that is `−0.693 / −0.105 = 6.6` steps. For 0.99 it is 69 steps; for 0.999 it is 693. The half-life of `β` close to 1 is approximately `0.693 / (1 − β)`, a rule worth memorising.

Growth is the mirror image. `1.01^100 = 2.70`, which is `e` to two decimal places, and this is not a coincidence: `(1 + 1/n)^n` approaches `e` as `n` grows, which is why `e` turns up whenever something compounds continuously.

The exponential moving average

Optimisers, loss smoothers and reinforcement learners all keep a running average that forgets the past gradually rather than all at once:

```
m_t = β × m_(t−1) + (1 − β) × g_t
```

Unroll it once and the structure shows. The newest value `g_t` has weight `(1 − β)`. The one before has weight `(1 − β) × β`, the one before that `(1 − β) × β^2`, and a value `k` steps back has weight `(1 − β) × β^k`. The weights are a decaying exponential, and they sum to 1.

How far back does the average remember? The weights form a geometric series whose mean position is `β / (1 − β)`, and the useful rule is that the **effective window is about `1 / (1 − β)`**. For `β = 0.9` that is ten steps. For `β = 0.999` it is a thousand.

This is where Adam's two constants come from. Its first moment uses `β1 = 0.9`, a ten-step average of the gradient. Its second moment uses `β2 = 0.999`, a thousand-step average of the squared gradient. The second needs a long memory because it is estimating a variance, and variance estimates from ten samples are noisy in a way that would make the step size lurch.

The bias at the start, and the correction

Start the average at `m_0 = 0`. After one step, `m_1 = (1 − β) × g_1`. With `β2 = 0.999` that is `0.001 × g_1`: the average claims the squared gradient is a thousand times smaller than it is. Adam divides by the square root of this

number to set the step, so an uncorrected first step would be about 30 times too large.

The correction divides by the total weight that has actually been applied, `1 − β^t`. At `t = 1` that is `1 − 0.999 = 0.001`, so the correction multiplies by a thousand and cancels the bias exactly. At `t = 1000` it is `1 − 0.999^1000 = 1 − 0.368 = 0.632`, still a noticeable 1.6 times. By `t = 5000` it is 0.993 and has faded. If you have ever wondered why Adam's code has two lines called `bias_correction`, that is the entire reason.

Schedules are exponentials too

An exponential learning-rate schedule multiplies by a factor `γ` every step. Over 50,000 steps at `γ = 0.9999`:

```
0.9999^50000 = e^(50000 × ln 0.9999) = e^(−5.0) = 0.0067
```

So the final learning rate is 0.67 per cent of the initial one. If that seems too aggressive, the arithmetic tells you the fix: `γ = 0.99995` gives `e^(−2.5) = 0.082`. Working in the exponent is easier than guessing at the factor.

The same shape is the discount in reinforcement learning. A discount of 0.99 per step gives rewards a horizon of about a hundred steps; beyond that they contribute less than a third of their face value, and beyond 460 steps less than one per cent.

Where the average misleads

An exponential average lags. With `β = 0.99` a sudden spike in the loss appears in the smoothed curve about a hundred steps late and a hundred times smaller, and a genuine step change is drawn as a gentle slope. The smoothing slider in TensorBoard is exactly this average, and at its default setting it hides the variance that would tell you the learning rate is too high. Look at the raw curve at least once per run.

The start is biased as well. Unless the tool applies the correction above, the first hundred points of a smoothed curve are pulled toward zero, which looks

like an implausibly good early loss. It is an artefact, not a result.

Tools

Everything here is arithmetic you can do on a calculator with `ln` and `e^x`, or in three lines of Python:

```
import math
beta = 0.999
print(1 / (1 - beta), math.log(0.5) / math.log(beta))   # win-
dow, half-life
```

BEFORE YOU MOVE ON

Adam keeps its second-moment estimate with β2 = 0.999. Roughly how far back does that average remember, and how?

A. About a thousand steps, with older gradients fading geometrically.

B. Exactly the last 999 gradients, each weighted equally, like a sliding window.

C. Every gradient since training began, all weighted equally, because β is so close to 1.

D. About ten steps, the same as the first-moment average.

41 · 8 min

Logs, and why losses are logged

THE ONE THING TO KEEP

Logs turn products into sums and make confident mistakes expensive, which is exactly what training needs.

What a log is

A logarithm answers: what power do I raise the base to, to get this number?

```
log10(1000) = 3     because 10^3 = 1000
log10(0.001) = −3   because 10^−3 = 0.001
```

Machine learning mostly uses base `e` ≈ 2.718, written `ln` or just `log` in code. The base changes the scale, not the behaviour.

Two properties are the whole reason logs are everywhere.

Logs turn multiplication into addition. `log(a × b) = log(a) + log(b)`. Products become sums.

Logs stretch out the small numbers. Between 0.1 and 0.01 there is barely any room on a normal ruler. In log terms they are a full unit apart, the same distance as 1 to 0.1.

The problem logs solve first

The probability a model assigns to a whole sentence is the probability of the first token, times the second given the first, times the third, and on. Multiply 500 numbers, each under 1.

```
0.1 ** 500   # 0.0 in float32. Not small. Zero.
```

The true value is around 10^-500, and a 32-bit float bottoms out near 10^-38. Your number is gone, and every comparison after it is meaningless. This is underflow, and it is not a corner case — it happens on every sentence.

Take logs and the product becomes a sum of 500 numbers each around -2.3. Total: about -1150. A float handles that without blinking. This alone would justify logs. The rest is a bonus.

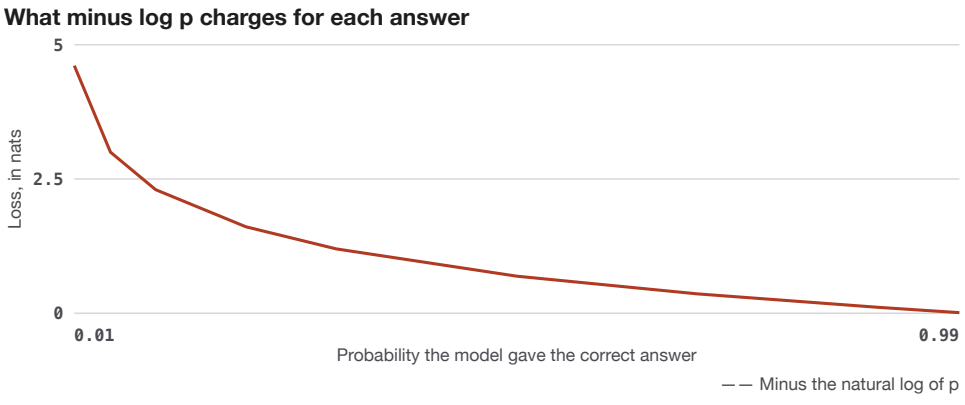
Cross-entropy, read as a shape

The standard loss for classification and language modelling is `-log(p)`, where `p` is the probability the model gave to the correct answer. Negative, because probabilities are below 1 so their logs are negative, and a loss should be positive.

Look at what that function does:

```
p = 0.99   loss = 0.01
p = 0.90   loss = 0.11
p = 0.50   loss = 0.69
p = 0.10   loss = 2.30
p = 0.01   loss = 4.61
p = 0.0001 loss = 9.21
p = 0      loss = infinity
```

Read the top and the bottom. Going from 0.90 to 0.99 saves you 0.10 of loss — a decent model getting slightly better barely moves the needle. Going from 0.01 to 0.10 saves you 2.31, twenty times as much.



Moving from 0.01 to 0.10 saves 2.31 of loss; moving from 0.90 to 0.99 saves 0.10. Cross-entropy spends almost all of its attention on what the model is confidently wrong about, and that priority comes free from the shape of the log rather than from anyone choosing it.

The loss cares enormously about the cases the model is confidently wrong on, and hardly at all about the cases it already has. That is exactly the right priority, and it comes free from the shape of the log. A squared-error loss would treat 0.90 → 0.99 and 0.01 → 0.10 as similar improvements, which is not what you want from a model.

The infinity at zero is real and it matters. Assigning 0 probability to something that then happens is an infinite penalty, which is why models never fully rule anything out, and why a stray `log(0)` shows up as `nan` in your training run.

Reading a loss number

Cross-entropy in nats is hard to feel. Exponentiate it and you get perplexity:

```
loss 2.0  ->  e^2.0 = 7.4
loss 3.0  ->  e^3.0 = 20.1
```

A loss of 2.0 means the model is about as uncertain as someone guessing uniformly among 7.4 options at each token. That is a number you can hold.

It also explains a thing that puzzles people watching loss curves. Dropping from 4.0 to 3.0 looks identical on the plot to dropping from 3.0 to 2.0. In perplexity, the first is 54.6 down to 20.1 and the second is 20.1 down to 7.4. Each unit of loss is the same multiplicative gain, which is why a curve that looks like

it has flattened out near the end is often still delivering real improvements. Log scales compress the top and stretch the bottom. That is their job, and it is why a flat-looking tail is not the same thing as a finished model.

BEFORE YOU MOVE ON

Two language models pick the same top token 30% of the time on the same test set, but one has a cross-entropy of 3.1 and the other 2.4. How can the losses differ that much when accuracy is identical?

A. The lower-loss model hedges. Cross-entropy rewards spreading probability around, while accuracy rewards commitment.

B. Equal accuracy with much lower loss is the signature of memorisation, so the lower-loss model has seen the test set.

C. Loss depends on the probability given to the correct token, not on which token came first, so it can improve greatly while the top choice never changes.

D. One must be reporting in bits and the other in nats, since accuracy already pins down how good a model is.

42 · 9 min

Surprise, bits, and why prediction is compression

THE ONE THING TO KEEP

The surprise of an outcome is –log p, additive across independent events and zero for certainty, and because any distribution can be turned into a code of that length, a model's loss in bits per token is literally the size it would compress the text to.

Measuring surprise

If something you were sure of happens, you learn nothing. If something you thought impossible happens, you learn a great deal. Information theory turns that into a number: the **surprise** of an outcome with probability `p` is

```
surprise = −log2(p)   bits
```

A fair coin landing heads has `p = 0.5`, so `−log2(0.5) = 1` bit. A specific card from a shuffled pack has `p = 1/52`, so 5.7 bits. An outcome with `p = 0.9` is worth `−log2(0.9) = 0.152` bits: mild. An outcome with `p = 1` is worth exactly zero.

Why a log rather than, say, `1/p`? Two reasons that pin it down. Surprise should be additive for independent events: two fair coins should be twice as surprising as one, and `−log2(0.25) = 2` gives exactly that where `1/p` would give 4. And certainty should cost nothing, which the log gives for free. There is a theorem that the log is the only function with these properties, but the additivity is the part to keep.

Bits are yes-or-no questions

A bit is one answer to a well-chosen yes-or-no question. Locating one item among 1,024 equally likely ones takes ten questions, and `log2(1024) = 10`. Locating one token among a 32,000-word vocabulary with no information at all takes `log2(32000) = 14.97` questions.

A language model reduces that count. If it assigns probability 0.25 to the token that actually comes next, that token cost it `-log2(0.25) = 2` bits rather than fifteen. If it assigned 0.001, the token cost 10 bits. The average of those costs over a text is the model's cross-entropy loss in bits, which the next two lessons build up properly.

Nats

PyTorch, TensorFlow and every paper's loss column report surprise in **nats**, using the natural log instead of base 2. One nat is `1 / ln 2 = 1.443` bits. A loss of 2.0 nats per token is 2.89 bits per token. Nothing about the model changes; only the unit does. When somebody quotes a loss without a unit, assume nats.

Prediction is compression

Here is the fact that a product page will not tell you. Any probability distribution over messages can be turned into a code in which a message of probability `p` is written in about `-log2(p)` bits, and no code can do better on average. The technique is called arithmetic coding, and it is not exotic; it is inside every video codec you use.

So a model's loss in bits per token *is* the size it would compress the text to. Take a model scoring 2.0 nats per token on a corpus of one billion tokens:

```
2.0 nats × 1.443 = 2.89 bits per token
2.89 × 1,000,000,000 = 2.89 × 10^9 bits = 361 MB
```

The raw corpus, at roughly four characters per token and one byte per character, is about 4 GB. The model has compressed it eleven to one. For compari-

son, `gzip` gets ordinary English to about 2.5 bits per character, which is roughly 10 bits per token, so this model is compressing three and a half times better than `gzip`. Claude Shannon estimated in 1951 that a fluent human predicts English at about one bit per character, which is roughly four bits per token. A modern large model beats that.

You can measure the `gzip` side yourself in four lines, and it is a good number to have in your head:

```
import zlib
text = open("some_english.txt", "rb").read()
bits_per_char = 8 * len(zlib.compress(text, 9)) / len(text)
print(round(bits_per_char, 2))    # about 2.5 for ordinary
prose
```

Two honest limits

Surprise is about the model, not about the world. A rare typo costs a model many bits because it did not expect it, yet the typo carries almost no meaning. High surprise means "this model predicted poorly here", nothing more profound. Two models can assign different surprises to the same token and both be reasonable.

The compressor must be counted. The 361 MB above only decompresses if the receiver has the model. A seven-billion-parameter model is about 14 GB in half precision, forty times larger than the compressed corpus. As a way of storing one billion tokens, the model is a terrible deal. The claim that stands is narrower and still striking: *per token*, a good predictor and a good compressor are the same object, and improving one improves the other by exactly the same amount.

The arithmetic to carry forward

Surprise is `-log p`. Bits use base 2, nats use base e, and one nat is 1.443 bits. A loss is an average surprise. That average is also a compressed size. Everything in the rest of this module is a consequence of those four sentences.

BEFORE YOU MOVE ON

A language model assigns probability 1/8 to the token that actually comes next. What did that token cost, and why?

A. 8 bits, one for each possible outcome the model was choosing between.

B. 0.125 bits, because the cost is the probability itself.

C. 3 bits, because $-\log_2(1/8) = 3$.

D. It cannot be said without knowing the vocabulary size.

43 · 9 min

Entropy: the average surprise, computed by hand

THE ONE THING TO KEEP

Entropy is the average surprise of a distribution, largest when outcomes are equally likely and zero when one is certain, and the entropy of your label column is the loss of a model that has learned nothing but the base rate, which is the first number to compute.

The definition, and one calculation

Entropy is the average surprise of a distribution: what you expect to pay per outcome, in bits, if outcomes arrive according to that distribution.

```
H(p) = − Σ p_i × log2(p_i)
```

Do one by hand and it stops being a formula. Four outcomes with probabilities 0.5, 0.25, 0.125 and 0.125:

```
0.5   × 1 bit  = 0.500
0.25  × 2 bits = 0.500
0.125 × 3 bits = 0.375
0.125 × 3 bits = 0.375
                 -----
H = 1.75 bits
```

The surprise of each outcome, weighted by how often it happens. Note that the rare outcomes cost three bits each but only occur an eighth of the time, so they contribute less than the common one.

Now the same four outcomes, equally likely. Each costs `log2(4) = 2` bits and `H = 2` bits. That is the maximum: **entropy is largest when every outcome is equally likely**, and then it equals `log2` of the number of outcomes.

And a nearly certain distribution, 0.97, 0.01, 0.01, 0.01:

```
0.97 × 0.044 = 0.043
0.01 × 6.644 = 0.066   (three times)
H = 0.043 + 0.199 = 0.242 bits
```

Rare outcomes are very surprising when they happen, 6.6 bits each, but they almost never happen, so the average is small. Entropy is zero only when one outcome has probability 1.

Effective number of choices

Two to the power of the entropy is the **effective number of equally likely outcomes**. The 1.75-bit distribution above behaves like `2^1.75 = 3.36` equally likely choices; the 0.242-bit one like `2^0.242 = 1.18`. This number has a name when the distribution is a model's prediction of the next token: perplexity. The course `how-llms-work` treats what it means for a language model; here the point is only that it is `2^H`, or `e^H` if `H` is in nats, and nothing more mysterious.

The number to compute before training anything

Suppose a binary label is 1 in ten per cent of rows and 0 in ninety. A model that knows nothing but that base rate predicts 0.1 every time, and its average loss is the entropy of the label:

```
H = −(0.1 × log2 0.1 + 0.9 × log2 0.9)
  = −(0.1 × −3.322 + 0.9 × −0.152)
  = 0.332 + 0.137 = 0.469 bits = 0.325 nats
```

That is the loss of a model that has learned nothing except the frequency. Any model reporting a cross-entropy above 0.325 nats on that data is worse than a constant. People routinely train for hours, see a loss of 0.40, and call it progress, because they never computed the number the constant predictor gets. For a balanced binary label the floor is `ln 2 = 0.693` nats; for ten balanced classes it is `ln 10 = 2.303`. Compute it, write it on the plot, and judge the curve against it.

Where else entropy appears

- **A model's own uncertainty.** The entropy of the next-token distribution says how decided the model is at that position. Low entropy on a factual token and high entropy on a name it is inventing is a pattern worth looking for.
- **Choosing a split.** Decision trees pick the feature whose split most reduces the entropy of the labels; `machine-learning-foundations` works through it.
- **How peaked a softmax is.** The entropy of an attention row tells you whether a head is looking at one token or spreading over many.

What entropy is blind to

Entropy sees only the list of probabilities, not what they are attached to. A distribution split evenly between "cat" and "kitten" has exactly one bit of entropy, and so does one split between "cat" and "lorry", though the second is far more uncertain in any sense that matters. Entropy has no notion of two outcomes

being close. If nearness matters, you need a metric on the outcomes as well, which is a different tool.

A second caution concerns continuous quantities. The entropy of a density, called differential entropy, can be negative and changes when you change units; a Gaussian with standard deviation `σ` has entropy `½ log2(2πe σ²)`, which is below zero once `σ < 0.24`. It is still useful for comparing two densities in the same units, but the intuition "entropy is a count of bits" holds only for discrete outcomes.

Doing it in code

```
import numpy as np
p = np.array([0.5, 0.25, 0.125, 0.125])
H = -(p * np.log2(p)).sum()      # 1.75
```

With zeros in `p` this gives a `nan` from `0 × log 0`; mask them out, because the correct contribution of an impossible outcome is zero. That single line, applied to your label column, is the baseline that the next lesson turns into a floor no model can go beneath.

BEFORE YOU MOVE ON

A binary label is positive in 10 per cent of rows. After an hour of training a model reports a cross-entropy of 0.40 nats. What does that number say?

A. It is a reasonable result, because 0.40 is well below the 0.693 nats of a coin flip.

B. It is worse than predicting the base rate, which scores 0.325 nats.

C. It is a good result, because any loss under 1 nat means the model is mostly right.

D. Nothing can be concluded without the accuracy figure as well.

44 · 10 min

Cross-entropy and KL: the cost of believing the wrong distribution

THE ONE THING TO KEEP

Cross-entropy is entropy plus KL divergence, so minimising it against fixed data can only shrink the KL and never go below the data's own entropy, which is why a loss plateau above zero can be the floor rather than a failure.

Paying with the wrong code

Entropy is what you pay when your predictions match reality. Cross-entropy is what you pay when they do not. If outcomes really arrive with probabilities `p` but you built your code, or your model, around probabilities `q`, the average cost is

```
H(p, q) = − Σ p_i × log2(q_i)
```

The probabilities that decide how often you pay are the true ones, `p`. The probabilities that decide how much you pay each time are yours, `q`.

Take the reality from the last lesson, `p = (0.5, 0.25, 0.125, 0.125)`, whose entropy is 1.75 bits. Suppose your model believes the four outcomes are equally likely, `q = (0.25, 0.25, 0.25, 0.25)`. Every outcome now costs `−log2(0.25) = 2` bits, so `H(p, q) = 2` bits. You pay a quarter of a bit more per outcome than a perfect model would.

Now a model that is wrong in a different way, `q = (0.7, 0.1, 0.1, 0.1)`:

```
0.5   × −log2(0.7) = 0.5   × 0.515 = 0.257
0.25  × −log2(0.1) = 0.25  × 3.322 = 0.830
0.125 × −log2(0.1) = 0.125 × 3.322 = 0.415
0.125 × −log2(0.1) = 0.125 × 3.322 = 0.415
                                     ─────
H(p, q) = 1.918 bits
```

This model is overconfident in the first outcome and pays for it on the second, which it thought was rare and which happens a quarter of the time.

The gap has a name

The extra cost over the entropy is the **Kullback-Leibler divergence**:

```
KL(p ‖ q) = H(p, q) − H(p) = Σ p_i × log2(p_i / q_i)
```

For the uniform model it is `2 − 1.75 = 0.25` bits; for the overconfident one, `1.918 − 1.75 = 0.168` bits. KL is never negative, and it is zero only when `q` equals `p` exactly. It measures, in bits per outcome, how wrong a belief is.

Where a cross-entropy of 2.00 bits actually goes

Cross-entropy, 2.00 bits	what a model believing q pays per outcome
KL divergence, 0.25 bits	the removable part: how wrong the belief is
Entropy of the data, 1.75 bits	the floor, which no model can go beneath

Cross-entropy is entropy plus KL divergence. Training against fixed data can only shrink the second term, so a loss that plateaus above zero may be sitting on the floor rather than failing. Compute the floor from your own label column before judging any curve.

That identity, `H(p, q) = H(p) + KL(p ‖ q)`, is the most useful line in this module. When you train a model by minimising cross-entropy against data, `H(p)` is a property of the data and does not move. So minimising cross-entropy is minimising KL, and the loss can never go below the data's own entropy. A language model's loss plateaus above zero not because it has failed but because language is genuinely uncertain: the floor is real. The previous lesson computed that floor for a label column; the same logic applies to every token.

With one-hot labels

In classification the "true" distribution for a single example is a one-hot: probability 1 on the correct class, 0 elsewhere. Every term in the sum vanishes except one, and the cross-entropy collapses to `-log q(correct)`: the surprise of the right answer. That is why the loss you see in a training loop is simply minus the log of the probability the model gave the label. The elaborate definition and the simple one are the same thing.

KL is not symmetric

Swap the roles and you get a different number. With `p = (0.5, 0.25, 0.125, 0.125)` and `q` uniform:

```
KL(p ‖ q) = 0.25 bits
KL(q ‖ p) = 0.25 × (log2(0.25/0.5) + log2(0.25/0.25) + 2 ×
log2(0.25/0.125))
          = 0.25 × (−1 + 0 + 2) = 0.25 bits
```

Equal in this case, by coincidence of the round numbers. Change `p` to `(0.7, 0.1, 0.1, 0.1)` and `KL(p ‖ uniform) = 0.53` while `KL(uniform ‖ p) = 0.64`. The direction matters, and it matters most when one distribution puts probability near zero where the other does not.

The practical consequence: if `q` assigns probability 0 to something `p` can produce, `KL(p ‖ q)` is infinite, because you would pay `-log 0` when it happened. A model trained against exact one-hot targets is therefore pushed to make the correct logit infinitely larger than the rest, and never gets there. Label smoothing, which replaces the target `(1, 0, 0, 0)` with something like `(0.97, 0.01, 0.01, 0.01)`, exists to give the model a finite place to stop.

The direction also changes behaviour. Minimising `KL(p ‖ q)` over `q` makes `q` cover everywhere `p` has mass, even if it must spread thin. Minimising `KL(q ‖ p)` makes `q` sit on one mode of `p` and ignore the others. When a preference-trained language model is penalised for drifting from its reference

model, the penalty is the second kind, taken over the model's own samples, which is one reason such models become narrower rather than broader.

Two things KL is not

It is not a distance: it is asymmetric and does not satisfy the triangle inequality, so "the KL between two embeddings" is not a meaningful phrase unless both are distributions. And it is not bounded: a single near-impossible event can make it arbitrarily large, so an average KL across a dataset can be dominated by one row.

The trap in the library

`torch.nn.functional.kl_div(input, target)` expects `input` to be **log** probabilities and `target` to be plain probabilities, and it computes `KL(target ‖ exp(input))`. Passing probabilities for both runs without error and returns nonsense. `scipy.special.rel_entr(p, q)` takes plain probabilities for both and returns the elementwise terms; sum them yourself. Compute one small example by hand, as above, and check the library agrees before trusting it on anything larger.

BEFORE YOU MOVE ON

A classifier's cross-entropy falls to 0.90 nats and stays there, with a flat rather than rising curve, while the labels' own entropy is estimated at 0.85 nats. What is going on?

A. The model is underfitting and needs more layers to push the loss toward zero.

B. The learning rate is too high and the optimiser is bouncing around the minimum.

C. The two figures cannot be compared, because entropy and cross-entropy use different units.

D. The model is within 0.05 nats of the best possible loss.

45 · 10 min

Loss functions are not arbitrary: maximum likelihood

THE ONE THING TO KEEP

Every standard loss is minus the log-likelihood under a stated noise assumption, so squared error means Gaussian errors of constant variance, absolute error means Laplace, cross-entropy means Bernoulli or categorical, and a loss that fits badly is a noise assumption that was wrong.

Why squared error, and not cubed

Nobody explains where the loss functions come from. Squared error for regression, cross-entropy for classification: they are presented as conventions, and a learner reasonably wonders why not absolute error, or the fourth power, or something else. There is an answer, and it is the same answer for every loss you will meet.

The principle is **maximum likelihood**. State how you believe the data were generated, as a probability distribution with some unknown parameters. Then choose the parameters that make the data you actually observed as probable as possible. The loss is minus the log of that probability. Every standard loss is this recipe applied to a particular assumption about the noise.

Squared error is a Gaussian assumption

Suppose you believe each target is the model's prediction plus Gaussian noise of fixed spread `σ`:

$$y = f(x) + \text{noise}, \quad \text{noise} \sim \text{Normal}(0, \sigma^2)$$

The probability density of observing `y` is then

```
p(y | x) = (1 / (σ√(2π))) × exp(−(y − f(x))² / (2σ²))
```

Take the log:

```
log p(y | x) = −(y − f(x))² / (2σ²) − log(σ√(2π))
```

The second term does not depend on the model. Maximising the log-likelihood over many independent examples means maximising the sum of the first terms, which means **minimising the sum of squared errors**. Mean squared error is not a convention. It is the statement "I believe the errors are Gaussian with constant variance", and the squaring is the shape of the Gaussian's exponent.

Do it with numbers once. Three points with targets 2.0, 4.1 and 5.9, and two candidate models. Model A predicts 2, 4, 6; model B predicts 2.5, 4.5, 5.5. With `σ = 1`:

```
A: squared errors 0.00 + 0.01 + 0.01 = 0.02  →  log-lik = −0.01
+ const
B: squared errors 0.25 + 0.16 + 0.16 = 0.57  →  log-lik =
−0.285 + const
```

A is `e^(0.275) = 1.32` times as likely to have produced the data. Smaller squared error and larger likelihood are the same fact.

Change the assumption, change the loss

- **Laplace noise**, whose density is `exp(−|y − f| / b)`, gives **absolute error**. The Laplace has fatter tails than the Gaussian, so a large error is merely unlikely rather than astronomically so, and the fit is not dragged toward it. This is why L1 loss is robust to outliers: it is the loss that expected them.
- **Bernoulli outcomes**, a label that is 1 with probability `q` and 0 otherwise, give `−(y log q + (1 − y) log(1 − q))`, which is **binary cross-entropy**.

- **Categorical outcomes** give `-log q(correct class)`, the cross-entropy of the previous lesson.
- **Poisson counts**, for something like the number of clicks in an hour, give `f - y log f`, the Poisson loss, and the model predicts a log-rate.

The method also explains a mistake you will see: training a house-price model with squared error when a few mansions are in the data. The Gaussian assumption says an error ten times the typical one has probability `e^(-50)`, so the fit distorts itself to avoid it. If your errors are not Gaussian, the loss has been told a falsehood, and it will act on it.

One gradient for all of them

A striking consequence. For every loss in the list above, the derivative of the loss with respect to the model's raw output is the same expression:

```
prediction - target
```

For squared error it is `f(x) - y` directly. For binary cross-entropy through a sigmoid it is `q - y`. For softmax cross-entropy it is `p_i - y_i`, which the lesson on softmax computes explicitly. Linear regression, logistic regression and a softmax classifier are the same algorithm with three different noise assumptions, and their training loops differ by one line.

What maximum likelihood gets wrong

It believes the data too much. Observe an event three times out of three and maximum likelihood says its probability is exactly 1: the value that makes three-for-three most probable is certainty. A model with many parameters and few examples will, in the same way, choose parameters that make the training set look inevitable, which is the mechanism behind overfitting. The fix is to add a prior, a statement of what parameters were plausible before seeing data, and that turns out to be what regularisation is. The next module derives it.

It also assumes the examples are independent. Two hundred rows from the same customer are not two hundred pieces of evidence, but the product of their likelihoods treats them as such.

The variance you dropped

In the Gaussian derivation, `σ` was fixed and fell out of the loss. If instead you let the model predict its own `σ(x)` for each input, the `log σ` term stays, and the loss becomes

```
(y − f(x))² / (2σ(x)²) + log σ(x)
```

The model can now admit that some inputs are harder to predict, by widening `σ` there, and it is charged `log σ` for doing so, which stops it widening everywhere. That is heteroscedastic regression, and it is one line of extra algebra away from the ordinary kind. Once you see losses as log-likelihoods, extensions like this stop looking like tricks.

BEFORE YOU MOVE ON

A house-price regressor trained with mean squared error is being pulled badly by a few mansions. In the language of likelihood, which assumption has been violated?

A. That house prices themselves follow a normal distribution across the dataset.

B. That the relationship between features and price is linear.

C. That the errors are Gaussian with a constant spread.

D. That each house is an independent sample, which mansions in one suburb are not.

46 · 9 min

Softmax, its gradient, and the shift it cannot see

THE ONE THING TO KEEP

Softmax exponentiates and normalises, so only logit differences matter and a shared shift is invisible, and with cross-entropy its gradient is simply probability minus target, which is why training pushes hard on confident mistakes and barely at all on what is already right.

Turning scores into probabilities

A classifier's last layer produces one number per class. They can be negative, they need not sum to anything, and they are called logits. Softmax turns them into probabilities:

```
softmax(z)_i = e^(z_i) / Σ_j e^(z_j)
```

Compute one. Logits `z = (2, 1, 0)`:

```
e^2 = 7.389,  e^1 = 2.718,  e^0 = 1.000,  sum = 11.107
p = (7.389, 2.718, 1.000) / 11.107 = (0.665, 0.245, 0.090)
```

Three properties follow from the exponential. Every output is positive, because `e^x` is. The outputs sum to 1, by construction. And larger logits give larger probabilities, monotonically.

Logits are log-probabilities up to a constant

Take the log of the softmax:

```
log p_i = z_i − log Σ_j e^(z_j)
```

The second term is the same for every class. So the logit *is* the log-probability, shifted by a shared constant. Differences of logits are therefore logs of probability ratios: `z_1 − z_2 = 1` means `p_1 / p_2 = e = 2.718`, and in the example `p_1 / p_2 = 0.665 / 0.245 = 2.72`. A logit gap of 4.6 means one class is a hundred times more probable than the other. This is the fastest way to read a row of logits.

The shift it cannot see

Add the same number to every logit and nothing changes. `(102, 101, 100)` gives exactly the same probabilities as `(2, 1, 0)`, because `e^(z+c) = e^c × e^z` and the `e^c` cancels between numerator and denominator.

Two consequences. First, a model's logits have no absolute meaning; only their differences do, so a logit of 12 is not "confident" unless you know the others. Second, you may subtract anything you like before exponentiating, and the next lesson shows why every library subtracts the maximum.

Temperature

Dividing the logits by a temperature `T` before softmax is the same operation as scaling the gaps. With `T = 2` the example becomes `(1, 0.5, 0)` and the output `(0.506, 0.307, 0.186)`: flatter. With `T = 0.5` it becomes `(4, 2, 0)` and the output `(0.867, 0.117, 0.016)`: sharper. As `T` goes to zero the largest logit takes everything; as it goes to infinity the output goes uniform. The course `how-llms-work` covers what that does to generated text. The arithmetic is only this: temperature rescales the logit differences, and a difference of `d` at temperature `T` is a probability ratio of `e^(d/T)`.

The gradient, and why it is worth seeing

Pair softmax with cross-entropy, so the loss is `-log p_correct`, and differentiate with respect to the logits. Several cancellations happen and the result is remarkably clean:

```
∂L / ∂z_i = p_i − y_i
```

where `y` is the one-hot target. For the example, with the correct class first:

```
gradient = (0.665 − 1, 0.245 − 0, 0.090 − 0) = (−0.335, 0.245,
0.090)
```

The correct logit is pushed up by 0.335, the others are pushed down by their own probability, and the three numbers sum to zero, as they always do. The size of the push is the size of the mistake. If the model had assigned 0.01 to the correct class the push would be 0.99, nearly the maximum; if it had assigned 0.999 the push would be 0.001. **Cross-entropy trains hard on what it gets wrong and barely at all on what it already knows.** That is the mechanism behind the shape of most loss curves: fast early, when everything is wrong, and slow late, when the remaining errors are few and the gradient on each is small.

It is also why the derivative has the same form as in linear regression, `prediction − target`, as the previous lesson promised. The exponential in the softmax and the log in the loss undo each other exactly.

Where it fails

Saturation. When one logit exceeds the rest by 20, its probability is `1 − 2 × 10^−9`. If that class is wrong, the gradient on the correct logit is essentially 1 and on the wrong one essentially −1, which sounds healthy, but the *input* to the softmax is far from any region where small changes matter, and it can take many steps to climb back. Confident mistakes are slow to unlearn. Label smoothing and clipping of logits both exist to keep the model out of that region.

Forced choice. Softmax insists the classes are exclusive and exhaustive. Add an irrelevant fourth class and every other probability shrinks to make room. If an example can belong to several classes at once, or to none, the right tool is one sigmoid per class, not a softmax.

Five lines of NumPy

```
import numpy as np
def softmax(z):
    z = z - z.max()              # the shift it cannot see,
used on purpose
    e = np.exp(z)
    return e / e.sum()
p = softmax(np.array([2.0, 1.0, 0.0]))
grad = p - np.array([1, 0, 0])   # (-0.335, 0.245, 0.090)
```

Run it once, change a logit, and watch the gradient move. The whole of classification training is that last line, repeated.

BEFORE YOU MOVE ON

Two logits are 3.0 and 1.0, with the rest far below. The model then adds 50 to every logit. What happens to the probability of the first class?

A. Nothing: only differences between logits matter.

B. It rises to nearly 1, because a logit of 53 is enormous.

C. It becomes NaN, because exponentiating 53 overflows.

D. It falls, because the shift dilutes every class equally.

47 · 9 min

The log-sum-exp trick, and why exp(1000) is not a number

THE ONE THING TO KEEP

Because float32 overflows at exp(88.7), softmax is computed by subtracting the maximum logit first, and log-probabilities are computed as z minus log-sum-exp rather than as the log of a softmax, which is why a loss function takes raw logits and why passing it probabilities trains toward a ceiling.

The overflow that produces NaN

A float32 number cannot exceed about `3.4 × 10^38`. The exponential passes that at `x = 88.7`, so `exp(89)` in single precision is not a large number: it is `inf`. Logits of 89 are not exotic; an untrained network with a bad initialisation or a training run whose learning rate is too high produces them routinely.

Apply the naive softmax formula to `z = (1000, 999)`. Both exponentials are `inf`. The sum is `inf`. Each ratio is `inf / inf`, which the floating-point standard defines as `NaN`. The answer should have been `(0.731, 0.269)`, since the logits differ by exactly 1 and `e / (e + 1) = 0.731`. The arithmetic was easy; the intermediate values were not representable.

The fix is the shift invariance

The previous lesson showed that subtracting any constant from every logit leaves the softmax unchanged. Subtract the maximum:

```
z − max(z) = (0, −1)
e^0 = 1,  e^(−1) = 0.368,  sum = 1.368
p = (0.731, 0.269)
```

After the shift the largest exponent is exactly 0, so the largest exponential is exactly 1, and nothing can overflow. The smaller logits become negative, and very negative ones underflow to zero, which is harmless: a class with probability `10^-50` contributes nothing to the sum, and the answer is unchanged to every digit float32 can hold.

Softmax of the logits 1000 and 999, computed two ways

Straight from the formula	Subtract the largest logit first
• e to the 1000 becomes inf	• the logits become 0 and minus 1
• e to the 999 becomes inf	• e to the 0 is 1, e to the minus 1 is 0.368
• their sum is inf	• their sum is 1.368
• inf divided by inf is NaN	• the answer is 0.731 and 0.269
• one NaN spreads to every weight it touches	• exact to every digit float32 can hold

Float32 overflows at exp of 88.7, so a logit of 89 is already infinite. Subtracting the largest logit changes nothing about the answer, because softmax cannot see a shift shared by every logit, and it is the difference between a number and NaN.

This is not an optimisation. It is the difference between an answer and `NaN`, and every deep-learning library does it silently inside `softmax`.

Log-sum-exp

The same idea gives a stable way to compute `log Σ e^(z_i)`, which appears in every log-probability, every normalising constant and every loss:

```
LSE(z) = m + log Σ_i e^(z_i − m),   where m = max(z)
```

For `z = (1000, 999)`: `m = 1000`, the shifted sum is `1 + 0.368 = 1.368`, its log is `0.313`, and `LSE = 1000.313`. No overflow, and the result is exact to float32 precision. The unstabilised version would have returned `inf`.

Log-softmax follows immediately:

```
log p_i = z_i − LSE(z)
```

Compute log-probabilities this way, never as `log(softmax(z))`. In the second form, a class whose exponential underflowed to zero produces `log(0) =`

`-inf`, and a single `-inf` in a loss makes the gradient `NaN` for the whole batch. The first form gives `1000 - 1000.313 = -0.313` for the top class and a large finite negative number for the others, which is what you wanted.

The bug this explains

`torch.nn.CrossEntropyLoss` takes **raw logits**. Internally it computes log-softmax with the trick above and picks out the correct class, in one fused, stable operation. A common mistake is to apply `softmax` first and pass the probabilities in:

```
probs = torch.softmax(logits, dim=-1)
loss = F.cross_entropy(probs, targets)    # wrong, and it runs
```

Nothing errors. But the loss function now treats the probabilities, all between 0 and 1, as logits, and applies a second softmax to them. Logits that differ by at most 1 give probabilities that are nearly uniform: three classes with input `(0.9, 0.05, 0.05)` come out as `(0.53, 0.23, 0.23)`. The model trains, slowly, toward a ceiling it cannot pass, and the loss never goes far below `ln(number of classes)`. If a classifier learns but its loss stalls near that value, look for a softmax that should not be there.

Adding probabilities in log space

Sequence models, beam search and hidden Markov models all need `log(p_1 + p_2)` when they hold only `log p_1` and `log p_2`, and the probabilities themselves are too small to form. The answer is log-sum-exp of the two logs: `LSE(log p_1, log p_2)`. For `log p_1 = -700` and `log p_2 = -701`, the direct route underflows both to zero and returns `log 0`; the trick returns `-700 + log(1 + e^(-1)) = -699.69`. This is the single identity that makes probabilistic computation over long sequences possible at all.

What the trick does not fix

Subtracting the maximum prevents overflow. It does not add precision. Float32 carries about seven significant digits, so a term whose shifted expo-

nential is below `10^-7` relative to the largest is lost from the sum entirely. That happens once a logit trails the maximum by more than about 16. The effect on the top class is nil, but the gradient with respect to a logit that far behind is computed as exactly zero rather than a tiny positive number. Usually harmless; occasionally the reason a rare class never moves. Module 8 explains where the seven digits come from.

In code

```
import numpy as np
from scipy.special import logsumexp   # does the shift for you
z = np.array([1000.0, 999.0])
log_p = z - logsumexp(z)              # (-0.313, -1.313)
p = np.exp(log_p)                     # (0.731, 0.269)
```

Write the naive version once, feed it `(1000, 999)`, and see the `NaN` with your own eyes. After that you will never wonder why the library code has that extra line.

BEFORE YOU MOVE ON

A learner writes `probs = softmax(logits)` and then `loss = F.cross_entropy(probs, y)`. Training runs without error but the loss will not go far below ln(number of classes). Why?

A. cross_entropy expects log-probabilities, so it should have been given log(probs) instead.

B. softmax should be replaced by a sigmoid, which is what cross-entropy is built for.

C. The learning rate is too low for the probabilities to move.

D. The loss softmaxed the probabilities a second time, flattening every output.

48 · 9 min

Power laws, and the straight line on a log-log plot

THE ONE THING TO KEEP

A power law is a straight line on log-log axes whose slope is the exponent, so a slope of −0.5 means quadrupling the input halves the output, and a straight line over two decades is weak evidence for extrapolating that promise further.

Two shapes that look alike and are not

An exponential, `y = c × b^x`, grows or decays by a fixed factor for each fixed *step* in `x`. A power law, `y = c × x^k`, changes by a fixed factor for each fixed *ratio* in `x`: doubling `x` always multiplies `y` by `2^k`, whether `x` goes from 1 to 2 or from a million to two million. Both curve on ordinary axes, and both are easily mistaken for each other by eye.

Logs separate them. Take logs of the power law:

```
log y = log c + k × log x
```

That is a straight line in `log x` with slope `k`. So a power law is a straight line on **log-log** axes, and the exponent is the slope you can read off with a ruler. An exponential is a straight line on **log-linear** axes, y logged and x not. Plot the data both ways; whichever is straight tells you which law you have.

Reading the slope

Two points on a log-log plot, `(10, 500)` and `(1000, 5)`:

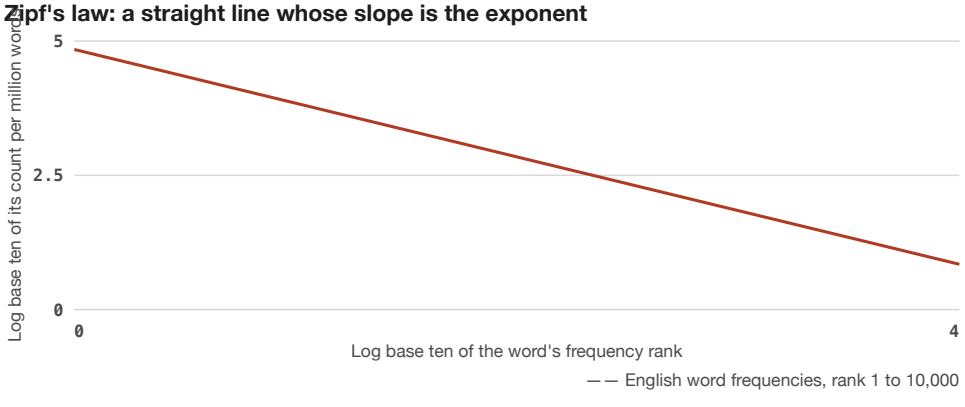
```
k = (log 5 − log 500) / (log 1000 − log 10)
  = (0.699 − 2.699) / (3 − 1)
  = −2 / 2 = −1
```

So `y ∝ 1/x`. The base of the log does not matter, since it cancels in the ratio. Any two points far enough apart give the slope; use points at least a decade apart, because near points make the slope hostage to noise.

A slope is a promise about doubling. A slope of `-0.5` says that quadrupling `x` halves `y`, since `4^(-0.5) = 0.5`. A slope of `-0.05` says doubling `x` cuts `y` by `2^(-0.05)`, a factor of 0.966: three and a half per cent. Small exponents are the mathematical form of diminishing returns, and they are what the loss-versus-compute curves in `how-llms-work` look like. Read the slope before you read the headline.

Zipf, and why the vocabulary has a tail

Rank the words of a large English text by frequency. The most common, "the", is about 7 per cent of all words. The second is about half that, the tenth about a tenth, the hundredth about a hundredth. Frequency is roughly proportional to `1 / rank`, a power law with exponent `-1`, and it holds across languages and across centuries of text. This is Zipf's law.



A slope of minus one means frequency is proportional to one over rank, so the hundredth commonest word appears about a hundredth as often as the first. Read the slope with a ruler, then ask what the line is being asked to promise beyond the points it was drawn through.

It has consequences you have met. A tokeniser gives short, single tokens to common words because they pay for themselves, and splits rare words into pieces because a vocabulary slot for each would be wasted; that is the head of the distribution being served and the tail being approximated. It also means the tail is enormous: with exponent `-1`, half of all the *distinct* words in a cor-

pus appear once, and no amount of data gets the model many examples of any one of them. Rare-word failures are not a bug in a particular model. They are the arithmetic of the distribution the model was trained on.

Power laws and heavy tails

The heavy-tailed distributions of module 4 are usually power laws in their tails. A Pareto distribution with tail exponent `α` has infinite variance when `α ≤ 2` and no finite mean at all when `α ≤ 1`. City sizes, wealth, file sizes, the number of followers an account has, and the length of the longest request in a day all sit in this family, which is why their averages misbehave and their percentiles do not. When you see a straight line on a log-log histogram, expect the mean to be unreliable, and expect the largest observation to grow as you collect more data rather than settling down.

Where the straight line lies

A straight line over one or two decades is weak evidence of a power law. A log-normal distribution looks straight on log-log axes over a limited range; so do several others. The distinguishing region is the far tail, which is exactly where you have the fewest points. Three specific cautions:

- **Do not fit the exponent by least squares on a log-log histogram.** Binning and logging make the noise unequal across the line, and the fit is biased. The maximum-likelihood estimate for a tail exponent from `n` values above a threshold `x_min` is one formula: `α = 1 + n / Σ ln(x_i / x_min)`. It takes one line and it is what the `powerlaw` package computes.
- **Do not extrapolate the line.** A slope measured between `10^3` and `10^5` says nothing about `10^7`. The curves you will most want to extrapolate, of loss against scale, are the ones with the fewest points at the far end.
- **Check the exponent's uncertainty.** With twenty points, the slope carries an error bar of several tenths, and a slope of `−1.1` against `−0.9` is not a distinction the data can make.

In code

```
import numpy as np, matplotlib.pyplot as plt
plt.loglog(x, y, "o")                        # straight means
power law
k, logc = np.polyfit(np.log(x), np.log(y), 1) # slope k; fine
for a curve, not a histogram
```

The `polyfit` on logs is acceptable for a smooth curve such as a loss against dataset size, where each point is a measurement rather than a count. For a histogram of a heavy-tailed quantity, use the likelihood formula. Both are free, and both fit on a laptop. The skill being trained is the one you use every time a paper shows a straight line on log axes: find the slope, find its uncertainty, and ask what the line is being asked to promise beyond the data it was drawn through.

BEFORE YOU MOVE ON

Error against dataset size is a straight line on log-log axes with slope −0.5. What does going from 10,000 to 40,000 examples buy?

A. Error halves.

B. Error falls by 0.5, since that is the slope.

C. Error falls to a quarter, because the dataset grew four times.

D. It cannot be said without the line's intercept.

CASE STUDY · MODULE 5

A loss of 2.34, and the floor nobody had computed

An electronics retailer in Hyderabad training a classifier to route customer support tickets into eleven categories, with a board demonstration on Friday.

The support team handled about four hundred tickets a day in a mixture of English, Hindi and Telugu, and routed them by hand into eleven queues. Routing took roughly three people-hours a day, and the point of the model was to give those three hours back.

An engineer trained a classifier on 38,000 labelled tickets from the previous eighteen months. It trained without any error. The loss fell from 2.44 to 2.34 over the first epoch and then sat there for four more, moving in the third decimal place. Accuracy on a held-out set was 34 per cent.

Thirty-four per cent across eleven categories sounds like learning. Eleven categories at random would be nine per cent, and the model was nearly four times that, so the team's reading was that the model had learned something real and needed more data or a bigger architecture. The plan agreed on Tuesday was to label another twenty thousand tickets, which would take the support team about three weeks of evenings.

One person asked what the loss would be for a model that had learned nothing at all, and computed it before the meeting ended.

The eleven categories were not balanced. Deliveries were 31 per cent of tickets, warranty claims 18, returns 12, payments 9, installation 7, missing parts 6, transit damage 5, cancellations 4, exchanges 3.5, spare parts 2.5, and everything else 2. The entropy of that distribution is the average surprise of the label column, and it came to 2.06 nats. That is the cross-entropy of a model that predicts the base rates every time and knows nothing else.

Their model's loss was 2.34. It was worse than the base rates.

The accuracy figure had hidden it, because 34 per cent looks respectable next to nine and nobody had computed the other comparison: always answering

"deliveries" scores 31 per cent, and the model was three points above that.

The second number pointed at the cause. The natural log of eleven is 2.398, and 2.34 is just under it. A loss stalling immediately below the log of the number of classes is the signature of a model whose outputs have been flattened almost to uniform before they reach the loss. The engineer looked, and the training loop applied a softmax to the logits and then passed the probabilities into a cross-entropy loss that expects raw logits. The loss applied a second softmax. Probabilities all lie between zero and one, so treated as logits they differ by at most one, and a softmax over such numbers is close to flat: three classes with 0.9, 0.05 and 0.05 come out as 0.53, 0.23 and 0.23. The model was training, honestly and slowly, towards a ceiling it could not pass.

That was Wednesday. The decision was whether to fix it and retrain, or to demonstrate on Friday with the model they had.

Retraining was four hours, so time was not the constraint. The constraint was that nobody could promise what the fixed model would score, and the board had already been told a number. Showing a working demonstration with a model that routes worse than a rule saying "send everything to deliveries" was, in the engineer's phrase, a thing that would be true for exactly as long as nobody checked.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They removed the softmax, retrained on Wednesday night, and the loss fell to 0.71 nats by the second epoch with accuracy at 79 per cent. The demonstration went ahead on Friday with the real number and with the base-rate floor drawn on the loss plot as a horizontal line at 2.06, which the team kept in every plot afterwards. The twenty thousand extra labels were not collected. The engineer added two assertions to the training script: one that the loss

function is given raw logits rather than probabilities, and one that fails the run if the first epoch's loss is above the entropy of the label column, since a model above that line has learned less than the base rates and there is no configuration in which continuing is the right response.

Why is the entropy of the label column the loss of a model that has learned nothing but the base rates?

A model that predicts the class frequencies and nothing else assigns each outcome its base-rate probability, so its average surprise is exactly the sum of each probability times minus the log of that probability, which is the entropy. Any model whose cross-entropy is above that number is paying more per ticket than a constant predictor, which means it has learned less than the frequencies. It is a single line of arithmetic on the label column and it should be computed before training starts, not after.

The loss stalled at 2.34, just under ln(11) = 2.398. Why is that particular value a fingerprint rather than a coincidence?

The log of the number of classes is the loss of a model that outputs a uniform distribution. A second softmax applied to probabilities compresses every output towards uniform, because probabilities differ by at most one and a softmax over differences of that size is nearly flat. The model can still improve slightly, which is why the loss sat just below the uniform value rather than exactly on it, but it can never get far from it. Whenever a classifier learns and then stalls close to ln of the class count, look for an extra softmax.

Accuracy of 34 per cent against nine per cent for random guessing looked like progress. What comparison should the team have made instead?

Against the trivial baseline that matters, which is always predicting the commonest class: deliveries at 31 per cent. Random guessing is not the alternative anyone would deploy, so it is the wrong ceiling to be measured against. The same mistake in loss terms is comparing against the log of the class count instead of against the entropy of the actual label distribution, and on skewed labels the two differ enough to turn a failure into an apparent success.

MODULE 5 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Why does every library subtract the largest logit before exponentiating inside softmax?

A. It makes the largest probability exactly one, which speeds up the division

B. float32 overflows at exp of 88.7, and softmax cannot see a shift shared by every logit

C. It centres the logits on zero, which improves the gradient's conditioning

D. It converts the logits into probabilities before the exponential is applied

2 A binary label is 1 in ten per cent of rows. A model reaches a cross-entropy of 0.40 nats. What should you conclude?

A. It is learning well, since 0.40 is fairly close to zero

B. It needs more data, because the loss has not yet reached its floor

C. It is worse than predicting the base rate, whose loss is 0.325 nats

D. The loss is meaningless without knowing the number of parameters

3 A language model's loss plateaus at 2.1 nats and will not fall further. Which reading does the identity relating cross-entropy, entropy and KL divergence support?

A. The optimiser has failed, since a correct model would reach a loss of zero

B. Language has its own entropy, so the loss has a floor above zero that no model goes beneath

C. The KL divergence has gone negative, which cancels the remaining entropy

D. The plateau proves the model has memorised its training set

4 With softmax and cross-entropy the gradient with respect to a logit is the predicted probability minus the target. What does that shape do to training?

A. It makes the gradient equal for every class, which balances the rare ones
B. It pushes hardest on confident mistakes and barely at all on what is already right
C. It makes the gradient largest for the classes the model already predicts well
D. It removes the need for a learning rate, since the step size is set by the error

5 You fit house prices with squared error and a few very large sales drag the whole fit. What does that say about the loss you chose?

A. Nothing about the loss: it is a data-cleaning problem and those sales should be dropped
B. Squared error requires standardised targets, and prices are not standardised
C. Squared error assumes Gaussian errors of constant spread, so a large error is treated as nearly impossible
D. Squared error is undefined for quantities that cannot be negative

6 Adam's second moment uses a decay of 0.999. Roughly how many steps does that average remember, and why longer than the first moment's?

A. About a thousand, because it estimates a variance, which is noisy from few samples
B. About ten, the same as the first moment, since both are running averages
C. About a million, since 0.999 is squared inside the update
D. It has no window: an exponential average keeps every past value at equal weight

MODULE 6

Estimation: from a sample to a number you can trust

What an estimator is and why the variance divides by n − 1, the mechanism behind the square-root law and the bell curve, standard errors and intervals you can compute on paper, what zero failures actually prove, the Beta distribution as a rate that learns, regularisation derived as a prior, and the selection effect that makes every winner look better than it is.

> BY THE END OF THIS MODULE YOU CAN
>
> Compute a standard error and a confidence interval for an accuracy figure by hand, say from the sample size and base rate whether the textbook interval can be trusted, bound a failure rate from a run with zero failures, derive an L2 or L1 penalty from a stated prior on the weights, and predict how much the best of k noisy candidates will fall back when re-measured

49 · 9 min

An estimator is a recipe, and why the variance divides by n − 1

THE ONE THING TO KEEP

An estimator is a rule from sample to number with a bias and a variance of its own, and the sample variance divides by n − 1 because distances measured from the sample's own mean are systematically too small by exactly one degree of freedom.

A number computed from a sample

An estimator is a rule: give it a sample, it gives back a number that stands in for something about the whole population. The sample mean estimates the population mean. The fraction of test items a model gets right estimates its accuracy on everything it will ever see. The rule is the estimator; the number it returns on one particular sample is the estimate.

Because the sample is random, the estimate is too. Draw a different sample and you get a different number. So an estimator has properties of its own, separate from any single result. Two matter most.

- **Bias**: if you repeated the sampling forever and averaged all the estimates, would you land on the truth? If yes, the estimator is unbiased.
- **Variance**: how far do the estimates scatter from one sample to the next?

The sample mean is unbiased for the population mean. The most common recipe for the variance is not, and the reason is worth seeing once with numbers.

Why dividing by n undercounts

Take a population of two values, 0 and 2. Its mean is 1 and its variance, the average squared distance from the mean, is 1. Now draw samples of size two

with replacement. There are four equally likely samples:

```
sample   sample mean   Σ(x − mean)²   ÷ n   ÷ (n−1)
(0, 0)       0              0          0       0
(0, 2)       1              2          1       2
(2, 0)       1              2          1       2
(2, 2)       2              0          0       0
                          average:    0.5     1.0
```

Divide by `n` and the estimates average to 0.5, half the true variance. Divide by `n − 1` and they average to exactly 1. The `n − 1` is not a fudge; it is the correction that makes the average of the estimates equal the truth.

The mechanism: the formula measures distances from the *sample* mean, and the sample mean was chosen to be as close to the sample points as possible. Module 2 showed that the mean is the point minimising the sum of squared distances. So distances from it are systematically smaller than distances from the true mean, which the sample does not know. One degree of freedom was spent estimating the mean, and `n − 1` is what is left.

At `n = 1000` the two recipes differ by a tenth of a per cent and nobody cares. At `n = 5` they differ by 25 per cent, and small-sample statistics is full of places where they matter.

The trap in the libraries

The same column gives two different variances depending on which library you ask:

```
import numpy as np, pandas as pd, torch
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
np.var(x)                      # 8.25   divides by n
pd.Series(x).var()             # 9.17   divides by n − 1
torch.tensor(x, dtype=torch.float).var()   # 9.17   n − 1 by
default
np.var(x, ddof=1)              # 9.17   ddof = 'delta degrees
of freedom'
```

The ratio is `n / (n - 1)`, here `10 / 9`. The standard deviation differs by the square root of that. If a number you computed disagrees with a colleague's by a few per cent, check `ddof` before anything else.

Unbiased is not the same as best

Bias is one property. An estimator's overall error combines both:

```
mean squared error = bias² + variance
```

A slightly biased estimator with much lower variance can be more accurate on any single sample. Dividing by `n` is biased but has lower variance than dividing by `n - 1`, and for some purposes it wins. The whole of regularisation, later in this module, is accepting a little bias to buy a lot less variance.

Two more surprises of the same kind. The sample standard deviation, the square root of the `n - 1` variance, is *still* biased, low, because the square root of an average is less than the average of square roots. And the sample maximum is always a biased estimate of the population maximum: the largest of ten draws is never larger than the largest thing there is, and is usually smaller.

A useful estimator you were not taught

That last fact has a famous fix. Suppose items are numbered `1` to `N`, you see `k` of them at random, and the largest number you saw was `m`. The best estimate of `N` is not `m` but

```
N ≈ m + m/k - 1
```

Seeing five items with a maximum of 60 suggests `N ≈ 60 + 12 - 1 = 71`. Allied statisticians used this on the serial numbers of captured German tanks and estimated monthly production at 246 against a true figure of 245, while intelligence reports said 1,400. The same recipe estimates how many distinct records a system holds from a few sampled ids, or how many bugs a codebase has from the ones found so far. An estimator is a recipe, and some recipes are much better than the obvious one.

BEFORE YOU MOVE ON

You compute the variance of a 10-row column with NumPy's default and with pandas' default and get 4.00 and 4.44. Which is which, and why do they differ?

A. NumPy gives 4.00 by dividing by n; pandas gives 4.44 by dividing by n – 1.

B. NumPy gives 4.44 because it applies the n – 1 correction; pandas gives 4.00 because it does not.

C. The difference means one of the two libraries has a rounding bug in its variance routine.

D. Pandas silently drops a missing value, so it divided by 9 rows instead of 10.

50 · 9 min

The law of large numbers, and how fast it works

THE ONE THING TO KEEP

Because variances of independent draws add, the spread of an average falls as σ/√n, which is slow, and the law stops applying when the variance is infinite or when correlated rows are counted as independent, in which case the effective n is the row count divided by 1 + (m – 1)ρ.

The law, and the mechanism behind it

The average of many independent draws settles toward the expected value. Everyone knows this; fewer people know how fast, and fewer still know when it stops being true. Both come from three lines of algebra.

Each draw has some variance σ^2. For independent draws, variances add, so the sum of n of them has variance $n\,\sigma^2$. The average is the sum divided by n, and dividing a quantity by n divides its variance by n^2. So

```
Var(average) = n σ² / n² = σ² / n
SD(average)  = σ / √n
```

The spread of the average shrinks with the square root of the sample size, and the word doing the work is *independent*. If the draws were correlated, the covariances would add too, and the `n` in the denominator would be smaller than it looks.

How slow the square root is

A coin flip has `σ = 0.5`. The average of 100 flips has spread 0.05; of 10,000 flips, 0.005. Ten times the precision costs a hundred times the samples. Module 4 stated this; here it is derived, and it is worth feeling the weight of it. An accuracy estimate you want to sharpen from ±3 points to ±1 point needs nine times the test items. There is no cleverness that changes the exponent for a plain average.

A quick sizing rule follows. For a proportion, `σ` is at most 0.5, so to get within `ε` with about 95 per cent confidence you need roughly `n = (2 × 0.5 / ε)² = 1 / ε²`. Within one point, `ε = 0.01`, gives `n = 10,000`. Within five points, 400. The course `evaluating-ai` turns this into a procedure; the arithmetic is only this.

A guarantee with no assumptions

The square-root law says how the spread shrinks. Chebyshev's inequality turns it into a promise that needs no assumption about the shape of the distribution at all:

```
P(|average − μ| ≥ k × σ/√n) ≤ 1 / k²
```

For 400 coin flips, `σ/√n = 0.025`. The chance the average is off by 0.125 or more, five spreads, is at most `1/25 = 4` per cent. The true chance is far smaller, under one in a million, because the bell curve of the next lesson has much thinner tails than Chebyshev allows for. But Chebyshev holds for *any* distribu-

tion with finite variance, and "at most four per cent, whatever the shape" is sometimes the sentence you need.

When the law fails: infinite variance

The derivation assumed `σ²` exists. Some distributions have none. The Cauchy distribution, which is what you get from the ratio of two normal variables, has a bell-shaped middle and tails so heavy that its variance, and even its mean, are undefined.

```
import numpy as np
rng = np.random.default_rng(0)
for _ in range(4):
    print(rng.standard_cauchy(200_000).mean())
# 2.67   0.24   -0.01   -1.66
```

Four averages of two hundred thousand draws each, and they disagree by more than four units. The average of a million Cauchy draws is exactly as spread out as a single draw. Every so often a single value of a hundred thousand arrives and drags the whole average with it. The law of large numbers does not fail slowly here; it does not apply.

Real quantities are never exactly Cauchy, but the heavy-tailed ones from module 4, response times, incomes, file sizes, the length of the longest document, live near that edge. Their averages converge, if at all, far more slowly than `1/√n` promises, and the estimate keeps jumping as rarer extremes turn up. That is the mechanism behind the advice to report percentiles: a percentile has a finite variance even when the data do not.

When the law fails: correlated draws

The second assumption was independence. Suppose an evaluation set has 2,000 answers, but they come from 80 users with 25 answers each, and answers from the same user tend to agree: a correlation of `ρ = 0.5` within a user. The variance of the average is inflated by the **design effect**:

```
design effect = 1 + (m − 1) ρ = 1 + 24 × 0.5 = 13
effective n   = 2,000 / 13 ≈ 154
```

Two thousand rows carry the information of 154 independent ones. An interval computed as if `n = 2000` is 3.6 times too narrow. This is the most common way a confident number turns out to be wrong: not a bad formula, but a sample that was less independent than it was counted as. Sessions, documents, days, batches, and anything generated by the same prompt are the usual culprits.

The third failure: the thing moved

Averaging assumes there is one fixed quantity to converge to. An average of a model's accuracy across six months of traffic converges to the average of a changing thing, which describes no month in particular. The law of large numbers is exact about a stationary world and silent about any other.

The rule to keep

Spread falls as `σ/√n`, provided the variance is finite and the draws are independent. Check both before trusting the `n`. When the variance is not finite, average something else. When the draws are clustered, count the clusters.

BEFORE YOU MOVE ON

An evaluation set has 2,000 answers from 80 users, 25 each, and answers from the same user correlate at 0.5 on the metric. Roughly how many independent answers is that worth?

A. 2,000: each answer is a separate row, and the formula uses the row count.

B. 80, since only one answer per user can be trusted as independent.

C. About 150.

D. 1,000, because a correlation of 0.5 halves the information in the set.

51 · 9 min

Why averages look normal, and when they refuse to

THE ONE THING TO KEEP

Averages of independent draws with finite variance become normal because adding smooths and the normal is the shape smoothing leaves alone, but skew slows it, heavy tails defeat it entirely, and the theorem describes the average rather than the data.

Watch it happen

The central limit theorem says that the average of enough independent draws is approximately normal, whatever shape the draws came from, provided their variance is finite. Rather than prove it, watch it.

```
import numpy as np, matplotlib.pyplot as plt
rng = np.random.default_rng(0)
for n in (1, 2, 10, 50):
    avg = rng.integers(1, 7, size=(100_000, n)).mean(axis=1)
# average of n dice
    plt.hist(avg, bins=60, alpha=0.5, label=f"n={n}")
plt.legend(); plt.show()
```

One die is flat: six equal bars. The average of two is a triangle, peaked at 3.5. The average of ten is already a recognisable bell. At fifty it is a bell to the eye and to any test you care to run, and its spread is `1.71 / √50 = 0.24`, exactly what the last lesson predicted.

Now change the source to something skewed:

```
avg = rng.exponential(1.0, size=(100_000, n)).mean(axis=1)
```

The exponential distribution starts at zero and has a long right tail. The average of five is still visibly lopsided. At thirty it is close to a bell; at a hundred, indistinguishable. Skew slows the convergence, and the rule of thumb that thirty is enough is a rule for mild skew, not a law.

Finally, a heavy tail:

```
avg = (rng.pareto(1.5, size=(100_000, 10_000)) +
1).mean(axis=1)
```

The Pareto with tail exponent 1.5 has a finite mean but infinite variance. The average of ten thousand draws is still wildly skewed, with a long tail of averages that were dragged by a single enormous value. There is no `n` at which this becomes a bell, because the theorem's one condition is not met.

Why adding things makes a bell

The distribution of a sum of two independent variables is the convolution of their two distributions, which is a kind of smoothing: every bump in one is smeared by the whole shape of the other. Do that repeatedly and bumps wash out. The normal distribution is the shape that convolution leaves unchanged, apart from widening; it is the fixed point of the process, so everything with finite variance drifts toward it. Heavy tails escape because a single draw can be larger than the sum of all the others, and no amount of smoothing removes a spike that keeps arriving.

What the theorem is about, and what it is not

The theorem describes the distribution of the **average**, across repeated samples. It says nothing about the data. Two confusions follow from mixing those up.

"My data are not normal, so I cannot compute a confidence interval for the mean." Wrong, usually. The interval is about the average, and the average of fifty non-normal values is close to normal unless the tail is heavy.

"The average is normal, so the data must be." Also wrong. Salaries are skewed, and the average of a thousand salaries is bell-shaped anyway.

The legitimate worry is the third case: "my metric is the mean of heavily skewed latencies, so fifty may be too few for the bell to have formed." There the honest move is to simulate, as above, with your own data, or to report a percentile, whose sampling distribution does not depend on the tail.

A rule that replaces the thirty

For a proportion, the average of zeros and ones, the bell needs both outcomes to have appeared a reasonable number of times. The usual condition is `np ≥ 10` and `n(1 − p) ≥ 10`. An event with `p = 0.01` therefore needs `n ≥ 1000` before the normal approximation to its rate is trustworthy, not thirty. Rare-event rates are exactly where people apply the thirty rule and get intervals that include negative numbers.

Where the bell appears inside a model

The same theorem is why so much of deep learning is analysed with Gaussians. A unit in a layer computes a sum of hundreds of products of weights and inputs; that sum is near-normal by the theorem, which is what the initialisation arithmetic in module 8 relies on. A mini-batch gradient is an average over the batch, so its noise around the full gradient is near-normal, which is why the noise in stochastic gradient descent is modelled as Gaussian. And a batch loss is a sum over examples. None of these are assumptions; they are the theorem applied.

The tails converge last

One limit to state plainly. The theorem promises the *middle* of the distribution of the average becomes a bell. The far tails take much longer. The probability that the average of thirty skewed values sits more than five spreads from its mean is not the Gaussian's `3 × 10^−7`; it can be a thousand times larger. This matters precisely when you are estimating something rare, a failure rate or a worst case, and it is why the next lessons treat rare rates with their own arithmetic rather than a bell.

BEFORE YOU MOVE ON

A metric is the mean of 50 request latencies, and the latencies are heavily skewed. Which statement is right?

A. The mean is normally distributed, because 50 is above the threshold of 30.

B. Whether the mean is near-normal depends on the tail, and 50 may be too few.

C. The mean cannot be used, because the underlying latencies are not normally distributed.

D. Taking the log of each latency first makes the mean of the raw latencies normal.

52 · 9 min

Standard error for a mean and a proportion, by hand

THE ONE THING TO KEEP

The standard error of a proportion is √(p(1−p)/n), at most 0.5/√n, so 400 items give about ±5 points; independent errors combine by adding squares, while a paired comparison on shared items depends only on the items the two systems disagree about.

Two formulas, and the numbers they give

The standard error is the spread of an estimate across repeated samples: `σ/√n` from two lessons ago, with the unknown `σ` replaced by something you can compute.

For a **mean**, use the sample standard deviation `s`:

```
SE(mean) = s / √n
```

For a **proportion** `p`, such as an accuracy, the standard deviation of a 0/1 variable is `√(p(1 − p))`, so

```
SE(p) = √( p (1 − p) / n )
```

A model scores 85 per cent on 200 items:

```
SE = √(0.85 × 0.15 / 200) = √0.000638 = 0.0252
```

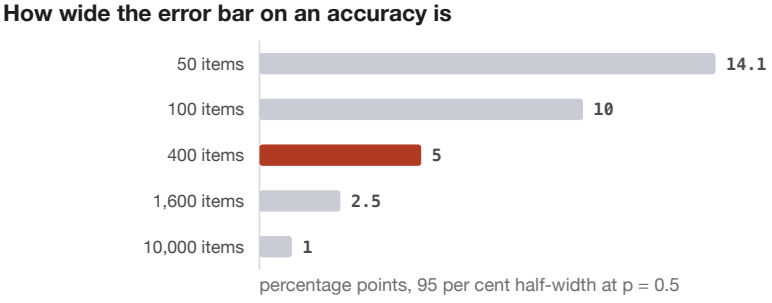
About 2.5 points. The rough 95 per cent range is two standard errors either side: 80 to 90. The same 85 per cent on 50 items gives `SE = 0.0505`, five points, and a range of 75 to 95. Twenty percentage points wide, from a number that was reported to one decimal place.

The table to carry in your head

The product `p(1 − p)` is largest at `p = 0.5`, where it is 0.25, so the worst-case standard error is `0.5/√n`. Two of those, for the 95 per cent half-width:

```
n =    100  →  ±10 points
n =    400  →  ± 5 points
n =  1,600  →  ±2.5 points
n = 10,000  →  ± 1 point
```

Quadrupling the items halves the width. At `p = 0.95` the factor `√(0.95 × 0.05) = 0.218` is under half the worst case, so the widths shrink accordingly, but the table is the bound to use when you do not yet know `p`.



Quadrupling the items halves the width, and for a plain proportion nothing does better than that. An 87 and an 84 on the same hundred items are the same number, and an accuracy quoted without its n is a rumour.

The difference between two models

Two models score 85 and 88 per cent, each on its own 200 items. Standard errors of independent estimates combine by adding their squares:

```
SE(A) = 0.0252,   SE(B) = √(0.88 × 0.12 / 200) = 0.0230
SE(B − A) = √(0.0252² + 0.0230²) = 0.0341
```

The gap is 3 points and the standard error of the gap is 3.4 points. The gap is smaller than its own uncertainty. Nothing has been shown.

Now suppose both were scored on the **same** 200 items. Item difficulty is then shared: a hard item is hard for both, and that shared part cancels in the difference. What remains is the items on which the two models *disagree*. Say there are 30 such items, and model B wins 18 of them to A's 12. The difference is `(18 − 12)/200 = 0.03`, as before, but its standard error is now approximately

```
SE(paired difference) ≈ √(18 + 12) / 200 = √30 / 200 = 0.0274
```

Smaller, because 170 items that both got right or both got wrong contribute nothing to the noise of the comparison. It is still only 1.1 standard errors, still nothing shown, but the paired design was closer. This is why comparing on shared items is worth so much more than comparing on separate sets, and the course `evaluating-ai` builds the practice on it. The arithmetic is: count the discordant items.

Standard error is not standard deviation

The standard deviation describes the spread of the data. The standard error describes the uncertainty of an estimate computed from the data. At `n = 100` the second is a tenth of the first. A plot with SE error bars looks ten times tighter than one with SD bars, and papers switch between them without saying which. When you read "mean ± 2.3", ask which. When you write one, say which.

A useful check: standard deviation does not shrink as you collect more data, because the data are as spread as they are. Standard error does. If a reported uncertainty gets smaller as `n` grows, it is a standard error.

Three conditions, each already met or not

The proportion formula assumes the items are **independent**; the design-effect lesson said what to do when they are not. It assumes `p` is **not near 0 or 1**, or `n` is large enough that `np` and `n(1 − p)` both exceed about 10; the next lesson shows what breaks when that fails. And it assumes the quantity is a **mean**. A median, a 95th percentile, an F1 score or an AUC is not a mean, and the formula does not apply; for those, the bootstrap, resampling the items and recomputing, is the tool, and `evaluating-ai` covers its pitfalls.

On paper, then in code

```
import math
k, n = 170, 200
p = k / n
se = math.sqrt(p * (1 − p) / n)
print(p, se, p − 2*se, p + 2*se)     # 0.85  0.0252  0.80  0.90
```

The whole computation fits on the back of an envelope, and doing it there, once, before quoting any accuracy, is the habit this lesson exists to install. An accuracy without an `n` is a rumour.

BEFORE YOU MOVE ON

Two models score 85 and 88 per cent on the same 200-item test. Which computation tells you whether the 3-point gap is more than noise?

A. Compare each model's own standard error, about 2.5 points, with the gap; since 3 is larger, the gap is real.
B. Combine the two standard errors as independent, giving about 3.4 points, so the gap is not real.
C. Any gap above one point on 200 items is significant, because 1/200 is the resolution of the test.
D. Count the items on which they disagree and compute the error from those.

53 · 10 min

What a 95 per cent interval promises, and where the textbook one breaks

THE ONE THING TO KEEP

A 95 per cent interval is a procedure that captures the truth in 95 per cent of repetitions, and the textbook estimate ± 1.96 SE breaks that promise near 0 or 1, covering only 63 per cent at p = 0.98 and n = 50, which the Wilson interval fixes by solving for the plausible true values instead.

The promise is about the procedure

A 95 per cent confidence interval is a recipe with a guarantee: if you drew sample after sample and ran the recipe each time, 95 per cent of the intervals it produced would contain the true value. The guarantee is about the long run of the *procedure*. It is not a statement that the truth has a 95 per cent chance of being inside the one interval in front of you; that reading belongs to the Bayesian interval two lessons on, and it needs a prior to earn it.

You can check the promise directly.

```
import numpy as np
rng = np.random.default_rng(1)
true_p, n, hits = 0.7, 200, 0
for _ in range(10_000):
    k = rng.binomial(n, true_p)
    p = k / n
    se = np.sqrt(p * (1 - p) / n)
    hits += (p - 1.96 * se) <= true_p <= (p + 1.96 * se)
print(hits / 10_000)            # about 0.95
```

The recipe here is the textbook one: estimate plus or minus 1.96 standard errors. At `p = 0.7` and `n = 200` it keeps its promise. The value 1.96 is the point on the bell curve that leaves 2.5 per cent in each tail, and the recipe leans on the central limit theorem to justify the bell.

Where the textbook interval breaks

Run the same check with `true_p = 0.98` and `n = 50`. The coverage is not 95 per cent. It is **63.5 per cent**. The recipe misses the truth more than a third of the time while claiming to miss it one time in twenty.

The mechanism is visible in one example. A model scores 49 of 50:

```
p = 0.98,   SE = √(0.98 × 0.02 / 50) = 0.0198
interval = 0.98 ± 1.96 × 0.0198 = 0.941 to 1.019
```

An upper bound above 1 for a proportion. And a model scoring 50 of 50 gives `SE = 0`, so the interval is `[1.00, 1.00]`: a claim of certainty from fifty observations. The standard error was computed from the *estimate*, and near the boundary the estimate is a poor stand-in for the truth, so the width is wrong. This is not a small-sample curiosity. At `p = 0.9` and `n = 100`, the coverage is 93 per cent, still short, and it oscillates unpredictably as `n` changes.

The interval that keeps the promise

The Wilson interval fixes the mechanism by solving for the values of the true `p` under which the observation would be plausible, instead of assuming the

estimate is the truth. With `z = 1.96` and `z² = 3.84`:

```
centre     = (p + z²/2n) / (1 + z²/n)
half-width = z × √( p(1−p)/n + z²/4n² ) / (1 + z²/n)
```

For 49 of 50:

```
centre     = (0.98 + 0.0384) / 1.0768 = 0.946
half-width = 1.96 × √(0.000392 + 0.000384) / 1.0768 = 0.0507
interval   = 0.895 to 0.997
```

For 50 of 50 it gives `0.929 to 1.000`. Both intervals stay inside `[0, 1]`, both are asymmetric, and both are wider than the textbook version on the side that matters. The centre is pulled toward 0.5 by the `z²/2n` term, which acts like adding two successes and two failures to the count; that pull is the correction.

Forty-nine right out of fifty, two intervals

Textbook: the estimate plus or minus 1.96 SE

- 0.941 to 1.019, an upper bound above 1
- Fifty out of fifty gives SE zero and the interval 1.00 to 1.00
- Symmetric, because the recipe assumes it is
- Real coverage at p = 0.98 and n = 50 is 63.5 per cent

Wilson: solve for the plausible true rates

- 0.895 to 0.997
- Fifty out of fifty gives 0.929 to 1.000
- Asymmetric, and always inside 0 to 1
- Coverage stays near the 95 it promises

The textbook width is computed from the estimate, and near 0 or 1 the estimate is a poor stand-in for the truth, so the width comes out wrong. Use Wilson whenever np or n times one minus p is under about ten, which for a rare failure rate is nearly always.

```
from statsmodels.stats.proportion import proportion_confint
proportion_confint(49, 50, method="wilson")     # (0.895,
0.997)
```

If `statsmodels` is not to hand, the two lines above are the whole formula. Use Wilson whenever `np` or `n(1 − p)` is under about 10, which for a rare failure rate is nearly always.

Means with small samples

For a mean rather than a proportion, the small-sample problem is different: `s` is a noisy estimate of `σ`, and the 1.96 should widen to account for it. The replacement is the t-distribution's quantile, which depends on `n − 1` degrees of freedom: 2.26 at `n = 10`, 2.09 at `n = 20`, 2.01 at `n = 50`, and 1.96 in the limit. At `n = 10` the interval is 15 per cent wider than the textbook one, which is the honest width.

Three things an interval cannot tell you

Bias. An interval quantifies sampling noise only. A test set drawn from the wrong population gives a tight interval around the wrong number, and nothing in the width warns you. Module 4 and `evaluating-ai` both return to where the items came from; no interval formula does.

Which values inside are likely. The interval is a range with a coverage guarantee, not a distribution over the range. The truth is not more likely to be at the centre in any sense the recipe licenses.

That 95 is special. It is a convention. A 90 per cent interval is narrower by a factor of `1.645/1.96 = 0.84`, a 99 per cent one wider by `2.576/1.96 = 1.31`. Report which one you used, and prefer showing the interval to a bare yes or no about whether it excludes some number.

The habit

An accuracy is a proportion; compute Wilson. A mean over fewer than fifty items; use t. Anything else, resample. And before quoting any of them, ask whether the items were independent, because that assumption is the one no formula on this page can check.

BEFORE YOU MOVE ON

A model gets 50 out of 50 on a test set, and the interval is computed as 1.00 ± 1.96 × 0, giving [1.00, 1.00]. What has gone wrong?

A. Nothing: fifty correct answers out of fifty is strong evidence, and the interval reports that faithfully.

B. The estimate's own error formula gives zero at p = 1; Wilson gives about 0.93 to 1.

C. Fifty items is too few to compute any interval at all, so the result is undefined.

D. 1.96 should have been replaced by the t-value for 49 degrees of freedom, which widens the interval.

54 · 8 min

Zero failures in 300 tries, and what that does not prove

THE ONE THING TO KEEP

Zero failures in n independent trials bounds the failure rate at roughly 3/n with 95 per cent confidence, so 300 clean runs are consistent with a one-per-cent failure rate, and observing a failure of rate p reliably needs about 3/p trials.

The most common wrong sentence in a release note

"We ran it 300 times and it never failed." The sentence is true, and the conclusion people draw from it, that the failure rate is zero or near enough, is not. Zero observed failures is evidence, and the question is how much. The answer fits on one line.

Deriving the rule

Suppose the true failure rate is `p` and the trials are independent. The chance of seeing no failures in `n` trials is `(1 − p)^n`. Ask: what is the largest `p` for which zero failures would still be unsurprising, say at least 5 per cent likely?

```
(1 − p)^n = 0.05
n × ln(1 − p) = ln 0.05 = −3.0
n × (−p) ≈ −3.0          (since ln(1 − p) ≈ −p for small p)
p ≈ 3 / n
```

That is the **rule of three**: after `n` trials with no failures, the 95 per cent upper bound on the failure rate is about `3/n`.

```
0 failures in     30  →  rate could be up to 10%
0 failures in    300  →  up to 1%
0 failures in  3,000  →  up to 0.1%
0 failures in 30,000  →  up to 0.01%
```

Three hundred clean runs are consistent with a one-in-a-hundred failure rate. If the system will handle a thousand requests a day, that is ten failures a day the test could not have seen. Fifty clean runs, which is what a demo usually amounts to, are consistent with six per cent.

The other direction: how many trials to see one

The same algebra answers the planning question. To have a 95 per cent chance of observing at least one failure of rate `p`, you need `n ≈ 3/p` trials. A failure that happens once in a thousand requests needs three thousand trials to show up reliably. One that happens once in a hundred thousand needs three hundred thousand. If your test budget is a thousand runs, you are blind to anything rarer than about one in three hundred, and you should say so rather than report a zero.

Seeing a few failures

Once failures do appear, the count over many trials is approximately Poisson, and the 95 per cent upper bounds on the expected count are:

```
observed 0  →  at most 3.0
observed 1  →  at most 4.7
observed 2  →  at most 6.3
observed 3  →  at most 7.8
```

Divide by `n` for the rate. Two failures in 1,000 runs means the rate is 0.2 per cent as a point estimate but could be as high as 0.63 per cent. Note how slowly the bound tightens: the upper bound for three observed failures is not much more than double that for zero. A handful of failures is a handful of information.

Estimating a rare rate to a given precision

Seeing a failure is one thing; measuring its rate is another. The standard error of a proportion, relative to the proportion itself, is about `1/√(np)` when `p` is small. To pin a rate to within ±50 per cent of itself, two standard errors, you need `1/√(np) = 0.25`, so `np = 16`: sixteen observed failures, whatever the rate. To within ±10 per cent you need `np = 400`, four hundred failures. A failure rate of 0.1 per cent therefore needs 16,000 trials to estimate coarsely and 400,000 to estimate well. This is why rare-event rates in published evaluations are so often quoted with one decimal place and no interval: the interval would swallow the number.

Where the rule is used, and misused

- **Safety evaluations.** "No jailbreak in 200 attempts" bounds the success rate for *attacks of that kind* at about 1.5 per cent. It says nothing about attacks the red team did not think of, which is a different limitation, covered in `evaluating-ai`.
- **Clinical trials.** A drug tested on 1,500 patients with no case of a side effect cannot rule out that it strikes one patient in 500. This is the standard textbook example, and it is why rare side effects are found after approval.

- **Hardware and infrastructure.** A component with no failures across 10,000 hours of testing may fail once per 3,300 hours.

The misuse is always the same: treating the point estimate of zero as the answer, when the honest answer is a bound.

Two assumptions to check

The trials must be **independent**, and they must be **drawn from the deployment distribution**. Three hundred test prompts written by one engineer in one afternoon are neither: they share that engineer's blind spots, and they were not sampled from what users will send. The rule of three then bounds the failure rate on prompts like those, which is a smaller claim than it sounds. The arithmetic is exact; the sample is where the honesty goes.

The sentence to write instead

Not "it never failed", but "in 300 independent trials we saw no failures, which bounds the rate below about 1 per cent at 95 per cent confidence for inputs of this kind." Longer, and true.

BEFORE YOU MOVE ON

A red team runs 200 attack prompts and none succeed. What can be concluded about the attack success rate?

A. It is zero, since no attack got through.

B. Probably under about 1.5 per cent, for attacks like these.

C. It is below 0.5 per cent, because one failure in 200 would be 0.5 per cent and none was seen.

D. Nothing at all can be concluded until at least one attack succeeds.

55 · 10 min

Updating a rate as evidence arrives: the Beta distribution

THE ONE THING TO KEEP

A Beta(a, b) belief about a rate updates to Beta(a + k, b + m) after k successes and m failures, so the prior is just pseudo-counts that fade as data accumulate, which is why the same arithmetic gives credible intervals, click-rate smoothing and Thompson sampling.

A belief about a rate, as a curve

The last three lessons treated a rate, an accuracy or a failure probability, as a fixed unknown and built intervals around estimates of it. The Bayesian route treats the rate itself as uncertain and describes that uncertainty with a distribution over the interval from 0 to 1. The distribution that makes this easy is the **Beta**, written `Beta(a, b)`, with two positive parameters.

The shapes worth knowing: `Beta(1, 1)` is flat, every rate equally plausible. `Beta(2, 2)` is a gentle hump at 0.5. `Beta(50, 2)` is a sharp peak near 0.96. The mean is always

```
mean of Beta(a, b) = a / (a + b)
```

and the peak sharpens as `a + b` grows.

The update is addition

Here is the fact that makes the Beta useful. If your belief about a rate is `Beta(a, b)` and you then observe `k` successes and `m` failures, your updated belief is

```
Beta(a + k, b + m)
```

That is the entire calculation. No integrals, no simulation. The parameters behave as **counts**: `a` is successes seen so far, `b` failures, and the prior `Beta(a, b)` is the statement "before this data, I had seen about `a` successes and `b` failures". Real or imagined, they add the same way.

Start flat, `Beta(1, 1)`, observe 49 correct out of 50, and the posterior is `Beta(50, 2)`, with mean `50/52 = 0.962` and a 95 per cent range from 0.896 to 0.995. Compare the Wilson interval from two lessons ago, 0.895 to 0.997: the two routes agree to the second decimal, as they generally do once the data outweigh the prior.

Start flat, observe 300 runs with no failures, and the posterior on the *failure* rate is `Beta(1, 301)`, mean `1/302 = 0.33` per cent, 95 per cent upper bound 0.99 per cent. The rule of three, recovered exactly. The flat prior's one imaginary failure is what keeps the estimate off zero, and `(k + 1)/(n + 2)` as the mean after `k` of `n` is Laplace's rule of succession, two centuries old.

What you get that a confidence interval does not

The interval `0.896 to 0.995` above is a **credible interval**: there is, given the prior and the data, a 95 per cent probability that the rate lies inside it. That is the sentence everyone wants to say about a confidence interval and cannot. The Bayesian version licenses it, at the price of stating a prior. The price is real. Two people with different priors reach different posteriors from the same fifty items, and either can be asked to defend the choice. A confidence interval has no such argument attached, and also no such sentence.

Smoothing, which is the same thing

A ranking system estimates click-through rates. Item A has 1 click from 1 view; item B has 40 clicks from 100 views. By raw rate, A is at 100 per cent and wins. Nobody believes that, and the Beta says why. Use a prior of `Beta(2, 8)`: ten pseudo-views at a 20 per cent rate, a modest belief about what items usually do.

```
A: (1 + 2) / (1 + 10)   = 3 / 11  = 0.273
B: (40 + 2) / (100 + 10) = 42 / 110 = 0.382
```

B wins. A's single view is nearly swamped by the ten imaginary ones; B's hundred real views hold their ground. As A accumulates views the prior fades, since its weight is `a + b` against the growing `n`. This is additive smoothing, and it is in every search engine, every recommender and every spam filter, usually without anyone calling it Bayesian.

Picking from uncertain rates

The Beta also gives the simplest decision rule that works for choosing between options with unknown rates. For each option, draw one random rate from its current Beta; pick the option whose draw is largest; observe the result; update that option's Beta. An option with little data has a wide Beta and is sometimes drawn high, so it gets tried; an option with much data and a low rate is almost never drawn high, so it is left alone. This is Thompson sampling, and it is a working A/B testing engine in a dozen lines:

```
import numpy as np
rng = np.random.default_rng(0)
succ, fail = np.ones(3), np.ones(3)          # Beta(1,1) for
three options
for _ in range(1000):
    i = np.argmax(rng.beta(succ, fail))       # one draw per
option
    reward = rng.random() < true_rates[i]
    succ[i] += reward; fail[i] += 1 - reward
```

Where the Beta is the wrong model

It describes a single fixed rate. If the rate differs by context, by user, by time of day, by input length, a single Beta averages over things that should be kept apart, and its narrow posterior expresses confidence about an average nobody experiences. The remedy is one Beta per context, or a model with the context as an input, which is the point at which this becomes machine learning.

And a strong wrong prior is slow to overcome. `Beta(200, 800)` claims a thousand prior observations; two hundred real successes in a row move its mean only from 0.20 to 0.33. State the prior in counts, and if you cannot defend the count, use a smaller one.

```
from scipy.stats import beta
beta(50, 2).ppf([0.025, 0.975])      # (0.896, 0.995)
```

BEFORE YOU MOVE ON

A ranking system smooths click rates with a Beta(2, 8) prior. Item A has 1 click from 1 view; item B has 40 clicks from 100 views. Which ranks higher, and why?

A. A, because its observed rate is 100 per cent against B's 40 per cent.

B. They tie, because the prior overrides whatever the data say.

C. A, because fewer views means a tighter estimate.

D. B, at about 38 per cent against A’s smoothed 27.

56 · 10 min

Regularisation is a prior: L2, L1 and the diamond

THE ONE THING TO KEEP

Adding a penalty to a loss is maximising a posterior rather than a likelihood, a Gaussian prior on the weights gives the L2 penalty and a Laplace prior gives L1, and L1 produces exact zeros because its pull stays constant as a weight shrinks while L2's pull fades with the weight.

Where maximum likelihood goes wrong, and the repair

Module 5 derived every standard loss from maximum likelihood: choose the parameters that make the data most probable. It also noted the flaw. Given enough parameters and too little data, the parameters that make the training data most probable are ones that have memorised it. The likelihood is happy to set a weight to 40 if that fits one row.

The repair is to say what you believed about the parameters before seeing data. Bayes' rule, from module 4, gives

```
posterior ∝ likelihood × prior
```

and maximising the posterior instead of the likelihood is **maximum a posteriori** estimation, MAP. Take logs:

```
log posterior = log likelihood + log prior + const
```

Minimising the negative of this gives

```
loss = data loss + (−log prior)
```

The second term is the regulariser. Every penalty term you have seen added to a loss is a negative log prior, whether or not the person who added it thought of it that way.

A Gaussian prior is weight decay

Suppose you believe each weight is probably small: normally distributed around zero with spread `τ`. The log of that density is `−w²/(2τ²)` plus a constant, so the penalty is

```
Σ w² / (2τ²)
```

That is the L2 penalty, with strength `λ = 1/(2τ²)`. A large `λ` is a narrow prior, a strong belief that weights are near zero. Weight decay in an optimiser is this penalty's gradient, `λ w`, applied each step. The hyperparameter you tune on a validation set is, in this reading, the width of your prior on the weights, chosen empirically. That is a legitimate way to choose a prior, and it is what nearly everyone does.

A Laplace prior is L1

Believe instead that weights follow a Laplace distribution, whose density is `exp(−|w|/b)`. The log is `−|w|/b`, and the penalty is

```
Σ |w| / b
```

The L1 penalty. Same recipe, different prior, and a qualitatively different result: L1 sets some weights to exactly zero, L2 never does. The course `machine-learning-foundations` uses both; here is the mechanism.

Why L1 reaches zero and L2 does not

Take one weight whose data loss is `(w − 0.3)²/2`, pulling it toward 0.3 with unit curvature.

With an L2 penalty `λw²/2` at `λ = 1`, the total is `(w − 0.3)²/2 + w²/2`. Its derivative, `(w − 0.3) + w`, is zero at `w = 0.15`. The weight is halved. Make `λ` larger and it shrinks further, but the derivative of the penalty is `λw`, which vanishes as `w` approaches zero. The penalty's pull fades exactly when it would need to finish the job. `w` is never exactly zero.

With an L1 penalty `λ|w|` at `λ = 0.5`, the pull is `0.5` toward zero regardless of how small `w` is. The data pull at `w` is `0.3 − w`. If the data pull at `w = 0` is weaker than 0.5, and it is, since `0.3 < 0.5`, the weight cannot leave zero. So `w = 0` exactly. At `λ = 0.1` the data win and `w = 0.3 − 0.1 = 0.2`. The general solution is

```
w = sign(w₀) × max(0, |w₀| − λ)
```

called soft thresholding: shrink by `λ`, and if that crosses zero, stop there. Weights whose data pull is weaker than `λ` are removed. That is a constant force against a fading one, and it is the whole reason L1 produces sparse models.

The usual picture says the same thing geometrically. The L2 penalty's level sets are circles, the L1 penalty's are diamonds, and the minimum of loss plus penalty tends to touch a diamond at a corner, where a coordinate is zero. The picture and the derivative are one fact.

Early stopping is a prior too

Start gradient descent from zero weights and stop after `t` steps. Directions in which the loss has high curvature converge quickly; directions with low curvature have barely moved from zero. Stopping early therefore holds the low-curvature directions near zero, which is what an L2 penalty of strength about `1/(learning rate × t)` would have done. The number of epochs is a regularisation strength, which is why it gets tuned on the validation set like one.

Two honest limits

MAP is the peak of the posterior, not its centre. Under a Laplace prior the peak is sparse, with weights at exactly zero; the posterior mean is not

sparse at all. Sparsity is a property of the point estimate you chose to report, not of the belief. Averaging models trained with L1 loses the zeros.

The prior is a modelling choice with consequences. A Gaussian prior on weights says a weight of 3τ is rare and 6τ nearly impossible. If the true function needs one large weight, L2 will fight it in every step and settle for many small ones instead. Regularisation encodes what you expect the solution to look like, and when the expectation is wrong the fit is worse, not merely slower.

One line to try

```
import numpy as np
w0, lam = 0.3, 0.5
l2 = w0 / (1 + lam)                                    # 0.2 for
lam=0.5
l1 = np.sign(w0) * max(0.0, abs(w0) - lam)              # 0.0
```

Change `lam` and watch L1 snap to zero while L2 only ever halves the distance.

BEFORE YOU MOVE ON

Why does an L1 penalty set some weights to exactly zero while an L2 penalty only shrinks them?

A. The L1 pull stays at λ however small the weight; the L2 pull fades with it.

B. L1 is simply a stronger penalty than L2 for the same λ, so it pushes harder.

C. L2 squares the weights, so small weights are penalised more heavily than under L1.

D. L1 rounds any weight below a threshold to zero after training finishes.

57 · 9 min

Regression to the mean, and the winner's curse in model selection

THE ONE THING TO KEEP

Any measurement is truth plus noise, so the best of k candidates was selected partly for lucky noise and falls back on re-measurement by an amount set by the reliability r and the count k, which is why the winner of a hyperparameter search scores lower on a fresh test set through no fault of its own.

The best of twenty is not the best

Train twenty models that are, in truth, identical: each has an accuracy of exactly 80 per cent on the population. Evaluate each on its own 200 items. The standard error is 2.8 points, so the twenty scores scatter around 80 with that spread. Pick the highest. Its expected value is not 80. The expected maximum of twenty draws from a bell curve is about 1.9 standard deviations above the mean, so the winner scores around

```
80 + 1.9 × 2.8 ≈ 85.3
```

Now evaluate that winner on a fresh 200 items. Expected score: 80. It "lost" five points. Nothing happened to the model. It was chosen because its noise was favourable, and fresh items come with fresh noise.

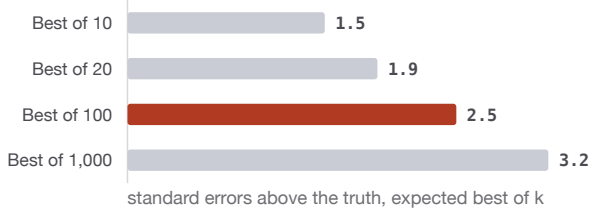
This is regression to the mean, and the mechanism is only this: any measurement is truth plus noise; selecting on the measurement selects partly on the noise; the next measurement draws new noise, which is on average zero. No force pulls the winner down. The luck simply does not repeat.

How much falls back

The size of the effect depends on how much of the measurement was noise. If the correlation between one measurement and a repeat is `r`, then something observed `z` standard deviations above the mean is expected to come back at `r × z`. With `r = 0.5`, a model at +2 comes back at +1. With `r = 0.9`, at +1.8. With `r = 0`, when the measurement was all noise, at exactly the mean. You can estimate `r` by measuring twice; it is the test-retest reliability, and it is the number that says how much of a leaderboard is real.

The inflation of a chosen maximum also depends on how many candidates there were. The expected maximum of `k` standard normal draws:

How much the best of k identical models overstates itself



A hundred configurations scored on a validation set with a one-point standard error give a winner about two and a half points better than it is. A reported 91 is honestly nearer 88.5, and the only cure is a fresh set that nothing was selected on.

```
k =    10  →  +1.5 SD
k =    20  →  +1.9 SD
k =   100  →  +2.5 SD
k = 1,000  →  +3.2 SD
```

A hundred hyperparameter configurations, scored on a validation set with a standard error of 1 point, will produce a winner about 2.5 points better than it is. This is the **winner's curse**: the act of choosing the best guarantees that its score overstates it. The course `machine-learning-foundations` describes the validation set wearing out; this is the arithmetic of how fast.

Where it shows up

- **Hyperparameter search.** The best configuration's validation score is optimistic by roughly the table above. Report the fresh test score, not the

validation score you selected on.

- **Leaderboards.** The top entry on any public benchmark is, on average, worse than its rank suggests, and the gap between first and fifth is often less than the noise. Re-evaluation on a private set nearly always compresses the top.
- **"Our best red-teaming result."** The attack that worked best on this model was the luckiest of many tried; expect it to work less well on the next.
- **Interventions on the worst.** Retrain on the examples the model got most wrong and the next measurement improves on them regardless of whether the retraining helped, because the worst examples were partly the unluckiest. Attribute the improvement carefully.
- **Everywhere else.** The sophomore slump, the sports team that regresses after a record season, the patient who improves after a treatment started at their worst point. Same mechanism.

Three fixes, all cheap

1. **Measure the winner again on data it was not chosen on.** A held-out test set exists for this reason and no other. Its number is unbiased *because* nothing was selected on it.
2. **Shrink the estimate toward the mean.** If you cannot re-measure, multiply the winner's excess over the average by `r`, or subtract the expected maximum from the table. A reported 91 from the best of 100 configurations, with a 1-point standard error, is honestly about 88.5.
3. **Report how many were tried.** "Best of 3" and "best of 300" are different claims, and a score without that count cannot be read.

Watch it, once

```
import numpy as np
rng = np.random.default_rng(0)
true_acc, n, k = 0.80, 200, 20
val  = rng.binomial(n, true_acc, size=k) / n     # 20 identical
models, one val set each
best = val.argmax()
test = rng.binomial(n, true_acc) / n             # winner on
fresh items
print(val[best], test)                           # about 0.85,
then about 0.80
```

Run it a few times. The first number is reliably above the second, and no model was ever better than another.

What it is not

Regression to the mean is not overfitting in the model; the model never saw the validation items. It is not a hard test set; the fresh items were drawn the same way. It is not fraud, and it is not fixed by training longer. It is selection on noise, and it operates whenever anyone picks the best of anything by a measured score. The only defence is fresh data, and the only honest report includes the count of things you chose from.

BEFORE YOU MOVE ON

A team tries 100 hyperparameter configurations, picks the one with the best validation score (91.0 per cent), and gets 88.5 per cent on a fresh test set. What most likely happened?

A. The model overfit the validation set during training and memorised its items.

B. The test set happens to be harder than the validation set.

C. The best of 100 noisy scores was picked for its luck, which the fresh set removed.

D. The learning rate should have been re-tuned against the test set before the final run.

CASE STUDY · MODULE 6

Ninety-four per cent on fifty scripts

A state board examination cell in Bhopal, deciding whether to use an automatic marker on 1.9 million Class 10 English answer scripts.

The vendor's claim was specific and testable: on fifty scripts marked by both the system and a senior examiner, the two agreed on the grade 47 times. Ninety-four per cent. There were also no cases in the fifty where the system was more than one grade away from the examiner.

The examination cell had four weeks to decide. Marking 1.9 million scripts by hand takes six weeks and about eleven thousand examiner-days, and the results date is fixed by the academic calendar. The system, if it worked, would take four days and let examiners handle only the scripts it was unsure of.

An officer in the cell took the vendor's fifty-script table and did three pieces of arithmetic on it.

First, the interval. Forty-seven out of fifty is a proportion near the top of the range, where the textbook recipe of the estimate plus or minus 1.96 standard errors misbehaves. The Wilson interval, which solves for the true rates under which forty-seven of fifty would be unsurprising, gives 84 to 98 per cent. That is the honest reading of the vendor's evidence: somewhere between eighty-four and ninety-eight. At the bottom of that range, one script in six gets the wrong grade, which across 1.9 million scripts is three hundred thousand students.

Second, the zero. "No script more than one grade wrong in fifty" bounds that failure rate at roughly three divided by fifty, which is six per cent, with 95 per cent confidence. Six per cent of 1.9 million is 114,000 scripts. The zero was not evidence that the failure does not happen; it was evidence that it happens in under about one script in seventeen, which was not a reassuring sentence when written out in full.

Third, where the fifty came from. They had all been drawn from one school, marked by one examiner. Scripts from a single school share a teacher, a syllabus emphasis and a house style of answering, so they are not fifty indepen-

dent observations of the system's behaviour. The officer could not compute the design effect without more data, but the direction was certain: the effective sample size was below fifty, so the true interval was wider than 84 to 98, not narrower.

There was a fourth thing, which the vendor volunteered when asked. Twelve configurations had been tried on the fifty scripts, and the one reported was the best of the twelve. The best of twelve draws from a distribution sits, on average, about one and a half standard errors above the truth, and the standard error here is about three and a half points. So the 94 was selected partly for having been lucky, and would be expected to fall back by four or five points when measured on scripts it had not been chosen on.

The decision had a cost on both sides and both were large.

A proper pilot meant drawing four thousand scripts across sixty districts, marking each twice by hand as well as by machine, and comparing. That is about eight hundred examiner-days and six weeks, and six weeks past the results date means 1.9 million students waiting, admissions calendars slipping and questions in the assembly.

Going ahead on the vendor's number meant accepting a system whose true agreement rate the cell did not know within fourteen points, evaluated on one school, tuned on the same fifty scripts it was measured on, and applied to every child in the state.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The cell did neither of the two options as posed. It ran the system in parallel on the whole cohort, taking no grading decision from it, and had human examiners mark as usual; the system's output was recorded but not used. That cost the four days of compute and nothing else, and it produced 1.9 million

paired comparisons against human marks by the end of the normal marking cycle. Agreement on the full set came to 88.6 per cent, inside the Wilson interval and five points below the vendor's figure, in line with what the winner's curse and the single-school sample predicted. Scripts more than one grade apart were 2.1 per cent, or about forty thousand. On that evidence the cell approved the system for the following year in a narrower role: it marks first, every script it grades below a measured confidence goes to a human, and a five per cent random sample of the rest is checked regardless. The officer's note said the parallel run was not a compromise between the two options but a better third one, and that it existed because somebody wrote down what the fifty scripts could and could not support.

Why does the textbook interval, the estimate plus or minus 1.96 standard errors, break down on 47 out of 50?

The standard error is computed from the estimate rather than from the truth, and near the boundary the estimate is a poor stand-in. At p = 0.94 and n = 50 the recipe produces an upper bound above 1, which is impossible for a proportion, and its real coverage is well below the 95 per cent it claims. The Wilson interval solves instead for the true rates under which the observation would be plausible, which keeps it inside zero to one and keeps the coverage near its promise.

The vendor reported the best of twelve configurations. How much should the cell have discounted the 94?

The expected maximum of twelve draws sits about one and a half standard errors above the truth, and the standard error on fifty items near 0.94 is about three and a half points, so roughly five points of the 94 were selection on noise rather than quality. The full-cohort figure came in at 88.6, which is about that much lower. The general rule is that a score without the count of things it was chosen from cannot be read, and the only unbiased number comes from data nothing was selected on.

Fifty scripts came from one school. Why does that make the interval wider rather than merely different?

Every standard error formula in this module assumes independent observations. Scripts from one school share a teacher, an emphasis and a house style, so the system's behaviour on one predicts its behaviour on the next, and the fifty carry the information of rather fewer independent cases. The effective sample size is the row

count divided by one plus the correlation times one less than the cluster size, so any positive correlation shrinks it, and a smaller effective n means a wider interval than the one computed from fifty.

MODULE 6 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Why does the sample variance divide by n minus one rather than by n?

A. To make the estimate larger, since small samples understate spread by convention

B. Because distances are measured from the sample's own mean, which sits closer to the points than the true mean does

C. Because one observation is discarded to make the estimate independent

D. Because the square root of n minus one is the standard error

2 An evaluation set has 2,000 answers from 80 users, 25 each, with a within-user correlation of 0.5. What is the effective sample size?

A. 2,000, because each answer is a separate observation

B. 1,000, because half of the variation is shared

C. About 154, because the design effect is one plus 24 times 0.5

D. 80, since only the number of users can be counted

3 Three hundred independent test runs produced no failures at all. What can you claim?

A. The failure rate is zero, to the precision the test allows

B. The failure rate is below about one in three hundred thousand

C. Nothing at all: zero failures carries no information

D. The failure rate is below about one per cent, with 95 per cent confidence

4 A model scores 50 out of 50 and the textbook interval comes out as 1.00 to 1.00. What is wrong with it?

A. Nothing: fifty out of fifty genuinely establishes certainty

B. It should be 0.98 to 1.02, and the part above one is simply truncated

C. The standard error was computed from the estimate, and at p equal to one that gives zero width

D. It needs a t-distribution rather than a normal, which would widen it slightly

5 Why does an L1 penalty drive some weights to exactly zero while an L2 penalty never does?

A. L1's pull towards zero stays constant as the weight shrinks, while L2's fades away with it

B. L1 rounds small weights to zero in a separate step after each update

C. L2 is applied to the squared weights, which can never be zero

D. L1 is used with a larger penalty strength by convention, which finishes the job

6 You try 100 hyperparameter configurations on a validation set with a one-point standard error and report the best, at 91. What should you report instead?

A. 91, since it is the score the best configuration actually achieved

B. The average of the hundred, which is unbiased for the set as a whole

C. 91 with a note that a hundred were tried, plus a fresh test score, which will be near 88.5

D. 91 minus one standard error, which corrects for any selection effect

MODULE 7

Counting the cost: complexity and the arithmetic of scale

Growth rates and why n^2 is the one to fear, counting the floating-point operations in a layer and a training run, the bytes a model needs to exist and to train, why a GPU spends most of its time waiting for memory, and the arithmetic of hashing, graphs, nearest-neighbour search and back-of-the-envelope estimation.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Count the FLOPs of a forward pass and a training run from a parameter count, compute the memory a model needs at each precision and in training, say from arithmetic intensity whether a workload is bound by compute or by memory bandwidth, recognise a quadratic cost in a pipeline and name the sub-quadratic replacement, and estimate the time or cost of an ML job to within a factor of three before running it

58 · 8 min

Big-O, and the four growth rates you will meet

THE ONE THING TO KEEP

Big-O keeps only how work grows as input doubles, and the line that matters runs between n log n, which survives any dataset, and n^2, which stops working around a million items however fast the hardware.

The question Big-O answers

When the input doubles, what happens to the work? That is the only question Big-O notation answers. It ignores the constant factors, the hardware and the language, and keeps just the shape of the growth. That sounds like throwing away everything useful, and yet it is the single most predictive thing you can know about a piece of code before running it, because the shape decides whether a job that took a minute on a thousand rows takes an hour or a decade on a million.

Four shapes

Take `n` items and a machine doing a billion simple operations a second.

- **`O(log n)`**: each step halves the problem. Binary search in a sorted list; finding a key in a balanced tree. At `n = 10^9`, thirty steps. Effectively free at any scale.
- **`O(n)`**: look at everything once. Summing a column, scanning a file, one pass of a tokeniser. At `n = 10^9`, one second.
- **`O(n log n)`**: look at everything, a logarithmic number of times. Sorting. At `n = 10^9`, thirty seconds. This is the cost of "sort it first", and it is nearly always affordable.
- **`O(n²)`**: compare everything with everything. At `n = 10^6`, `10^12` operations: seventeen minutes. At `n = 10^9`, `10^18`: thirty-one years.

The gap between the third and fourth lines is the gap that matters. Anything up to `n log n` scales to any dataset you will meet. Anything quadratic scales to about a million and then stops, and the next lesson but one is about recognising it.

There is a fifth shape, `O(2^n)`, for problems where every subset must be tried. At `n = 60` it is thirty years; at `n = 100` it is longer than the universe. Exact solutions to such problems are not slow; they are unavailable, and the field's answer is to approximate.

Why the constants are dropped, and when they matter

`3n² + 200n + 5000` is `O(n²)` because for large `n` the first term is all that counts: at `n = 10^6`, the `3n²` is `3 × 10^12` and the rest is `2 × 10^8`, a rounding error. Big-O keeps the term that wins.

But the constants are real, and two `O(n)` implementations can differ by a factor of a hundred. A Python loop adding a million numbers takes about 50 milliseconds; `numpy.sum` on the same array takes half a millisecond. Same shape, hundred-fold constant. The rule: **Big-O tells you whether the approach can work; the constant tells you whether it will be pleasant.** Fix the shape first, because no constant rescues `n²` at a billion. Then fix the constant, because a hundredfold is a hundredfold.

The trap in one character

```
seen = []                    # list
for item in stream:
    if item in seen:         # O(n) scan every time
        continue
    seen.append(item)
```

`item in seen` on a list scans the whole list. Inside a loop over `n` items, that is `n` scans of up to `n` elements: `O(n²)`. Change one word:

```
seen = set()                 # set
```

and `item in seen` becomes a hash lookup, `O(1)`, and the loop is `O(n)`. On a million items the first version takes hours, the second a second. This is the most common accidental quadratic in data code, and the lesson on hashing explains what the set is doing that the list is not.

Memory has a shape too

The same notation describes space. A similarity matrix between `n` embeddings has `n²` entries. At `n = 10^5` and four bytes each, that is 40 GB, which does not fit in a laptop's RAM, and the code that builds it will not fail with a helpful message; it will swap for an hour and then be killed. A pairwise matrix is `O(n²)` memory whatever you do with it, and the fix is the same as for time: do not build it.

Amortised, and average

Two refinements you will meet. Appending to a Python list is `O(1)` **amortised**: occasionally the list must be copied to a bigger block, which costs `O(n)`, but that happens rarely enough that the average per append is constant. And hash lookups are `O(1)` **on average**; a pathological set of keys that all collide gives `O(n)`, which is why languages randomise their hash functions.

What Big-O cannot see

It cannot see the difference between reading from RAM and reading from disk, which is a factor of a thousand. It cannot see that a GPU does a thousand things at once, so an `O(n)` loop and an `O(n)` matrix operation take very different times. It cannot see cache. The lesson on arithmetic intensity is about exactly the costs Big-O is blind to. Use both: Big-O for whether the shape survives scale, and the arithmetic of bytes and FLOPs for how long it will actually take.

The habit

Before running anything on the full data, ask: when `n` doubles, does this double, or quadruple? Time it on a thousand rows and on two thousand. If the

second run takes four times as long, you have a quadratic, and now is the moment to fix it, not at a million.

BEFORE YOU MOVE ON

A deduplication script runs in 4 seconds on 10,000 records and 16 seconds on 20,000. What should you expect on a million records, and why?

A. About 400 seconds, since a million is a hundred times 10,000.

B. About 40,000 seconds: time quadruples when input doubles.

C. It cannot be predicted, because Big-O ignores constants and hardware.

D. About 800 seconds, allowing for some overhead on top of linear growth.

59 · 9 min

Counting the floating-point operations in a layer and a training run

THE ONE THING TO KEEP

A matrix multiply costs 2mnk FLOPs, so a forward pass costs about two FLOPs per parameter per token and a training step about six, which lets you compute that seven billion parameters on a trillion tokens is 4.2×10^{22} FLOPs before anyone tells you the price.

One entry, one dot product

Module 2 showed that every entry of a matrix product is a dot product between a row of the left matrix and a column of the right. A dot product of length `k` is `k` multiplications and `k` additions: `2k` floating-point operations, or FLOPs. Multiplying an `m × k` matrix by a `k × n` one produces `m × n` entries, so

```
FLOPs of (m × k) · (k × n) = 2 × m × n × k
```

That one line is most of the cost accounting of deep learning.

A linear layer taking a 4,096-dimensional vector to another 4,096-dimensional vector is a `4096 × 4096` matrix applied to one column:

```
2 × 4096 × 4096 × 1 = 33.5 million FLOPs
```

for one token. The matrix has 16.8 million parameters, and each parameter was used exactly once, for one multiplication and one addition. That generalises.

The rule: two FLOPs per parameter per token

In a forward pass, every weight in every linear layer touches each token once, multiplying and adding. So

```
forward FLOPs per token ≈ 2 × (number of parameters)
```

A seven-billion-parameter model spends about 14 billion FLOPs to process, or generate, one token. Embedding lookups cost nothing, being a table read, and the attention scores are an extra term the next lesson counts separately; for ordinary context lengths the rule above is within twenty per cent.

Training is three times a forward pass

Backpropagation, from module 3, computes two things per layer: the gradient with respect to the layer's input, to pass backward, and the gradient with respect to its weights. Each is a matrix multiply of the same size as the forward one. So the backward pass costs about twice the forward, and a full training step, forward plus backward, is about three times the forward:

```
training FLOPs ≈ 6 × parameters × tokens
```

Seven billion parameters on one trillion tokens:

```
6 × 7 × 10^9 × 10^12 = 4.2 × 10^22 FLOPs
```

That number is the one to carry. Everything about the cost of a training run follows from it and from how fast the hardware actually runs.

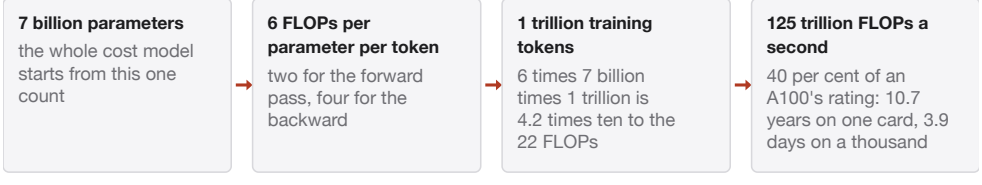
From FLOPs to time

A data-centre GPU of the A100 generation is rated at 312 trillion FLOPs per second in half precision. Nobody achieves that. Real training runs sustain around 40 per cent of the rating, for reasons the arithmetic-intensity lesson explains, so call it 125 trillion per second.

```
4.2 × 10^22 / 1.25 × 10^14 = 3.4 × 10^8 seconds ≈ 10.7 years on
one GPU
                                                ≈ 3.9 days on
1,000 GPUs
```

That is roughly what a seven-billion-parameter model on a trillion tokens costs, and the arithmetic is two multiplications long. The course `how-llms-work` turns it into money; the point here is that you can do it yourself, from a parameter count and a token count, in under a minute.

From a parameter count to days on a cluster



Two multiplications and a division. Every published training run can be checked this way in under a minute, from numbers that are always in the announcement, and long before anyone quotes a price.

On a laptop CPU, NumPy manages on the order of `10^11` FLOPs per second on a well-shaped matrix multiply. A forward pass of the seven-billion model, at 14 billion FLOPs per token, would take 0.14 seconds per token if FLOPs

were the only limit. They are not, and the third lesson from here says what is, but the FLOP count sets the floor.

Batches do not change the count

Processing a batch of 32 tokens through the same layer is `2 × 4096 × 4096 × 32` FLOPs: thirty-two times the work, for thirty-two times the tokens. The FLOPs per token are unchanged. What changes with batching is *efficiency*, how close the hardware gets to its rating, and that is a story about memory, not arithmetic.

Where the rule misleads

- **Attention is extra, and grows with context.** The scores between every pair of tokens add `4 × L × d` FLOPs per token per layer for context length `L` and width `d`. At `L = 4096` and `d = 4096` that is one sixth of the linear layers' cost; at `L = 24,576` it equals them. The next lesson has the arithmetic.
- **Mixture-of-experts models** have more parameters than they use per token. Count the parameters that a token actually passes through, not the total; a model with 400 billion parameters that routes each token through 40 billion costs like a 40-billion model per token.
- **FLOPs are not time.** A layer that does `10^9` FLOPs on `10^9` bytes of weights is waiting for memory, and no FLOP count sees that.
- **Rated FLOPs are for one shape.** The 312 trillion figure applies to large half-precision matrix multiplies. Elementwise operations, small matrices and single precision get a fraction of it.

Do it once

```
import numpy as np, time
A = np.random.rand(4096, 4096).astype(np.float32)
x = np.random.rand(4096, 32).astype(np.float32)
t = time.perf_counter(); A @ x; dt = time.perf_counter() - t
print(2 * 4096 * 4096 * 32 / dt / 1e9, "GFLOP/s")
```

Run it on whatever you have. The number you get is the constant that turns every FLOP count in this module into seconds on your own machine.

BEFORE YOU MOVE ON

A team plans to train a 3-billion-parameter model on 600 billion tokens. Using the standard rule, roughly how many FLOPs is that, and what is the rule made of?

A. About 1.8×10^{21}, from two FLOPs per parameter per token for the forward pass alone.

B. About 1.1×10^{22}, from six FLOPs per parameter per token.

C. About 3.6×10^{21}, from two FLOPs per parameter per token doubled for the backward pass.

D. It cannot be estimated without knowing the number of layers and the hidden width.

60 · 9 min

Anything pairwise is quadratic: distances, attention and deduplication

THE ONE THING TO KEEP

Any computation that compares every item with every other costs n^2 in time and memory, which is fine at ten thousand items and impossible at a million, and the fix is always the same: sort, bucket, block or approximate so that most pairs are never formed.

The shape to recognise

Every time a computation compares each item with every other item, the work is proportional to the square of the item count. It arrives in many disguises: a distance matrix, an attention layer, a deduplication pass, a join without an in-

dex, a nested loop over the same list. The disguises differ; the arithmetic does not, and it is the arithmetic that decides when the pipeline stops working.

Pairwise distances

Ten thousand embeddings of dimension 768, and you want every pairwise similarity. That is `n²/2 = 5 × 10^7` pairs, each a dot product of 768 multiply-adds:

```
5 × 10^7 × 768 × 2 ≈ 7.7 × 10^10 FLOPs   → under a second on a laptop
matrix: 10^8 entries × 4 bytes = 400 MB   → fits
```

Now a million embeddings:

```
5 × 10^11 pairs × 1536 ≈ 7.7 × 10^14 FLOPs   → about two hours at 10^11 FLOP/s
matrix: 10^12 entries × 4 bytes = 4 TB       → does not fit anywhere you own
```

A hundredfold more items, ten-thousandfold more work and memory. The FLOPs are survivable with patience; the matrix is not. Whatever you were going to do with all the pairs, you have to do it without materialising them.

Attention

A transformer layer computes, for every token, a score against every other token in the context. For context length `L` and model width `d`, the scores and the weighted sum together cost about `4 × L × d` FLOPs per token per layer. The layer's ordinary linear parts cost about `24 × d²` per token. Set them equal:

```
4 L d = 24 d²   →   L = 6 d
```

For `d = 4096` the crossover is at about 24,600 tokens. Below that, attention is the smaller cost; at `L = 4096` it is a sixth of the linear layers. Above it, at-

tention dominates, and at `L = 100,000` it is four times the rest. A model with a long context is a model whose cost per token grows with how much it has already read.

The memory is worse than the FLOPs. The score matrix is `L × L` per head. At `L = 32,768` in half precision that is `32768² × 2` bytes: 2.1 GB, per head, per layer. It never fits. The technique called flash attention exists to compute the softmax and the weighted sum in blocks without ever writing that matrix out, and the reason it was necessary is on this page. The course `how-llms-work` covers what was built; this is the number that forced it.

Deduplication

A corpus of a million documents, and you want to remove near-duplicates. Comparing every pair is `5 × 10^11` comparisons, each of which reads two documents. At a microsecond per comparison, six days. And the corpora people actually deduplicate are a thousand times larger, which makes it sixteen thousand years.

Nobody does that. The standard replacement hashes each document into a short signature such that similar documents share signature pieces with high probability, then only compares documents that landed in the same bucket. The hashing lesson explains the mechanism; the effect is to turn `n²` into roughly `n log n`, and sixteen thousand years into an afternoon.

Four ways out

The pattern is always the same: the cost comes from *comparing everything with everything*, so stop.

1. **Sort, then scan neighbours.** Exact duplicates, and anything with a total order, fall out of a sort in `n log n`.
2. **Bucket by a hash.** Put items that could match into the same bucket, then compare within buckets. Works when a cheap key predicts a match.
3. **Block by structure.** Compare only within the same date, the same language, the same customer. The pairs across blocks were never going to match.

4. **Accept approximation.** For nearest neighbours, the index structures in the lesson on search find the right answer nineteen times in twenty at a ten-thousandth of the cost.

Spotting it in code

```
for a in items:
    for b in items:          # same collection twice: quadratic
        if similar(a, b): ...
```

The give-away is the same collection in both loops, or a call like `distance_matrix`, `pairwise`, `cdist`, or a join with no key. None of these is wrong at a thousand items. All of them are wrong at a million, and the code gives no warning between the two.

The one honest quadratic

Sometimes you genuinely need all the pairs, because the answer is about the pairs: a full attention matrix for a short sequence, a correlation matrix among a hundred features, a confusion matrix among fifty classes. The rule is not "never quadratic". It is: know the `n`, square it, and multiply by the cost of one pair *before* you run it. If the product is under `10^10`, run it. If it is over `10^14`, you are choosing among the four ways out, and the only question is which.

BEFORE YOU MOVE ON

A transformer with width d = 4096 is run at a context length of 4,096 tokens and then at 100,000. What happens to the share of compute spent on attention scores?

A. It stays roughly constant, because attention is a fixed fraction of every layer.

B. It rises from about a sixth of the linear layers' cost to about four times it.

C. It rises in proportion to the context, so about 25 times more in absolute terms but the same share.

D. It falls, because a longer context amortises the score computation over more tokens.

61 · 9 min

Bytes per parameter, and the arithmetic of what fits on a card

THE ONE THING TO KEEP

Memory is parameters times bytes per parameter, so a seven-billion model is 14 GB in fp16 and 3.5 GB in int4, while training the same model holds about 16 bytes per parameter plus activations that scale with batch and context, which is why fine-tuning needs adapters to fit where inference already did.

A model is a number of bytes

Before a model can do anything it has to fit in memory, and the arithmetic for that is a multiplication: parameters times bytes per parameter. The bytes per parameter depend on the number format, which module 8 opens up; for now the four sizes are enough.

```
float32   4 bytes
bf16/fp16 2 bytes
int8      1 byte
int4      0.5 byte
```

A seven-billion-parameter model:

```
float32  28 GB
fp16     14 GB
int8      7 GB
int4    3.5 GB
```

Set those against the memory you have. A consumer graphics card has 8, 12, 16 or 24 GB. A laptop with a shared memory pool has 8 to 32 GB, of which the model can use perhaps three quarters. A phone has 6 to 12 GB shared with everything else. So the seven-billion model runs on an 8 GB card in int4, on a 16 GB card in fp16, and on a phone only in int4 with the operating system complaining. A seventy-billion model at int4 is 35 GB and needs either two cards or a machine built for it. None of this requires a benchmark; it is the multiplication.

Room to think: the KV cache

At inference a transformer keeps, for every token in the context, the key and value vectors of every layer, so that it need not recompute them for each new token. Per token that is

```
2 (key and value) × layers × width × bytes
= 2 × 32 × 4096 × 2 = 524,288 bytes ≈ 0.5 MB
```

for a typical seven-billion model in fp16. A 4,096-token context is therefore 2.1 GB on top of the weights, and a 32,000-token context is 17 GB, more than the weights themselves. Serving eight such conversations at once needs eight caches. When a hosted model charges more for long contexts, this is what it is charging for. The course `how-llms-work` covers the engineering around the cache; the size is this one product.

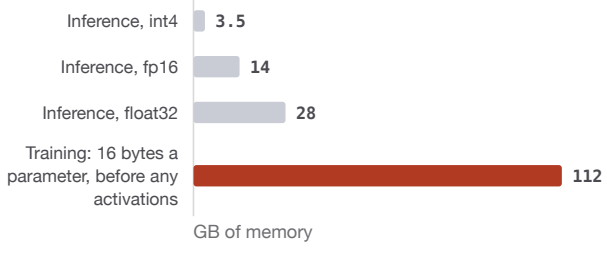
Training is far larger than the model

To train, you hold more than the weights. In the usual mixed-precision recipe with Adam, per parameter:

```
weights in fp16              2 bytes
master copy of weights fp32  4 bytes
gradient fp16                2 bytes
Adam first moment fp32       4 bytes
Adam second moment fp32      4 bytes
                            16 bytes per parameter
```

Seven billion parameters times sixteen is 112 GB, before a single activation is stored. That is why full fine-tuning of a seven-billion model needs a data-centre card or several consumer ones, though inference needs one.

A seven-billion-parameter model, four ways



Inference fits on a consumer card and training the same model does not, and almost all of the difference is optimiser state. Freeze the base and train adapters, and the sixteen bytes apply only to the half per cent of parameters you are training.

Then the activations. Backpropagation, from module 3, must keep every layer's inputs until the backward pass. For a transformer block the saved tensors come to about 34 bytes per hidden unit per token per layer in mixed precision, once the attention scores are handled in blocks. For a 2,048-token sequence, width 4,096, 32 layers:

```
2048 × 4096 × 32 × 34 ≈ 9.1 GB per sequence
```

A batch of eight is 73 GB. Activation memory scales with batch and sequence length, weight memory does not, and this is why the batch that fits is decided by activations. Gradient checkpointing throws most of them away and recom-

putes them during the backward pass, trading about a third more compute for a fivefold memory reduction; it exists because of this arithmetic.

Why adapters made fine-tuning affordable

Freeze the base model and train small added matrices, the LoRA recipe: the 16 bytes per parameter apply only to the adapter's parameters, perhaps 0.5 per cent of the total. The base sits at 2 bytes per parameter, read-only:

```
base 7B × 2 bytes         = 14 GB
adapters 35M × 16 bytes   = 0.56 GB
activations, batch 1      ≈ 9 GB (less with checkpointing)
```

About 24 GB, which is one high-end consumer card, against 120 GB or more for full fine-tuning. Quantise the frozen base to int4 and it is under 15 GB. The technique has a name and a paper, but the reason it works on a laptop is that most of the sixteen bytes were optimiser state, and optimiser state exists only for what you train.

The arithmetic of the data

The same habit applies to datasets. A text corpus of a billion tokens at two bytes per token id is 2 GB; as raw UTF-8 text at four characters a token, about 4 GB. An image dataset of a million 224 × 224 colour images as raw uint8 pixels is `10^6 × 224 × 224 × 3 = 150 GB`, which will not fit in RAM and must stream; as JPEGs, about a tenth of that on disk, decoded on the fly. Embeddings for ten million documents at 768 float32 dimensions are 30 GB, which is the reason the search lesson exists.

Three cautions

Peak memory is not average memory: a temporary buffer during a large matrix multiply, or the moment when an optimiser step holds old and new weights together, can add half again. Frameworks reserve memory in blocks, so the number in the monitoring tool is above what the arithmetic says. And a model that fits with 200 MB to spare will fail at the first long input. Leave a

fifth free, and when a thing does not fit, do the multiplication again before buying anything.

BEFORE YOU MOVE ON

A seven-billion-parameter model runs comfortably for inference on a 24 GB card in fp16. Full fine-tuning with Adam in mixed precision fails to fit. What is the main reason?

A. The training data must be loaded into GPU memory alongside the model.

B. The backward pass doubles the size of the weights, so 14 GB becomes 28 GB and overflows the card.

C. fp16 cannot represent gradients, so training must convert the whole model to float32, doubling it.

D. Training holds about 16 bytes per parameter of state, over 110 GB.

---

62 · 10 min

Why a GPU idles: memory-bound versus compute-bound

THE ONE THING TO KEEP

A computation runs at the slower of its compute and bandwidth limits, and its arithmetic intensity in FLOPs per byte decides which, so single-token generation at one FLOP per byte is bandwidth-bound and takes weights ÷ bandwidth seconds regardless of how fast the chip's arithmetic is.

Two ceilings, not one

A chip can only do so many floating-point operations per second: its compute ceiling. It can also only move so many bytes per second between memory and the units that do the arithmetic: its bandwidth ceiling. Every computation needs both, and it runs at the speed of whichever ceiling it hits first. FLOP

counts, from three lessons ago, see only the first. Most of the surprises in ML performance come from the second.

The numbers for an A100-class GPU:

```
compute:    312 × 10^12 FLOP/s (half precision)
bandwidth:    2 × 10^12 bytes/s
```

For a laptop CPU, roughly `10^11` FLOP/s and `5 × 10^10` bytes/s.

Arithmetic intensity

Divide the FLOPs a computation performs by the bytes it must move, and you get its **arithmetic intensity**, in FLOPs per byte. Compare it with the chip's **ridge point**, the compute ceiling divided by the bandwidth ceiling:

```
A100 ridge:   312 × 10^12 / 2 × 10^12 = 156 FLOP/byte
laptop ridge: 10^11 / 5 × 10^10        =   2 FLOP/byte
```

Below the ridge, the computation is **memory-bound**: the arithmetic units wait for data and the achieved FLOP/s is the intensity times the bandwidth. Above it, **compute-bound**: the chip is doing arithmetic as fast as it can. Plot achieved performance against intensity and you get a rising line that flattens at the ridge, shaped like a roof, which is why this is called the roofline model.

Generating one token is memory-bound

A model generating text produces one token at a time, and for each token it multiplies every weight matrix by a single vector. Each weight, 2 bytes in fp16, is read from memory once and used for one multiply and one add: 2 FLOPs per 2 bytes, an intensity of 1.

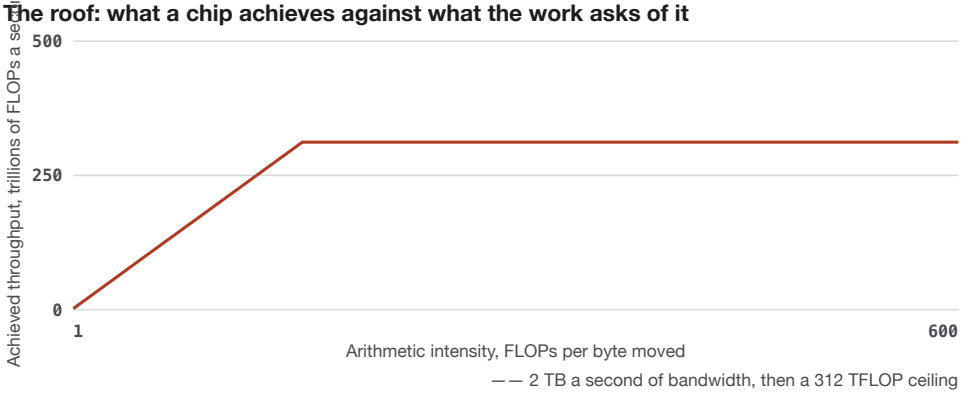
On the A100 that is 156 times below the ridge. The chip is doing arithmetic less than one per cent of the time. What it is doing is streaming 14 GB of weights past the arithmetic units, and the time that takes is set by bandwidth alone:

```
14 GB / 2 TB/s = 7 ms per token  →  at most about 140 tokens
per second
```

No FLOP count predicts that. On a laptop with 50 GB/s of bandwidth and a 4 GB int4 model, the same arithmetic gives 80 ms per token, about twelve tokens a second, which is what people see running a small model locally. Quantising to int4 speeds up generation not because the arithmetic is cheaper but because there are a quarter as many bytes to stream.

Batching is how you climb the roof

Process 64 sequences at once and each weight, still read once, is used 64 times: intensity 64. Process 256 and it is 256, above the ridge, and now the chip is compute-bound and the FLOP count starts to predict the time. This is why serving systems batch requests, why throughput per GPU rises almost linearly with batch size until the ridge, and why the latency for a single user is what it is regardless of how fast the chip's arithmetic is. Training uses batches of thousands of tokens against each weight and lives well above the ridge, which is why FLOP arithmetic works for training time and fails for single-stream inference.



Generating one token at a time sits at intensity 1, at the far left: the arithmetic units idle and the speed is 14 GB divided by 2 TB a second, about seven milliseconds a token. Batching 256 sequences moves the same work past the ridge, where a FLOP count finally predicts the time.

The operations that never climb

Elementwise work has an intensity near a quarter: adding two tensors reads 8 bytes and writes 4 to do one FLOP. Activation functions, layer normalisation, dropout, residual additions are all like this, and all are memory-bound on every chip ever made. A transformer layer that spends 5 per cent of its FLOPs on them can spend 40 per cent of its *time* on them. The fix is fusion: combine several elementwise steps into one kernel so the data is read once and written once. Every compiled-model toolchain does this, and the reason is on this page.

Why training reaches 40 per cent of the rating

The 312 trillion is for one ideal operation running continuously. A real step alternates large matrix multiplies, which approach the rating, with attention blocks, normalisations, optimiser updates and communication between GPUs, none of which do. The average over a step is the utilisation figure, and 35 to 45 per cent is a good result. Quoting a job's cost from the rating alone underestimates it by that factor.

Measure the roof you own

```
import numpy as np, time
n = 4096
A = np.random.rand(n, n).astype(np.float32); x =
np.random.rand(n, 1).astype(np.float32)
for cols in (1, 16, 256):
    X = np.repeat(x, cols, axis=1)
    t = time.perf_counter(); A @ X; dt = time.perf_counter() -
t
    print(cols, round(2 * n * n * cols / dt / 1e9), "GFLOP/s")
```

The one-column case is a matrix-vector product with intensity near 0.5 and runs at bandwidth speed. The 256-column case is compute-bound. The ratio between the two lines is your machine's version of the gap this lesson is about, and it will be somewhere between ten and a hundred.

What the model leaves out

The roofline has two ceilings; real chips have more. Cache levels give a small working set a higher effective bandwidth. Communication between GPUs in a cluster is a third, lower ceiling that bounds large training runs. And the model says nothing about latency between dependent operations, which matters for small models where launching a kernel takes longer than running it. It is still the first tool to reach for, because the single most common performance mistake is counting FLOPs for a computation that was never going to be limited by them.

BEFORE YOU MOVE ON

A team quantises a 14 GB fp16 model to 3.5 GB int4 and sees single-user generation speed rise about four-fold, though the int4 arithmetic is not faster. Why?

A. Quantisation reduces the FLOPs per token by four, so the compute ceiling allows more tokens.

B. Generation is bound by streaming the weights, and int4 has a quarter of the bytes.

C. The smaller model fits in cache, which is faster than main memory.

D. int4 lets the GPU process four tokens in parallel for every one it processed before.

63 · 9 min

Hashing, collisions, and the square-root law behind them

THE ONE THING TO KEEP

Hashing gives constant-time lookup by scattering inputs evenly, and the birthday bound says a repeat is likely after only √(2m) draws from m values, which is why a 32-bit key collides by 77,000 items and why Bloom filters, the hashing trick and locality-sensitive deduplication can each be sized from one formula.

Turning anything into a number

A hash function takes an input of any size and returns a fixed-size number, in a way that scatters inputs evenly across the possible outputs and makes similar inputs land far apart. Change one character of a document and its 64-bit hash changes in about half its bits. That property, plus speed, is why hashing sits under half the data structures in a machine-learning pipeline.

The hash table is the first. Store an item in the bucket its hash points to; to check membership, hash the query and look in that one bucket. The lookup does not depend on how many items are stored. That is the `O(1)` from the Big-O lesson, and it is why `item in some_set` is a thousand times faster than `item in some_list` at a million items: the set hashes, the list scans.

Collisions are a counting problem

Two different inputs can hash to the same value. With `n` items and `m` possible hash values, the number of pairs is about `n²/2` and each pair collides with probability `1/m`, so

```
expected collisions ≈ n² / 2m
P(at least one)     ≈ 1 − exp(−n² / 2m)
```

Collisions become likely when `n²/2m` approaches 1, that is when `n ≈ √(2m)`. This is the **birthday bound**: you need only the square root of the number of possible values before two items share one. In a room of 23 people, with 365 possible birthdays, the chance of a shared one is 50 per cent, and `1.18 × √365 = 22.5`.

Apply it to hash sizes:

```
32-bit hash, m = 4.3 × 10^9:   collision likely by n ≈ 77,000
items
64-bit hash, m = 1.8 × 10^19:  collision likely by n ≈ 5 × 10^9
items
128-bit:                        n ≈ 2 × 10^19
```

A million documents keyed by a 32-bit hash will certainly collide: `n²/2m = 116`, so `1 - e^(-116)` is 1 to every decimal place. The same million under a 64-bit hash collide with probability `2.7 × 10^-8`. If a deduplication or caching system uses a 32-bit key, it is silently merging unrelated records somewhere past the first hundred thousand. This is a real and common bug, and the arithmetic above finds it before the data does.

The hashing trick

When a vocabulary is enormous, every distinct URL, user id or n-gram, you can skip building a dictionary and hash each feature directly to one of `2^20` positions in a vector. Collisions are tolerated: two features sharing a slot add their values, which acts as a small amount of noise. How much? With a million features in a million slots, the count per slot is Poisson with mean 1: 37 per cent of slots are empty, 37 per cent hold one feature, 26 per cent hold two or more. A quarter of the features share a slot with something. If that is too much, use `2^24` slots and the sharing falls to about 6 per cent; the arithmetic tells you the trade before you train.

Bloom filters: membership in a tenth of the space

To remember which of a billion URLs you have already fetched, a set of the URLs themselves is tens of gigabytes. A Bloom filter keeps a bit array of `m`

bits and `k` hash functions; to add an item, set the `k` bits it hashes to; to query, check whether all `k` are set. It never says "absent" wrongly, but it can say "present" wrongly, with probability

```
P(false positive) ≈ (1 − e^(−kn/m))^k
```

At ten bits per item and `k = 7`, that is `(1 − e^(−0.7))^7 = 0.8` per cent. A billion URLs in 1.25 GB, with under one wrong answer in a hundred, and no way to list what is in it. The formula lets you buy exactly the error rate you can afford.

Locality-sensitive hashing: collisions on purpose

Ordinary hashing scatters similar inputs apart. For deduplication and nearest-neighbour search you want the opposite: similar inputs *should* collide. Locality-sensitive schemes are built so that the probability two items share a hash equals, or tracks, their similarity. The MinHash signature of a document, for instance, collides with another's with probability equal to the Jaccard overlap of their word sets. Bucket by a few signature bands and only compare within buckets, and the all-pairs deduplication from two lessons ago becomes near-linear. The pairs that never shared a bucket were, with high probability, never near-duplicates.

The reproducibility trap

Python randomises its string hashing per process, so that adversarial inputs cannot force collisions. A consequence: the iteration order of a set of strings differs between runs. A pipeline that builds a vocabulary by iterating a set will assign different ids on Tuesday than on Monday, and a model trained on one will read garbage from the other. Set `PYTHONHASHSEED=0` for reproducible runs, or, better, sort before assigning ids. The lesson on random numbers in module 8 has the rest of that story.

In code

```
import hashlib, math
h = int(hashlib.blake2b(b"some document", digest_size=8).hexdi-
gest(), 16)  # 64-bit
n, m = 1_000_000, 2**32
print(1 - math.exp(-n*n / (2*m)))     # 1.0: a 32-bit key will
collide
```

Use a 64-bit or 128-bit digest for anything you key on, size the Bloom filter from the formula, and read the birthday bound as the general warning it is: whenever something is drawn at random from `m` possibilities, a repeat is due after about `√m` draws, not `m`.

BEFORE YOU MOVE ON

A caching layer keys entries by a 32-bit hash of the request. It works in testing and starts returning wrong responses in production at a few hundred thousand distinct requests. What is happening?

A. The cache is evicting entries, so requests are being recomputed with a subtly different result.

B. The hash function is poorly chosen and clusters similar requests together.

C. Collisions should only appear near 4.3 billion entries, so the cause must be elsewhere.

D. Distinct requests collide: a repeat among 4.3 billion values is due by about 77,000 items.

64 · 9 min

A graph is a matrix, and its paths are its powers

THE ONE THING TO KEEP

A graph's adjacency matrix turns paths into matrix powers and random walks into repeated multiplication whose fixed point is the leading eigenvector, which is PageRank, and a graph neural network layer is simply that matrix averaging each node's neighbours before a learned linear map.

Nodes, edges, and a grid of zeros and ones

A graph is a set of things and a set of connections between them: pages and links, users and follows, papers and citations, atoms and bonds. Write the things down the side and across the top of a grid, put a 1 wherever two are connected and a 0 elsewhere, and you have the **adjacency matrix**. Everything in module 2 now applies to it.

Four nodes in a square, 1 joined to 2, 2 to 3, 3 to 4, 4 to 1:

```
    1  2  3  4
1 [ 0  1  0  1 ]
2 [ 1  0  1  0 ]
3 [ 0  1  0  1 ]
4 [ 1  0  1  0 ]
```

The row sums are the **degrees**, how many neighbours each node has: 2 each. For a directed graph the matrix need not be symmetric; a link from 1 to 2 puts a 1 in row 1, column 2, and nothing in row 2, column 1.

Paths are powers

Multiply the matrix by itself. Entry `(i, j)` of A^2 is the dot product of row `i` with column `j`, which counts the nodes `k` such that `i` connects to `k` and `k`

connects to `j`: the number of two-step paths from `i` to `j`. For the square:

```
      1  2  3  4
1  [  2  0  2  0 ]
2  [  0  2  0  2 ]
3  [  2  0  2  0 ]
4  [  0  2  0  2 ]
```

Two two-step paths from 1 to 3, via 2 and via 4; two from 1 back to itself, out and back along either edge; none from 1 to 2, which is an odd number of steps away. `A³` counts three-step paths, and in general `A^k` counts walks of length `k`. Whether two nodes are connected at all is whether any power has a non-zero entry between them. That is how "friends of friends" and "papers citing papers that cite this one" are computed: one matrix multiply per hop.

Random walks and the vector that stops changing

Divide each row by its degree and the matrix becomes a **transition matrix** `P`: entry `(i, j)` is the probability that a walker at `i` steps to `j`. Multiply a probability vector by `P` and you get where the walker is expected to be one step later. Do it again and again, and for most graphs the vector settles to one that no longer changes, the **stationary distribution**: the fraction of time a long random walk spends at each node.

Three pages: A links to B and C, B links to C, C links to A.

```
start:   (1/3, 1/3, 1/3)
step 1:  (0.333, 0.167, 0.500)
step 2:  (0.500, 0.167, 0.333)
step 3:  (0.333, 0.250, 0.417)
   ...
limit:   (0.400, 0.200, 0.400)
```

The walker spends 40 per cent of its time at A and at C and 20 per cent at B. That vector is the eigenvector of `P` with eigenvalue 1, module 2's "direction the matrix does not turn", and repeated multiplication finds it because every other component decays.

This is PageRank, before its one refinement: with probability 0.15 per step the walker jumps to a random page, which stops it being trapped in a page with no outgoing links or circling forever in a closed loop. The refinement changes the matrix slightly and the arithmetic not at all. Google's original ranking was the stationary distribution of a random walk on the web, computed by repeated multiplication of a matrix with a few billion rows.

Sparse, or it does not exist

A graph with a billion nodes has an adjacency matrix with `10^18` entries. Nobody stores that; as float32 it is four million terabytes. But the web has perhaps a hundred links per page, so only `10^11` entries are non-zero, a ten-millionth of the matrix. Store the edge list, `(from, to)` pairs, and multiply by iterating the edges: the cost is the number of edges, not the square of the nodes. Module 1's sparse-versus-dense distinction is the whole difference between a computation that takes a minute and one that cannot start.

A graph neural network is one matrix multiply

Give each node a feature vector and stack them as rows of a matrix `X`. Then `A X` replaces each node's features with the sum of its neighbours' features, and `D^(-1) A X`, dividing by degree, with the average. Follow that with a weight matrix and a nonlinearity:

```
X_next = ReLU( D^(-1) A X W )
```

That is a graph convolution layer: average your neighbours, then apply a learned linear map. Stack `k` of them and each node has seen information from `k` hops away, which is `A^k` again. The models that predict molecular properties, detect fraud rings and recommend friends are variations on this one line, and the adjacency matrix is the part that makes them graph models rather than ordinary ones.

Doing it yourself

```
import numpy as np
A = np.array([[0,1,0,1],[1,0,1,0],[0,1,0,1],[1,0,1,0]])
print(A @ A)                                  # two-step path
counts
P = A / A.sum(axis=1, keepdims=True)          # transition ma-
trix
v = np.ones(4) / 4
for _ in range(50): v = v @ P                 # random walk
print(v)                                      # stationary
distribution
```

The `networkx` library does all of this for real graphs, free, on a laptop, and its `pagerank` function is the loop above with the jump added. The value of having done it by hand once is that every graph algorithm you meet afterwards reads as a matrix operation you already know.

BEFORE YOU MOVE ON

For an adjacency matrix A, what does the entry in row 2, column 5 of A^3 count?

A. Whether node 2 and node 5 are connected by an edge, cubed.

B. The number of nodes within three hops of node 2.

C. The number of walks of exactly three steps from node 2 to node 5.

D. The probability that a random walker starting at 2 is at 5 after three steps.

65 · 10 min

Finding the closest vector among ten million: the arithmetic of approximate search

THE ONE THING TO KEEP

Exact nearest-neighbour search costs n × d per query and n × d × 4 bytes to hold, which is fine to about a hundred thousand vectors and impossible at ten million, so every method past that shrinks the vectors, searches a fraction of them or walks a graph, and each pays for its speed in measured recall.

The honest version: compare with everything

Given a query embedding and a store of `n` embeddings of dimension `d`, the exact way to find the nearest is to compute all `n` similarities. That is `n × d` multiply-adds per query, and the store occupies `n × d × 4` bytes in float32.

```
n = 100,000, d = 768:   1.5 × 10^8 FLOPs per query, 300 MB
n = 10,000,000:         1.5 × 10^10 FLOPs per query, 30 GB
```

At a hundred thousand, one query is about a millisecond on a laptop and the store fits in RAM. Brute force is the right answer, and building an index for it is wasted effort that also costs recall. The course `machine-learning-foundations` builds a working search at this scale with NumPy; nothing more is needed until the numbers change.

At ten million, one query is around 150 milliseconds on a CPU, tolerable for one user and hopeless for a thousand, and the store is 30 GB, which is beyond a laptop and beyond most graphics cards. Something has to give, and there are exactly four things that can.

Shrink the vectors

Drop dimensions. Module 2's singular values, or a model trained to put its important dimensions first, let you keep 256 of 768 with a modest loss in ranking quality: threefold less memory and compute. Then shrink each number. Float32 to int8 is fourfold at almost no loss. Product quantisation goes further: split the vector into sub-vectors, replace each with the index of its nearest entry in a small learned codebook, and store perhaps 64 bytes per vector instead of 3,072:

```
10^7 vectors × 64 bytes = 640 MB
```

The 30 GB store now fits in a phone. The similarity computed from codes is approximate, which is the first place recall is spent.

Search fewer of them

Partition the store into clusters, module 4's k-means will do, with about `√n` centres: 3,000 or so for ten million. A query is compared first with the 3,000 centres, then only with the vectors in the nearest few clusters. Probe 32 clusters and you examine about one per cent of the store:

```
3,000 centre comparisons + 1% of 10^7 = about 130,000 compar-
isons, against 10,000,000
```

Roughly a hundredfold faster. The cost is that the true nearest neighbour sometimes sits in a cluster you did not probe; at one per cent probed, expect to miss it a few times in a hundred, and to find it nineteen times in twenty. Probing more clusters buys recall with time, and the curve between them is the thing you tune.

Walk a graph instead

Build a graph in which each vector links to a few dozen near neighbours, with sparser long-range links layered on top. A query starts at an entry point and greedily hops to whichever neighbour is closest to it, descending the layers,

until no neighbour is closer. This is HNSW, and a query typically touches a few hundred to a thousand vectors:

```
~1,000 comparisons per query, against 10,000,000: ten-
thousandfold
```

The price is memory for the links, around 100 to 300 bytes per vector on top of the vector itself, and a build time that is itself a nearest-neighbour problem, hours for ten million on a laptop. Recall is typically above 95 per cent and adjustable with a search-width parameter. Most hosted vector databases run this.

What recall costs you

Every approximate method returns the true nearest neighbour some fraction of the time, and the fraction is measured, not guaranteed. A retrieval system at 95 per cent recall silently gives the wrong first passage to one query in twenty. Whether that matters depends on what sits downstream: a recommender barely notices; a system answering questions from a single retrieved document notices every time. Measure recall on your own queries against brute force on a sample, because the number in the library's documentation was measured on a different dataset.

The rule of thumb

```
under 10^5 vectors:   brute force, NumPy, no index
10^5 to 10^6:         brute force on a GPU or with int8; index
only if latency demands
above 10^6:           IVF or HNSW, with quantisation once memo-
ry bites
above 10^8:           quantise first, then partition, and ex-
pect to tune for weeks
```

Every one of those lines is a consequence of `n × d` and `n × d × 4`. Do the two multiplications for your own `n` before choosing a tool.

Free tools, and one more number

`faiss` on a CPU implements everything above and is free; `hnswlib` is a small, fast HNSW; NumPy is brute force. All run on a laptop. The one further number to know is that the query embedding must be produced by the same model, with the same normalisation, as the stored ones; module 1's cosine lesson explained why, and a store of unit vectors searched with an un-normalised query returns confidently wrong answers at any scale, with a perfect index.

BEFORE YOU MOVE ON

A team with 40,000 document embeddings sets up a hosted vector database with an HNSW index before building anything else. What does the arithmetic say?

A. Brute force is a millisecond per query and 120 MB; the index only costs recall.

B. The index is essential, because 40,000 pairwise comparisons per query is far too slow.

C. An index is needed for memory, because 40,000 vectors will not fit in a laptop's RAM.

D. HNSW gives exact results, so there is no downside beyond the setup cost.

66 · 9 min

Estimating anything to within a factor of three

THE ONE THING TO KEEP

Break a question into factors you can guess to within three, multiply, and track the powers of ten; with k independent guesses the result is uncertain by about 3^√k rather than 3^k, so six rough factors still land within an order of magnitude, which is usually enough to decide.

The skill under every lesson in this module

Enrico Fermi is said to have estimated the yield of the first atomic test by dropping scraps of paper as the blast wave passed and pacing out how far they travelled. The method is general: break a question you cannot answer into factors you can guess to within a factor of three, multiply them, and keep track of the powers of ten. The answer is rarely exact and almost always right about which order of magnitude you are in, and that is the thing you need to know before committing a week or a budget.

Worked: how many tokens is English Wikipedia?

```
articles              ≈ 7 million
words per article     ≈ 650 (most are short; a few are enor-
mous)
tokens per word       ≈ 1.3
7 × 10^6 × 650 × 1.3  ≈ 6 × 10^9 tokens
```

The published figures are around four to five billion. Within a factor of 1.5, from three guesses. And the answer is immediately useful: a trillion-token training set is two hundred Wikipedias, which tells you why training corpora are mostly not Wikipedia.

Worked: embedding ten million documents

```
documents                 10^7
tokens per document       ≈ 500
total tokens              5 × 10^9
embedding model           ≈ 10^8 parameters → 2 × 10^8 FLOPs
per token
total FLOPs               10^18
laptop, 10^11 FLOP/s      10^7 s ≈ 4 months
one GPU, 10^14 effective  10^4 s ≈ 3 hours
```

The plan changes at the fifth line. On a laptop this is not a job; it is a season. Rent a GPU for an afternoon, or use a smaller model, or embed a sample first. Any of these is a good decision, and all of them were invisible before the multiplication.

Worked: what a labelling project costs

```
items                 10,000
minutes per item      2
hours                 333
rate                  $20 per hour
cost                  ≈ $6,700
plus a second labeller on 20% for agreement:   ≈ $8,000
```

That is a number a manager can act on, produced in thirty seconds. The course `evaluating-ai` says why the second labeller is not optional; the arithmetic says what it costs.

Worked: the monthly bill for a retrieval assistant

```
users                     1,000
queries per user per day  20
tokens per query          ≈ 3,000 (retrieved passages dominate)
tokens per day            6 × 10^7
price                     ≈ $3 per million input tokens
daily cost                ≈ $180  →  monthly ≈ $5,400
```

If the true answer is $2,000 or $15,000, you were within a factor of three and the estimate did its job, which was to tell you this is thousands a month and not tens or hundreds of thousands. It also shows where the lever is: the 3,000 tokens per query. Halve the retrieved context and halve the bill; nothing else on the list is as easy to move.

Why factors of three are enough

Each guess is off by some factor, and the factors multiply. If every one of `k` guesses is uncertain by a factor of three, the product is *not* uncertain by `3^k`, because errors in different directions partly cancel. Working in logs, each error is a step of size `ln 3` in a random direction, and `k` random steps travel about `√k` steps in total. So the product is uncertain by about `3^√k`:

```
k = 4 factors:  3^2   ≈ 9
k = 6 factors:  3^2.4 ≈ 15
```

Six careless guesses give an answer within an order of magnitude. That is often enough to decide, and when it is not, the estimate tells you which factor to go and measure, since the one you were least sure of dominates the error.

Habits that make estimates good

- **Write the factors down.** An estimate in your head hides the guess that was wrong; on paper it is the line someone can correct.
- **Anchor on numbers you know.** A page is about 500 words. A token is about four characters. A GPU-hour is a few dollars. A person labels a few hundred simple items an hour. A laptop does `10^11` FLOP/s and a datacentre GPU a thousand times that. Ten such anchors cover most questions in this field.
- **Sanity-check the answer against something independent.** If the estimate says a job takes four months, ask whether anyone has done that job in an afternoon. If they have, one of your factors is wrong by a thousand, and it is worth finding which.

- **Estimate before, then measure after.** A pipeline that runs in a tenth of the estimated time is as interesting as one that takes ten times longer; both mean you misunderstood something, and the misunderstanding is usually the more valuable lesson.

Where it fails

Fermi estimation multiplies independent factors. It cannot see a factor you forgot entirely, and in this field the forgotten factor is usually a limit rather than a rate: the memory that does not fit, the rate limit on an API, the bandwidth ceiling from three lessons ago, the disagreement between two labellers. The estimate tells you the size of the job if nothing is in the way. Then look for what is in the way.

BEFORE YOU MOVE ON

An estimate is built from six factors, each of which you believe is right to within a factor of three either way. Roughly how uncertain is the product, and why?

A. About a factor of 3^6, around 700, since each factor's error multiplies in.

B. About a factor of 3, since the errors average out across six guesses.

C. About a factor of 18, since six factors each contribute three.

D. About a factor of 15: a $\sqrt{6}$-step random walk in log space.

CASE STUDY · MODULE 7

Eight million descriptions, and a server nobody needed to buy

A garment exporter in Tiruppur with eight million product descriptions accumulated across fourteen years, and a merchandiser who wants to know whether they have made a style before.

The question the business asked was ordinary. A buyer sends a technical sheet; a merchandiser needs to know within an hour whether the factory has made something close enough to quote from. Fourteen years of catalogues, spreadsheets and order files held about eight million descriptions, and the only way to search them was a name-based lookup that failed on anything phrased differently.

A consultant proposed a search system and quoted for a graphics-card server, about four and a quarter lakh rupees, plus setup. The owner asked her nephew, who was in the second year of a computer science degree, to check the number before she signed.

He did four multiplications and then a fifth.

Deduplication first, because half the eight million were near-copies of each other and the consultant's plan compared every description with every other one to find them. Eight million items make about thirty-two trillion pairs. At a microsecond a pair that is three hundred and seventy days. It was not a slow step in the plan; it was a step that would never finish, and nothing in the proposal said so. The standard replacement hashes each description into a short signature so that similar ones land in the same bucket, and only compares within buckets, which turns the work from the square of the item count into something close to linear. It runs in an afternoon on a laptop.

Embedding next. Eight million descriptions of about sixty tokens each is 480 million tokens. A small free embedding model of 33 million parameters costs about two FLOPs per parameter per token, so 66 million FLOPs per token, and the whole corpus is about 3.2 times ten to the sixteenth FLOPs. His laptop manages around a hundred billion FLOPs a second, which is 3.7 days of con-

tinuous running. A rented graphics card, at a hundred times that in practice, does it in under an hour for a few hundred rupees.

Memory. Eight million vectors of 384 numbers at four bytes each is 12.3 GB, which does not fit in the 8 GB laptop the office had. Stored as one-byte integers instead, it is 3.1 GB, which does.

Query time. Comparing a query against all eight million vectors is about six billion FLOPs, or sixty milliseconds on that laptop. For one merchandiser at a desk that is instant. For the catalogue website the owner also wanted to connect, at forty queries a second, it is not.

The fifth calculation was the one with a real decision in it. Partitioning the vectors into about two thousand eight hundred clusters and searching only the sixteen nearest clusters brings a query from eight million comparisons down to about forty-eight thousand: a hundred and sixty times faster, comfortably fast enough for the website. The cost is recall. The true nearest description sometimes sits in a cluster the search did not open, and at that setting it will be missed roughly six times in a hundred.

Six in a hundred is not an abstract number in a garment factory. A missed match means the merchandiser is told the style is new when it is not, and the factory re-develops a sample it already has: about eleven thousand rupees and nine days, each time it happens.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The nephew wrote the five calculations on one sheet and the owner did not buy the server. The work ran on the office laptop with one afternoon of rented graphics-card time for the embedding step, at a total cost under four thousand rupees. Deduplication by signature hashing cut eight million descriptions to 3.4 million distinct ones, which made every later number smaller than the

ones he had estimated. For the merchandisers at their desks the system searches all 3.4 million exactly, because sixty milliseconds a query needs no index and an index would only lose recall. The clustered index runs only behind the website, where the speed is needed and a missed match costs a visitor a scroll rather than a factory a sample. He measured the recall himself against exact search on a sample of two hundred real queries and got 94.5 per cent, close enough to his estimate that he trusted the rest of the arithmetic.

The consultant's deduplication step compared every description with every other. Why is that not merely slow?

Comparing every item with every other is quadratic: eight million items give about thirty-two trillion pairs, which at a microsecond each is over a year, and the pairwise matrix would not fit in any storage the business owns. The fix is never to form most of the pairs. Hashing each description into a signature so that similar ones share a bucket, then comparing only within buckets, turns the work into something close to linear, and the pairs that never shared a bucket were with high probability never near-duplicates anyway.

Why did he use exact search for the merchandisers and an approximate index only for the website?

Because the two have different costs for a miss. An exact scan of 3.4 million vectors takes tens of milliseconds, which is instant for one person at a desk, and it returns the true nearest match every time. An approximate index is a hundred and sixty times faster and misses the true nearest a few times in a hundred. On the website the speed is needed and a miss costs a visitor a scroll; for a merchandiser a miss costs a re-developed sample. You buy speed with recall only where the recall is worth less than the speed.

He estimated 3.7 days on a laptop and under an hour on a rented card, and neither number was measured. Why was that good enough to decide on?

Because the decision only needed the order of magnitude. The point of the estimate was to distinguish an afternoon from a season, and a factor of three either way does not change which of those it is. A Fermi estimate built from a few factors each good to within a factor of three lands within about an order of magnitude, which is usually enough to choose. When it is not, the estimate at least names the factor you were least sure of, and that is the one worth going and measuring.

MODULE 7 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A loop checks whether each item is already in a Python list of seen items. On a million items it takes hours; changing the list to a set makes it seconds. Why?

A. A set stores less data, so it fits in memory while the list does not

B. Membership in a list scans it, so the loop is quadratic; a set hashes, so it is linear

C. Sets are implemented in C while lists are implemented in Python

D. The set removes duplicates automatically, so the loop runs fewer times

2 A 13-billion-parameter model is trained on 2 trillion tokens. Roughly how many floating-point operations is that?

A. About 1.6 times ten to the twenty-third

B. About 2.6 times ten to the twenty-second, counting the forward pass only

C. About 2.6 times ten to the thirteenth, at one operation per parameter per token

D. It cannot be estimated without knowing the architecture and the hardware

3 Quantising a model from fp16 to int4 makes single-token generation about four times faster. What is the mechanism?

A. Integer arithmetic is four times faster than half-precision arithmetic on all hardware
B. Smaller weights let the model skip layers whose contribution rounds to zero
C. The model fits in cache, which removes the memory access entirely
D. There are a quarter as many bytes to stream, and generation is limited by bandwidth not arithmetic

4 A deduplication system keys a million documents by a 32-bit hash. How many unrelated documents will it silently merge?

A. None: 32 bits gives four billion values for only a million documents
B. About one, since a million is a quarter of a per cent of four billion
C. Around a hundred, because collisions become likely at the square root of the space
D. All of them, since a 32-bit hash has only 32 distinct values

5 In a transformer of width 4,096, attention costs about four times L times d FLOPs per token per layer, and the linear parts about 24 times d squared. Where do they cost the same?

A. At 4,096 tokens, where the context length equals the width
B. At about 24,600 tokens, which is six times the width
C. At 1,024 tokens, a quarter of the width
D. They are always equal, since both scale with the width

6 You estimate a cost by multiplying six factors, each of which you believe to within a factor of three. Roughly how uncertain is the product?

A. A factor of 729, which is three to the sixth
B. A factor of 18, which is three times six
C. A factor of about 15, since the errors partly cancel
D. A factor of 3, since the largest single error dominates

MODULE 8

Numbers inside a machine

What a float32 actually stores and where its seven digits run out, the half-precision formats and why training keeps a full-precision copy, cancellation and summation order, integer overflow, quantisation as rounding with a scale, condition numbers, the variance that travels through a layer, random numbers that are not, checking a gradient numerically, and reading NaN and Inf for what they are.

BY THE END OF THIS MODULE YOU CAN

Predict from the bit layout of float32, fp16 and bf16 which computation will overflow, underflow or lose its digits, explain why a long sum drifts and how to fix it, quantise a weight to int8 by hand and say what an outlier does to the rest of the tensor, derive the initialisation scale that keeps variance constant through a layer, and diagnose a NaN in training back to the operation that produced it

67 · 10 min

What a float32 actually stores, and where its seven digits run out

THE ONE THING TO KEEP

A float32 is a sign, an 8-bit exponent and a 24-bit mantissa, giving about seven decimal digits and a range near 10^±38, so its spacing grows with the number, an integer counter stalls at 2^24, and equality between computed floats is an accident of rounding rather than a fact.

Thirty-two bits, three fields

A float32 is scientific notation in binary. Its 32 bits are split into a sign (1 bit), an exponent (8 bits) and a fraction called the mantissa (23 bits):

```
value = (−1)^sign × 1.mantissa × 2^(exponent − 127)
```

The leading 1 is not stored, because in binary the first significant digit is always 1, so the mantissa carries 24 bits of precision for the price of 23. Everything about how floats behave follows from those three field widths.

Precision: about seven digits

Twenty-four binary digits is `24 × log10(2) = 7.2` decimal digits. The gap between 1 and the next representable number is `2^−23 = 1.19 × 10^−7`, called machine epsilon. Any number is stored to a relative accuracy of about half that, six parts in a hundred million. The absolute gap between neighbouring floats grows with the number: near 1 it is `10^−7`, near 1,000 it is `10^−4`, near a million it is `0.06`, and past `2^24 = 16,777,216` it is 2.

The 32 bits of a float32, and what each field costs you

Sign, 1 bit	positive or negative
Exponent, 8 bits	range about ten to the plus or minus 38; exp overflows at 88.7
Mantissa, 23 bits stored and 24 used	7.2 decimal digits: the gap at 1 is 0.00000012, at 16.8 million it is 2

Every surprise in this module comes from those three widths. Seven digits is why a variance computed by subtraction fails; the growing gap is why a float32 counter stalls at 16,777,216 and why adding 1 to a hundred million does nothing at all.

That last fact bites. A float32 counter incremented by 1 stops at 16,777,216, because `16,777,217` is not representable and the addition rounds back down:

```
import numpy as np
x = np.float32(16_777_216)
print(x + np.float32(1))       # 16777216.0
```

A running total of tokens, steps or samples kept in float32 silently stalls at seventeen million. Keep counters as integers.

Range: about 10^±38

Eight exponent bits give exponents from −126 to +127, so the largest float32 is about `3.4 × 10^38` and the smallest normal one about `1.2 × 10^−38`. Below that, the format gives up precision to reach a little further, down to `1.4 × 10^−45`, after which the number is zero. Module 5 used the top of that range: `exp(x)` overflows to infinity at `x = 88.7`. The bottom matters too. A probability of `10^−40` is representable only as a denormal with a few bits of precision, and the product of two probabilities of `10^−25` each is exactly zero. This is the concrete reason everything probabilistic is done in log space.

Why 0.1 + 0.2 is famous, and why the format matters

One tenth is not representable in binary, any more than one third is in decimal. Stored as float64, `0.1` is really `0.1000000000000000055`, and

```
0.1 + 0.2 == 0.3          # False in float64:
0.30000000000000004
```

In float32 both sides happen to round to the same number, `0.300000012`, and the comparison is True. The lesson is not about 0.3. It is that equality between floats is an accident of rounding, and a test like `if loss == 0.0` or `assert a == b` on computed values will pass or fail depending on the format, the order of operations, and the hardware. Compare with a tolerance:

```
np.isclose(a, b, rtol=1e-5, atol=1e-8)
```

and choose the tolerances from the format: `rtol` near `10^-5` for float32, `10^-12` for float64.

The gap you should picture

Representable floats are not evenly spaced. They are dense near zero and sparse far from it, with the same *number* of floats between 1 and 2 as between 1,024 and 2,048. This has one consequence you will meet constantly: adding a small number to a large one loses the small one. `10^8 + 1` in float32 is `10^8`, because the gap between floats near `10^8` is 8. The next lesson but one is about the damage that does.

Doubles, and when to use them

Float64 has 11 exponent bits and 52 mantissa bits: about 16 digits and a range of `10^±308`. NumPy and Python use it by default; deep-learning frameworks use float32 by default because it halves memory and the arithmetic units are faster. The right rule: float32 for the model, float64 for anything that accumulates, checks, or is compared. A loss averaged over a million batches, a gradient check, a statistic computed by subtraction: float64. A weight matrix: float32 or smaller, and the next lesson says how much smaller.

Reading the bits yourself

```
import struct
bits = struct.unpack(">I", struct.pack(">f", 0.1))[0]
print(f"{bits:032b}")     # 0 01111011 10011001100110011001101
```

Sign 0, exponent `01111011 = 123`, so `2^(123 - 127) = 2^-4 = 1/16`, mantissa `1.6000000238...`, and `1.6 / 16 = 0.1000000015`. The repeating `1001` in the mantissa is the binary expansion of one tenth being cut off at 23 bits, and that cut is where the `0.0000000015` came from. Once you have seen it, every rounding surprise in the rest of this module has an address.

BEFORE YOU MOVE ON

A training script keeps a running count of processed tokens in a float32 tensor. After a long run the count is stuck at 16,777,216 though tokens are still arriving. What is happening?

A. The tensor has overflowed its maximum value and wrapped around, like an integer would.

B. Float32 cannot represent integers above about seven digits at all.

C. The data loader has stopped delivering tokens and the count is correct.

D. Above 2^{24} the float spacing is 2, so adding 1 rounds away.

68 · 9 min

bf16, fp16, and why training keeps a float32 copy

THE ONE THING TO KEEP

fp16 keeps three digits and a range to 65,504 while bf16 keeps two digits and float32's full range, and because a small update added to a half-precision weight rounds away entirely, training keeps a float32 master copy of every weight and accumulates matrix products in float32.

Two ways to spend sixteen bits

Halve the width of a float and you must give up either range or precision. The two sixteen-bit formats make opposite choices.

format	sign	exponent	mantissa	max value	epsilon	dig-its
fp16	1	5	10	65,504	9.8e−4	~3
bf16	1	8	7	3.4e38	7.8e−3	~2
float32	1	8	23	3.4e38	1.2e−7	~7

fp16 keeps ten bits of mantissa, three decimal digits, and spends only five on the exponent. Its largest value is 65,504 and its smallest normal value is `6 × 10^−5`. **bf16**, "brain float", keeps float32's eight exponent bits and therefore its whole range, and pays with a seven-bit mantissa: two and a bit decimal digits. A bf16 number is a float32 with its bottom sixteen bits cut off, which makes conversion a truncation rather than a computation.

Module 7 established why either is wanted: half the bytes per parameter, twice the throughput on hardware built for them. The question is which half of the information to keep.

Why range beats precision for training

Gradients span an enormous range. Early layers of a deep network routinely see gradients of `10^-8`, and activations after a bad step can reach `10^5`. In fp16 the first underflows to zero and the second overflows to infinity, and neither problem announces itself except as a model that does not learn or a loss that becomes NaN. Module 5's overflow point moves from `exp(88.7)` to `exp(11.1)`: a logit of 12 is enough to produce infinity in fp16.

bf16 has none of these problems, because its range is float32's. What it lacks is precision, and it turns out that a weight update does not need much: a gradient known to two digits still points the right way. So bf16 became the default for training wherever the hardware supports it, and fp16 survives mainly for inference on older cards.

Sixteen bits, spent two ways

fp16: 5 exponent bits, 10 mantissa bits

- Largest value 65,504
- Smallest normal value 0.00006
- About three decimal digits
- A logit of 12 already overflows exp
- Training survives only with dynamic loss scaling

bf16: 8 exponent bits, 7 mantissa bits

- Largest value 3.4 followed by 38 zeros
- Exactly float32's range
- About two decimal digits
- Converting from float32 is a truncation
- The default for training wherever hardware allows

A weight update does not need many digits; it needs to be representable at all, which is why range beat precision. It is also why the optimiser keeps a float32 master copy: near 1 the gap between bf16 numbers is 0.0078, so an update of 0.0001 added there vanishes entirely.

Loss scaling: the fp16 workaround

Where fp16 must be used for training, the underflow is handled by multiplying the loss by a large constant, say 1,024, before the backward pass. Every gradient is then 1,024 times larger, which lifts the `10^-8` gradients into fp16's range. The optimiser divides by the same constant before applying the update. If any gradient overflows to infinity, the step is skipped and the scale is halved; if a run of steps succeeds, the scale is doubled. This is *dynamic loss scaling*, it is in every mixed-precision training loop that uses fp16, and its entire purpose is to fit a range of `10^13` into a format that holds `10^9`.

Why there is a float32 copy

Here is the fact that explains the 16 bytes per parameter in module 7. Suppose a weight is 1.0 and the update is `0.0001`. In bf16, with its two digits, `1.0 + 0.0001 = 1.0`: the update is smaller than the gap between neighbouring bf16 numbers near 1, which is `0.0078`. The weight never changes. In fp16 the gap near 1 is `0.001`, and the same update is lost.

Small updates are the normal case late in training, when the learning rate has decayed. So the optimiser keeps a **master copy** of every weight in float32, applies the update there, where the gap near 1 is `10^−7`, and then rounds the result to bf16 for the next forward pass. The half-precision weights are a working copy; the float32 ones are the truth. Remove the master copy and training stalls once the updates fall below the half-precision gap, which is typically within the first few thousand steps.

Where the precision is actually spent

Mixed precision is precise about what runs in what:

- **Matrix multiplies**: inputs in bf16, but the products *accumulated* in float32 inside the hardware, then rounded once at the end. Accumulating a 4,096-term dot product in bf16 would lose most of it, for the reasons the next lesson gives.
- **Softmax, layer normalisation, the loss**: computed in float32, because they involve sums of many terms and subtractions of nearly equal ones.
- **Weights and activations in memory**: bf16.
- **Master weights, optimiser moments**: float32.

The framework's `autocast` does this routing for you. What it cannot do is make a computation that you wrote in bf16 by hand, an accumulating sum or a variance, come out right.

Inference is more forgiving

No updates, no gradients, no accumulation across steps. Weights and activations in fp16 or bf16 lose almost nothing measurable, which is why every served model is at most half precision and usually smaller, and why the quantisation lesson can go further still. The one inference computation that still wants float32 is the final softmax over a large vocabulary, where bf16's two digits turn a distribution over 100,000 tokens into something visibly lumpy.

Seeing the gap

```
import torch
w = torch.tensor(1.0, dtype=torch.bfloat16)
print(w + 0.001)              # 1.0     the update vanished
w32 = torch.tensor(1.0)
print((w32 + 0.001).to(torch.bfloat16))   # 1.0 again after
rounding, but 1.001 was kept in w32
```

The second line is the master copy in miniature: the float32 value remembers what the bf16 one cannot, and after ten such updates it has moved to 1.01, which bf16 can then represent.

BEFORE YOU MOVE ON

A team trains in pure bf16 with no float32 master weights. Training looks normal for a few thousand steps, then the loss flattens and the weights stop changing though the gradients are non-zero. Why?

A. The updates fell below the gap between adjacent bf16 values.

B. The learning rate schedule has decayed to zero.

C. bf16 has too small a range and the gradients have underflowed to zero.

D. bf16 matrix multiplies accumulate error until the gradient direction is meaningless.

69 · 9 min

Catastrophic cancellation, and why a sum depends on its order

THE ONE THING TO KEEP

Subtracting nearly equal floats leaves only their rounding error, and adding a small float to a large one loses it, so variances must subtract the mean before squaring, long sums must accumulate in float64 or pairwise, and because addition is not associative a GPU's varying reduction order makes identical runs diverge.

Subtracting two nearly equal numbers

Every float carries about seven digits, in float32, of which the leading ones are the most trustworthy. Subtract two numbers that agree in their first five digits and the result has two digits left: the five that agreed have cancelled, and what remains is the rounding error of the inputs, promoted to the front. This is **catastrophic cancellation**, and it is the mechanism behind most numerical bugs that are not overflow.

The cleanest example is a variance computed by the textbook shortcut, `E[x²] − E[x]²`. A thousand values near 10,000 with a true spread of 1:

```
E[x²]  ≈ 100,000,001   (seven digits: stored as 100,000,000)
E[x]²  ≈ 100,000,000
variance = E[x²] − E[x]² → 8.0 in float32, or 0.0, or negative
```

Both terms are about `10^8` and are each known to about 8 units. Their difference is supposed to be 1, which is smaller than the error in either. In one run the float32 answer was 8; on other data it comes out as zero or as a negative number, from which `sqrt` returns NaN. The remedy is to subtract the mean first and then square, so that the numbers being squared are near 1 rather than near 10,000, and there is nothing large to cancel. Or use float64, whose

sixteen digits leave nine to spare. Every serious library uses one of these; every hand-written `mean(x2) - mean(x)2` in a notebook is a bug waiting for data with a large mean.

Adding a small number to a large one

The mirror image: `10^8 + 1` in float32 is `10^8`, because the spacing between floats near `10^8` is 8, and 1 is below half of it. Nothing dramatic happens; the 1 is simply gone.

Now do that ten million times. Sum `0.1` repeatedly in float32:

```
after 1,000,000 additions:   100,958   (should be 100,000: 1%
high)
after 10,000,000 additions: 1,087,937  (should be 1,000,000: 9%
high)
```

The running total grows; the increment does not; and once the total passes a few million, each `0.1` is rounded to whatever the local spacing allows, which is not 0.1. The error is systematic, not random, because the rounding of `0.1` at each magnitude is always in the same direction. A loss averaged by accumulating `total += batch_loss` in float32 over a long run drifts in exactly this way, and the reported average is wrong by a per cent or more, silently.

Three fixes, in order of effort

1. **Accumulate in float64.** One word changes and the error falls by nine orders of magnitude. This is the right default for anything that sums across a training run.
2. **Sum pairwise.** Add neighbours, then add the pairs, and so on, so that every addition combines numbers of similar size. NumPy's `sum` does this, which is why `np.sum` of ten million `0.1`s gives `1,000,000.0` while a Python loop gives `1,087,937`. The error grows like `log n` instead of `n`.
3. **Compensated summation.** Kahan's algorithm keeps a second variable holding the part that was rounded off at each step and adds it back. Four lines; the float32 sum of a million `0.1`s comes out as exactly `100,000.0`.

```
def kahan_sum(xs):
    s, c = 0.0, 0.0
    for x in xs:
        y = x - c
        t = s + y
        c = (t - s) - y      # the rounding error, recovered
        s = t
    return s
```

Addition is not associative

In exact arithmetic `(a + b) + c = a + (b + c)`. In floating point it does not hold:

```
(1e16 + 1) - 1e16 = 0.0      (the 1 was lost in the first addi-
tion)
1e16 + (1 - 1e16) = 1.0
```

Same three numbers, different order, different answer. Most of the time the differences are in the last digit and nobody notices. But a GPU adds up the contributions to a gradient from thousands of threads, and the order in which they arrive is not fixed from one run to the next. So the same code on the same data with the same seed gives a gradient that differs in its last bits, the weights diverge by a little, and after ten thousand steps the two runs are visibly different models. This is the arithmetic beneath the observation in `how-llms-work` that temperature zero is not deterministic: the sampling is fixed; the addition order is not.

Frameworks offer deterministic modes that force a fixed reduction order at a cost in speed. Use them when you need bitwise reproducibility, and know that without them, two runs agreeing to three digits is agreement.

Where else it hides

- **Softmax without the max subtraction** (module 5): `exp` of large numbers, then a division of huge by huge.

- **Numerical derivatives** (later in this module): `f(x + h) − f(x)` is a subtraction of nearly equal numbers by construction.
- **Normalising a vector of tiny values**: dividing by a norm that was itself rounded to zero.
- **Any formula with a minus sign between two terms of similar size.** Rearranging algebra to avoid the subtraction, as with the variance above, is a legitimate and common numerical technique.

The habit: when a computed number is the small difference of large ones, ask how many digits the large ones had, and subtract that from seven. What is left is what you know.

BEFORE YOU MOVE ON

A monitoring script computes the variance of a latency column as mean(x^2) − mean(x)^2 in float32 and occasionally reports a negative variance. The latencies are around 3,000 ms with a spread of a few ms. What is wrong?

A. The mean of the squares should be divided by n − 1 rather than n.

B. Some latencies are negative due to clock skew, which makes the variance negative.

C. float32 cannot hold numbers as large as 9 million, so the squared terms overflow.

D. Both means are near 9×10^6; their small difference is lost to rounding.

70 · 8 min

Integers, overflow, and the token id that would not fit

THE ONE THING TO KEEP

Fixed-width integers are exact inside their range and wrap silently outside it, so counts, offsets and products of sizes belong in int64, pixels and ids belong in the smallest type that fits, and a result of exactly 2,147,483,647, 255 or 16,777,216 is a limit rather than a value.

Integers are exact, until they are not

An integer type stores a whole number exactly, with no rounding, within a fixed range set by its width:

```
uint8    0 to 255
int8     −128 to 127
int16    −32,768 to 32,767
int32    −2,147,483,647 to 2,147,483,647   (about ±2.1 billion)
int64    about ±9.2 × 10^18
```

Inside the range, every operation is exact, which is why counters, indices, ids and pixel values are integers. Outside it, the behaviour is not an error. In NumPy and most compiled code, an int32 that passes 2,147,483,647 **wraps** to −2,147,483,648, and the program continues with a negative number where a large positive one should be. Python's own `int` grows without limit, so this never happens in plain Python; it happens the moment data enters a NumPy array, a PyTorch tensor, a database column or a file format, all of which have fixed widths.

Where 2.1 billion arrives

Two billion sounds large until you count tokens. A training corpus of a trillion tokens has positions up to `10^12`, and an int32 offset into it wraps at the second billion. A dataset index built with `np.arange(len(corpus))` on a platform whose default integer is 32 bits, which Windows NumPy was until recently, produced negative positions past 2.1 billion tokens, and negative indices in NumPy are legal: they count from the end. The bug did not crash; it read the wrong tokens.

The same limit appears in file sizes (a 32-bit length field caps at 2 GB, the reason older formats could not store a file larger than that), in a `sum` over an int32 column of byte counts, and in any `count × size` multiplication where both are int32: `50,000 × 50,000 = 2.5 × 10^9` overflows before it is assigned to the int64 you were about to store it in.

```
import numpy as np
a = np.int32(50_000)
print(a * a)         # -1794967296, with at most a warning
```

The fix is a word: `int64` for anything that counts tokens, bytes, rows or products of sizes. Vocabulary ids are safe in int32, since no vocabulary approaches two billion; positions, offsets and totals are not.

Small integers are where the memory goes

At the other end, small integer types are how large data fits. An image is stored as uint8: 256 levels per channel, one byte per value, and a 224 × 224 colour image is 150 KB rather than the 600 KB of float32. A million such images are 150 GB, which is the figure module 7 used. Token ids fit in uint16 when the vocabulary is under 65,536 and in int32 otherwise, and a billion tokens are 2 or 4 GB accordingly. Labels for a thousand classes fit in uint16.

The habit that goes with this: convert to float only at the point of computation, not at the point of loading. A pipeline that decodes images to float32 on disk has quadrupled its storage for nothing.

uint8 arithmetic is a trap of its own

Subtracting two uint8 pixels, `100 − 150`, does not give −50. It wraps to 206. Adding two, `200 + 100`, gives 44. Image code that computes a difference between frames in uint8 produces bright artefacts where it should show darkness, and the code looks correct. Cast to int16 or float32 before any subtraction, and back to uint8, with clipping, only for storage.

Products of small integers

Quantised inference, the next lesson, multiplies int8 weights by int8 activations. Each product can be as large as `127 × 127 = 16,129`, which does not fit in int8 or int16 once you add four thousand of them together. The accumulation is therefore done in int32, and the result rescaled. If you ever write such a loop yourself, the accumulator's width is the first thing to get right; the hardware's integer matrix units get it right for you.

Integers inside floats

The previous lessons showed that float32 represents integers exactly only up to `2^24 = 16.8` million, and float64 up to `2^53 = 9 × 10^15`. A record id above 16.8 million stored in a float32 column is rounded to an even number, and two distinct customers become one. This happens when a CSV with a missing value in an id column is loaded, because a column with a NaN cannot be integer, so the loader silently promotes it to float. Check `dtype` on id columns after loading, and store ids as strings if the values can be large.

Booleans and bits

A boolean is one byte in NumPy, not one bit; a mask over a billion tokens is a gigabyte. Packed bit arrays, `np.packbits`, are eight times smaller and are what Bloom filters and large masks use. And a `sum` over a boolean array counts the Trues, which is the fastest way to count anything in NumPy, provided the accumulator is not a uint8: `np.sum(mask, dtype=np.int64)` if there are more than 255 of them.

The rule

Count in int64. Store in the smallest type the values fit, and know what that type's range is. Never subtract in uint8. And when a number is exactly `2,147,483,647`, `−2,147,483,648`, `65,535`, `255` or `16,777,216`, you are looking at a limit, not a measurement.

BEFORE YOU MOVE ON

A frame-differencing script subtracts consecutive video frames stored as uint8 arrays and shows bright speckles where the scene is static. What is happening?

A. Sensor noise is being amplified by the subtraction.

B. uint8 cannot store the difference, so the array was silently promoted to float and rescaled.

C. A slightly darker pixel gives a negative difference that wraps to near 255.

D. The frames are misaligned by one pixel, producing edge artefacts across the image.

---

71 · 10 min

Quantisation is rounding with a scale: int8, int4 and the outlier problem

THE ONE THING TO KEEP

Quantisation stores each weight as round(w / scale) with the scale set by the largest value in its group, so the error is half a step for everyone and a single outlier can round an entire tensor to zero, which is why scales are kept per channel or per block of 128.

The whole idea in one line

To store a float weight as an 8-bit integer, pick a scale and round:

```
q = round(w / scale)          an integer from −127 to 127
w ≈ q × scale                 to recover it
```

The scale is chosen so that the largest weight in the tensor lands on 127: `scale = max|w| / 127`. That is symmetric int8 quantisation, and it is most of what the word means. The course `running-models-yourself` covers the tools; this lesson is the arithmetic those tools perform.

One weight, by hand

A tensor whose largest weight is 0.02 has `scale = 0.02 / 127 = 0.000157`. A weight of 0.0034:

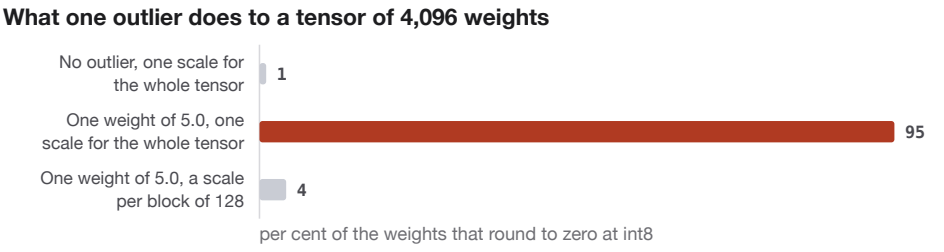
```
q = round(0.0034 / 0.000157) = round(21.6) = 22
w ≈ 22 × 0.000157 = 0.003465
error = 0.000065, about 2 per cent of the weight
```

The error of any weight is at most half a scale step, `0.0000787` here. For weights near the maximum that is 0.4 per cent; for weights a tenth the size it is 4 per cent; for a weight of 0.00005 it rounds to zero and the error is 100 per cent. Quantisation is exact for the large values and brutal for the small ones, because the step size is set by the largest.

The outlier problem

Now suppose that same tensor contains one weight of 5.0. The scale becomes `5 / 127 = 0.039`. The weight of 0.0034 is `round(0.086) = 0`. So is every weight below 0.02: the whole tensor apart from the outlier rounds to zero or to one step. One number has destroyed a million.

This is not hypothetical. Large language models develop a few channels with activations tens or hundreds of times larger than the rest, and naive int8 quantisation of those activations wrecks the model while the weights alone quantise fine. Three responses, all arithmetic:



The step size is set by the largest value in the group, so one weight a hundred times the rest rounds almost everything else to zero. A scale per channel or per block of 128 confines the damage to the block the outlier sits in, for about an eighth of a bit per weight.

1. **Per-channel scales.** One scale per row or column instead of per tensor, so the outlier only sets the step for its own row. Costs one float per row; nearly free.
2. **Group-wise scales.** One scale per block of 64 or 128 weights. An int4 model with a 16-bit scale per 128 weights spends `4 + 16/128 = 4.125` bits per weight, and the outlier's damage is confined to its block of 128.
3. **Move the outlier.** Multiply an activation channel by a constant and divide the matching weight row by the same constant; the product is unchanged, and the difficulty has been shifted from the activations, which are hard to quantise, to the weights, which are easy. This is what the smoothing methods do.

Asymmetric, and the zero-point

If the values are not centred, an activation after a ReLU is never negative, half the integer range is wasted. The asymmetric form adds an offset:

```
q = round(w / scale) + zero_point
```

so that the minimum maps to 0 and the maximum to 255 in uint8. The zero-point is chosen so that a real zero maps exactly onto an integer, because padding and ReLU produce exact zeros in quantity and rounding them to something else adds a bias everywhere.

Four bits

Int4 has sixteen levels. With a per-tensor scale that is hopeless; with group-wise scales it works for weights, and int4 weights with fp16 activations is the standard recipe for running a large model on a small device. The arithmetic of why: weights are static, so they can be quantised carefully, once, with all the tricks above; activations change with every input and must be quantised on the fly, which is why they are usually left at higher precision.

The error introduced is measurable and it is not zero. A model at int4 typically loses a few tenths of a point of perplexity relative to fp16; at int3 it loses several points and at int2 it stops working. The loss is larger for smaller models, because they have fewer redundant weights to absorb the noise, and larger in the layers nearest the output. Anyone claiming int4 is free has not measured it on their own task; the honest statement is that it is cheap and the price is known.

Why it is faster, and when it is not

Module 7's arithmetic-intensity lesson said single-token generation is bound by streaming the weights. Int4 weights are a quarter of the bytes, so generation is up to four times faster, before any change in the arithmetic. For large-batch throughput, which is compute-bound, the gain depends on whether the hardware has integer matrix units; if it must convert int4 back to fp16 before multiplying, there is no speed-up at all, only memory saved.

Doing it yourself

```
import numpy as np
w = np.random.randn(4096).astype(np.float32) * 0.01
w[17] = 5.0                                  # one outlier
scale = np.abs(w).max() / 127
q = np.round(w / scale).astype(np.int8)
print((q == 0).mean())                       # fraction of
weights that became zero: nearly all
row_scale = np.abs(w[:2048]).max() / 127     # a group without
the outlier
q2 = np.round(w[:2048] / row_scale)
print((q2 == 0).mean())                      # a few per cent
```

Run it, then delete the outlier line and run it again. The difference between the two fractions is the whole reason quantisation papers exist.

BEFORE YOU MOVE ON

A tensor of weights mostly between −0.02 and 0.02 also contains one weight of 5.0. It is quantised to int8 with a single scale for the whole tensor. What happens to the typical weight?

A. It is stored with about 2 per cent error, as the arithmetic for a 0.0034 weight showed.

B. It rounds to zero: the step is now 0.039 and the weight is under half of that.

C. The outlier is clipped to 127 × scale and the rest are unaffected.

D. The int8 range overflows and the values wrap to negative numbers.

72 · 9 min

The condition number, and how many digits a problem throws away

THE ONE THING TO KEEP

The condition number σ_max/σ_min says how many digits a problem throws away, about log10(κ) of them, and the usual source of a large one is unscaled or correlated features, which is why standardising columns fixes both a numerically meaningless fit and a slow gradient descent at once.

Some problems amplify error

Give a computation an input that is wrong in its seventh digit. Some computations return an answer wrong in the seventh digit; others return an answer wrong in the first. The second kind are **ill-conditioned**, and the condition number is how much they amplify the error. Nothing about the algorithm or the hardware changes this. It is a property of the problem.

For a matrix, module 2's singular values give the number directly:

```
κ = σ_max / σ_min
```

the ratio of the largest stretch the matrix applies to the smallest. A matrix that stretches one direction by 1,000 and another by 0.001 has `κ = 10^6`. The identity has `κ = 1`. A singular matrix, which flattens some direction entirely, has `κ = ∞`.

The rule of thumb

Solving a linear system, fitting a least-squares model, or inverting a matrix with condition number `κ` loses about `log10(κ)` digits of accuracy. Float32 has seven; float64 has sixteen.

```
κ = 10^2   float32 keeps 5 digits   float64 keeps 14
κ = 10^6   float32 keeps 1 digit    float64 keeps 10
κ = 10^8   float32 keeps nothing    float64 keeps 8
κ = 10^16  float64 keeps nothing
```

The Hilbert matrix, whose entries are `1/(i + j − 1)`, is the classic case: the 5 × 5 version has `κ ≈ 5 × 10^5`, the 10 × 10 version `κ ≈ 10^13`. Solving a 10 × 10 Hilbert system in float64 gives three correct digits; in float32, garbage with no warning. Module 2's advice to solve rather than invert stands, but it does not rescue a problem this badly conditioned. Nothing does, except changing the problem.

Where it comes from in practice: scale

The commonest ill-conditioned matrix in machine learning is the one you build from unscaled features. Take a regression with one feature in the range 0 to 1 and another in the range 0 to 1,000,000. The matrix `XᵀX` from module 2's normal equations has one direction stretched by about `10^12` relative to the other: `κ ≈ 10^12`. Solving it in float64 leaves four digits. Solving it in float32 leaves none, and the coefficient on the small feature comes out as noise.

Standardise both features to mean 0 and spread 1, and `κ` drops to the ratio set by their correlation, usually under 10. Same data, same model, twelve orders of magnitude of conditioning recovered by subtracting a mean and dividing by a spread. This is the numerical reason behind the feature-scaling advice in `machine-learning-foundations`; the statistical reasons are there, and this is the arithmetic one.

Polynomial features are worse. Fitting `x, x², …, x^10` on `x` between 0 and 1,000 gives columns ranging from `10^3` to `10^30`, a condition number beyond float64's reach, and coefficients that are pure rounding error. Rescale `x` to `[-1, 1]` first, or use an orthogonal polynomial basis, which is the same fix by another name.

Gradient descent feels it too

Module 3's curvature lesson showed that plain gradient descent zigzags when the loss is much more curved in one direction than another, and that the ratio of curvatures sets the number of steps. That ratio *is* the condition number of the Hessian. A quadratic loss with `κ = 10^4` needs on the order of `10^4` steps of plain gradient descent to converge to a fixed accuracy; with `κ = 10` it needs about ten. Feature scaling, normalisation layers, and adaptive optimisers such as Adam are all, from this angle, ways of reducing the condition number the optimiser sees. The numerical and the optimisation problems are one problem.

Correlated features

Two columns that are nearly copies of each other, a price in dollars and the same price in cents, make `XᵀX` nearly singular, because the matrix cannot tell them apart. The condition number rises without limit as the correlation approaches 1, and the fitted coefficients become huge, opposite in sign and meaningless, while the predictions stay fine. The L2 penalty of module 6 adds `λ` to every singular value, which bounds `κ` by `σ_max / λ` and is the numerical reason ridge regression was invented, before anyone described it as a prior.

Computing it

```
import numpy as np
X = np.column_stack([np.random.rand(1000), 1e6 *
np.random.rand(1000)])
print(np.linalg.cond(X))                # about 1e6 for X,
1e12 for X.T @ X
Xs = (X - X.mean(0)) / X.std(0)
print(np.linalg.cond(Xs))               # a few
```

`np.linalg.cond` computes the singular-value ratio. Print it for any matrix you are about to solve with, and read `log10` of the result as the digits you are about to lose. Above `10^8` in float32 or `10^15` in float64, the answer the solver returns is not an approximation of the truth; it is an artefact of rounding, and the solver will not say so.

BEFORE YOU MOVE ON

A least-squares fit with a feature in metres and another in nanometres gives coefficients that change wildly when one row of data is edited, though predictions look fine. What is going on?

A. The two features are nearly collinear, so the model cannot separate them.

B. Least squares is unstable and gradient descent should be used instead.

C. The edited row is an outlier that squared error weights heavily.

D. A 10^9 scale ratio gives κ near 10^{18}; the coefficients are noise.

73 · 9 min

How variance travels through a layer, and the initialisation that follows

THE ONE THING TO KEEP

Because a unit's output is a sum of fan_in independent products, its variance is fan_in × σ_w² × σ_x², so a weight spread of √(2/fan_in) keeps activations level through every ReLU layer while 0.05 or 0.01 explodes or vanishes them within twenty.

One unit's output is a sum

A unit in a linear layer computes `y = Σ w_i x_i` over `fan_in` inputs. Treat the weights and inputs as random: each `w_i` drawn with variance `σ_w²`, each `x_i` with variance `σ_x²`, all independent and centred on zero. Module 4's rule that variances of independent terms add, and that the variance of a product of independent centred variables is the product of the variances, gives

```
Var(y) = fan_in × σ_w² × σ_x²
```

That one line decides whether a deep network can be trained from a random start.

What happens over twenty layers

Suppose you initialise every weight with a "small random" standard deviation of 0.05, a common default in hand-written code, and the layer has `fan_in = 1024`. Then

```
Var(y) / Var(x) = 1024 × 0.0025 = 2.56
```

Each layer multiplies the variance by 2.56. A ReLU after it zeroes half the values and roughly halves the variance, so the net factor is about 1.28 per layer. After twenty layers the activations have variance `1.28^20 ≈ 139` times the input's. After sixty, `10^6`. The final logits are enormous, the softmax saturates, and module 5's gradient `p − y` is either 0 or ±1 everywhere: the model begins its life confidently wrong and with no useful gradient.

Try 0.01 instead:

```
1024 × 0.0001 × 0.5 = 0.051 per layer
```

After twenty layers the variance is `10^−26`. Every activation is effectively zero, every gradient with it, and by forty layers the values are below anything float32 can represent. The model does not learn because there is no signal to learn from. Module 3's vanishing gradients, from the other end.

The value that keeps it level

Set the per-layer factor to 1:

```
fan_in × σ_w² × ½ = 1    →    σ_w = √(2 / fan_in)
```

For `fan_in = 1024`, `σ_w = 0.044`. Between the two failures above sits a single number that keeps the variance constant through every layer, and it depends only on the width of the layer. That is the He, or Kaiming, initialisation, and the `2` is the ReLU's halving. Without a ReLU, or with tanh, the factor is `1/fan_in`, the Xavier or LeCun rule. `machine-learning-foundations` states these rules; this is where they come from, and the derivation is short enough that you could rediscover it if you forgot.

The backward pass has the same arithmetic

Gradients flowing backward through the layer are also sums, over `fan_out` terms this time, so the same argument gives `σ_w = √(2 / fan_out)` to keep gradient variance level. The two demands conflict unless `fan_in = fan_out`; the usual compromise averages them, and the disagreement is small enough

not to matter for layers of similar widths. It matters for a layer that maps 4,096 dimensions to 32: initialise for the forward pass and the backward variance is off by a factor of 128 in that layer alone.

Why normalisation layers were the other answer

Initialisation gets the variance right at step zero. Training moves the weights, and nothing then holds the variance in place. Layer normalisation and batch normalisation re-standardise the activations at every step, subtracting the mean and dividing by the spread, which enforces `Var = 1` by construction regardless of what the weights have become. They are the runtime version of the calculation above, and the reason very deep networks became trainable. Residual connections are a third answer: adding the input straight through means the variance grows only additively per block rather than multiplicatively, and `20 × small` is survivable where `1.28^20` is not.

Check it in three lines

```
import numpy as np
rng = np.random.default_rng(0)
x = rng.standard_normal((1024, 512))                    # 512
examples, 1024 wide
for sigma in (0.05, 0.01, np.sqrt(2 / 1024)):
    h = x.copy()
    for _ in range(20):
        W = rng.standard_normal((1024, 1024)) * sigma
        h = np.maximum(W @ h, 0)                        # ReLU
    print(sigma, h.std())                               # huge,
~0, ~1
```

The first run's standard deviation is in the tens, the second's is around `10^-13`, and the third's is about 1. Twenty lines of NumPy reproduce the whole argument, and the same script, pointed at your own architecture, tells you whether its initialisation is sane before you spend a GPU-hour finding out.

What the argument assumed

Independence between weights and inputs, which holds at initialisation and fails once training correlates them. Zero means, which a ReLU breaks by producing only non-negative outputs, so the halving is approximate. And a linear layer; convolutions have `fan_in = channels × kernel area`, attention has its own scaling, the `1/√d` in module 1's dot-product lesson, which is the same variance argument applied to a dot product of two `d`-dimensional vectors. The idea survives all of these. The constant changes.

BEFORE YOU MOVE ON

A 40-layer ReLU network with layers 2,048 wide is initialised with weights of standard deviation 0.05. At the first step the loss is enormous and the gradients are all zero or ±1. Why?

A. Each layer multiplies the variance by about 2.6, and after 40 layers the softmax has saturated.

B. The learning rate is too high and the first update has already blown up the weights.

C. 0.05 is too small for such a wide layer, so the activations have vanished and the gradients underflowed.

D. ReLU zeroes half the activations, which halves the information at every layer until none remains.

74 · 9 min

Random numbers that are not, and what a seed does and does not fix

THE ONE THING TO KEEP

A pseudo-random generator is a deterministic recipe started from a seed, so seeding makes shuffles, masks and splits repeat, but it cannot fix a GPU's reduction order, a split that shifts when data grow, or Python's per-process string hashing, each of which needs its own remedy.

Determinism wearing a disguise

A computer cannot produce a random number. What it produces is a **pseudo-random** sequence: a fixed recipe that turns one state into the next, chosen so that the output passes every statistical test for randomness while being entirely determined by where it started. The starting state is the **seed**. Same seed, same recipe, same sequence, forever.

The simplest recipe, a linear congruential generator, is one line: `state = (a × state + c) mod m`. Modern generators, the Mersenne Twister behind older NumPy and Python's `random`, and the PCG64 behind `np.random.default_rng`, are more elaborate but no less deterministic. Their period, the number of outputs before the sequence repeats, is around `2^19937` for the Twister and `2^128` for PCG64. Neither will repeat in your lifetime. But a sequence that never repeats is not the same as one that cannot be predicted; if the seed is known, every number is known.

What a seed buys

Set it, and a shuffle, a dropout mask, an initialisation and a train-test split are the same on every run. That is what makes a bug reproducible and a comparison fair. Set it in every library that draws numbers, because each keeps its own state:

```
import random, numpy as np, torch
random.seed(0)
np.random.seed(0)                          # legacy global state
rng = np.random.default_rng(0)             # preferred: an explic-
it generator you pass around
torch.manual_seed(0)                       # CPU and all GPUs
```

The explicit generator is better than the global seed, because two parts of a program drawing from one global stream change each other's numbers when one of them adds a draw. Give each component its own generator, seeded from a master seed, and adding a draw in one place no longer changes the shuffle in another.

What a seed does not buy

Bitwise identical training on a GPU. The previous lessons showed that floating-point addition depends on order and that a GPU adds in whatever order its threads arrive. The seed fixes the dropout mask and the initial weights; it does not fix the reduction order. Two runs with the same seed diverge in their last bits at the first step and in their third digit within a few thousand. `torch.use_deterministic_algorithms(True)` forces fixed-order kernels, at a cost in speed and with some operations refusing to run; it is the price of exact reproducibility, and it is sometimes worth paying.

Identical results across machines. Different hardware, driver versions and library builds choose different kernels with different orders. Reproducibility across machines is a matter of agreeing to three digits, not thirty-two bits.

A stable split when the data grow. Shuffling with a seed and taking the first 80 per cent as training is reproducible only until a row is added, after which every row's position shifts and yesterday's test examples leak into today's training set. The fix uses module 7's hashing: assign each row by a hash of its id, `hash(id) mod 100 < 80`, so that a row's split depends on the row alone. New rows land in either split; old rows never move.

Python's string hashing. It is randomised per process by design, which changes the order of iteration over a set of strings from one run to the next. A

vocabulary built by iterating a set is different on every run unless `PYTHONHASHSEED` is fixed or the set is sorted first. This is the single most common cause of "the same code produces a different model".

Turning uniform into anything

A generator produces uniform numbers between 0 and 1. Every other distribution is manufactured from those. The general method is the **inverse cumulative distribution**: if `F` is the cumulative distribution you want, then `F^(-1)(u)` for uniform `u` has that distribution. For the exponential with rate `λ`:

```
x = −ln(u) / λ
```

For the normal, the inverse has no closed form, and the Box-Muller transform takes two uniforms and returns two independent normals:

```
z₁ = √(−2 ln u₁) × cos(2π u₂)
z₂ = √(−2 ln u₁) × sin(2π u₂)
```

For a discrete distribution, a token sampled from a softmax, the method is the same: compute the cumulative sums, draw `u`, and pick the first token whose cumulative probability exceeds `u`. That is what `torch.multinomial` does, and it is why sampling a token costs a scan over the vocabulary rather than a lookup.

Randomness you did not ask for

`DataLoader` workers each get a seed derived from the main one plus their worker id; if you seed NumPy inside a worker without that id, all workers produce the same augmentations. Dropout at evaluation time, if the model was never switched out of training mode, makes the same input give different outputs. Sampling with `temperature > 0` is random by intent. And any code that reads the clock, a network, or the filesystem order has a source of variation no seed touches.

The habit

Seed everything, pass generators explicitly, hash for splits, sort before assigning ids, and expect two GPU runs to agree to three digits rather than exactly. When they agree to fewer, look for the nondeterminism, and when they agree exactly, check that the second run actually ran.

BEFORE YOU MOVE ON

A team seeds Python, NumPy and PyTorch identically and trains twice on the same GPU, yet the two final models differ in the third decimal place of every weight. What is the most likely cause?

A. One of the libraries was not actually seeded, so the initial weights differed.

B. The pseudo-random generator has a period shorter than the number of draws in a training run and wrapped around.

C. GPU sums run in a varying order, and the rounding differences compound.

D. Dropout produces genuinely random masks that no seed can control.

75 · 9 min

Checking a gradient with finite differences, and the step size that lies

THE ONE THING TO KEEP

A two-sided finite difference has truncation error of order h^2 and rounding error of order ε/h, so the best step is about $\varepsilon^{(1/3)}$, which in float32 leaves only five correct digits and makes gradient checks meaningful only in float64, where a relative error above 10^{-3} means the gradient is wrong.

The one test every hand-written gradient needs

Module 3 defined the derivative as the slope between two points as the gap shrinks, and noted that you can compute a usable one with two evaluations and a small number. That is also the way to check a gradient you derived by hand, or a custom backward function, or an autograd system you do not fully trust. Nudge one parameter, see how the loss moves, compare with what the gradient claimed:

```
numerical = ( L(θ + h) − L(θ − h) ) / (2h)
```

The two-sided form, with `+h` and `−h`, is worth the second evaluation: its error shrinks with `h²` rather than `h`, which the next section shows matters.

Why h cannot be tiny

The obvious instinct is to make `h` as small as possible, because the definition of the derivative is a limit. Two errors pull against each other.

Truncation error. The slope between two points is not the slope at one point; the curvature of the function contributes an error of order `h²` for the two-sided formula. Smaller `h` reduces it.

Rounding error. `L(θ + h) − L(θ − h)` is a subtraction of two nearly equal numbers, and the previous lessons showed what that does: the difference is known only to about machine epsilon `ε` times the size of `L`. Dividing by `2h` amplifies that to `ε/h`. Smaller `h` *increases* it.

The total error is roughly `h² + ε/h`, smallest when the two terms balance, at `h ≈ ε^(1/3)`:

```
float64: ε = 2.2e−16  →  best h ≈ 6e−6,   best achievable error
≈ 4e−11
float32: ε = 1.2e−7   →  best h ≈ 5e−3,   best achievable error
≈ 2e−5
```

The float32 line is the important one. In single precision the best possible numerical derivative agrees with the truth to about five digits, and with `h = 1e−6`, which is what everyone tries first, the rounding term is `0.1`: the check is meaningless. Do gradient checks in float64. Convert the model, run the check, convert back.

What to compare, and the thresholds

A raw difference between the numerical and analytic gradients means nothing without a scale, so use the relative error:

```
rel = |analytic − numerical| / ( |analytic| + |numerical| + 1e−
12 )
```

In float64 with `h ≈ 1e−5`:

```
rel < 1e−7    correct
rel ~ 1e−5    probably correct; possibly a kink nearby, see be-
low
rel ~ 1e−3    something is slightly wrong: a missing factor, a
transposed matrix
rel > 1e−2    the gradient is wrong
```

Check a random sample of parameters rather than all of them; a model with a million weights would need two million forward passes, and twenty randomly chosen weights find a systematic error just as well.

The kinks

A ReLU has no derivative at zero; it has a slope of `0` on one side and `1` on the other. If a parameter nudged by `h` moves some unit across zero, the numerical derivative averages the two slopes and disagrees with the analytic one, which picked a side. This is not a bug. It shows up as a handful of parameters with relative error near `0.5` while all the others are at `10^-8`, and the tell is that the disagreeing ones change when you change `h` or the input. `max`, `abs`, sorting and any thresholding have the same kinks. If *every* parameter disagrees, the gradient is wrong; if a few do and the pattern moves, it is the kinks.

The code

```
import numpy as np
def grad_check(loss, theta, analytic, h=1e-5, n=20, rng=np.ran-
dom.default_rng(0)):
    theta = theta.astype(np.float64)
    for i in rng.choice(theta.size, n, replace=False):
        e = np.zeros_like(theta); e.flat[i] = h
        num = (loss(theta + e) - loss(theta - e)) / (2 * h)
        ana = analytic.flat[i]
        rel = abs(ana - num) / (abs(ana) + abs(num) + 1e-12)
        print(i, f"{ana:.6e} {num:.6e} {rel:.1e}")
```

Run it once against a gradient you are sure of, so you know what "passes" looks like on your machine, before running it against the one you doubt.

When the numerical derivative is the derivative

Sometimes there is no analytic gradient: a loss that calls a simulator, a black-box metric, a function with a lookup table in it. Finite differences then *are* the gradient, at a cost of two function evaluations per parameter, which is affordable for tens of parameters and hopeless for millions. That is the honest state-

ment of why backpropagation exists: it computes every parameter's gradient for the cost of about two forward passes in total, where finite differences cost two per parameter. Module 3 made that point from the algorithm's side. From the numerical side, the finite-difference gradient is also less accurate, by the `ε^(2/3)` above, so backpropagation is both a million times cheaper and a hundred thousand times more precise. Use finite differences to *check* it, in float64, at the `h` the arithmetic prescribes, and for nothing else.

BEFORE YOU MOVE ON

A learner checks a hand-written backward function in float32 with h = 1e-6 and finds relative errors around 0.1 on every parameter. What should they conclude?

A. The backward function is wrong, since 0.1 is far above any acceptable threshold.

B. The network's ReLU kinks are causing every parameter to disagree.

C. h should be made smaller, since the definition of a derivative takes h to zero.

D. The check is uninformative in float32 at that h; rerun it in float64.

76 · 9 min

NaN, Inf, and the five numbers to print about any tensor

THE ONE THING TO KEEP

NaN arises from 0/0, Inf – Inf, sqrt of a negative or log of a negative, propagates through everything, and is unequal even to itself, so it is found with isnan rather than ==, traced upstream rather than where it appears, and prevented by the stable log, variance and normalisation forms the earlier lessons gave.

Two special values, with rules

The floating-point standard reserves bit patterns for two things that are not numbers. **Inf**, positive or negative, is what overflow produces, and what division of a non-zero number by zero produces. **NaN**, "not a number", is what you get when the arithmetic has no defensible answer:

```
0 / 0          NaN
Inf − Inf      NaN
0 × Inf        NaN
Inf / Inf      NaN
sqrt(−1)       NaN
log(−1)        NaN
log(0)         −Inf   (not NaN, and the distinction is useful)
```

Both propagate. Any operation with a NaN input gives NaN, and most operations with Inf give Inf or NaN. One NaN in a gradient becomes a NaN in every weight it touches after one optimiser step, and then in every activation, and then in the loss. A NaN in training is therefore rarely where it appears; it is downstream of its origin, and the work is tracing it back.

The rule that surprises everyone

NaN is not equal to anything, including itself:

```
x = float("nan")
x == x            # False
```

This is by design, and it is the standard test: `x != x` is True exactly when `x` is NaN. It also means `x in some_list`, a `==` filter in pandas, and a dictionary lookup all fail to find a NaN. `np.isnan` and `torch.isnan` are the honest tests; `math.isnan` for scalars.

Sorting puts NaN at the end. `np.max` of an array containing one NaN returns NaN; `np.nanmax` skips it. Pandas `mean()` **skips NaN silently**, which is the more dangerous behaviour: a column with 60 per cent missing values reports a confident mean of the other 40 per cent, and nothing says so unless you print `count()` beside it. An accuracy computed over a column with NaN labels is an accuracy over the rows that had labels, which may not be the rows you meant.

Where NaN comes from in training

Each of the operations above has a training-time source. The ones that account for most incidents:

- **log of zero.** A probability underflowed to 0, then `-log(0) = Inf` in the loss, then NaN in the gradient. Module 5's log-sum-exp is the fix; so is never computing `log(softmax(x))` as two steps.
- **Square root of a small negative.** A variance computed by cancellation, as in the summation lesson, comes out at `-1e-9`; `sqrt` returns NaN; a normalisation layer spreads it everywhere. Clamp to zero, or compute the variance the stable way.
- **Division by a zero norm.** Normalising a vector that happens to be all zeros, a padding embedding, gives `0/0`. Add an epsilon to the denominator.

- **Overflow.** A learning rate too high pushes a weight to `1e20`, an activation to `Inf`, and `Inf - Inf` in the next layer norm gives NaN. Module 3's gradient-norm plot shows this coming several steps early; gradient clipping stops it.
- **fp16.** Everything above happens at `65,504` instead of `10^38`, which is the half-precision lesson.

Finding the origin

```
torch.autograd.set_detect_anomaly(True)     # raises at the
first backward op producing NaN
```

This slows training badly and names the operation, which is what you need once and then never again. Cheaper, and worth leaving on: assert after each step that the loss is finite, and log the step at which it stopped being so, because the first non-finite loss is usually a few steps after the first non-finite gradient, and the gradient-norm history around that step tells the story.

```
if not torch.isfinite(loss):
    raise RuntimeError(f"non-finite loss at step {step}")
```

The five numbers

For any tensor you are suspicious of, print five things:

```
def describe(t, name=""):
    t = t.detach().float()
    print(name, tuple(t.shape), t.dtype,
          f"mean={t.mean():.3g} std={t.std():.3g} maxabs=
{t.abs().max():.3g}",
          f"nonfinite={(~torch.isfinite(t)).sum().item()}")
```

Shape and dtype, because half of all bugs are a transposed matrix or an integer where a float was meant. Mean and standard deviation, because the variance lesson said what healthy looks like: activations with mean near 0 and

spread near 1, weights at their initialisation scale, gradients with a spread that is stable across steps. Maximum absolute value, because an activation of `10^4` is the overflow of ten steps from now. And the count of non-finite entries, because one is already too many.

Call it on the input batch, the first layer's output, the logits and the loss, in that order, at the step things went wrong. The first tensor whose numbers look wrong is upstream of the fault.

Inf on purpose

`-Inf` is also a tool. Masking in attention sets the scores of forbidden positions to `-Inf` before the softmax, and `exp(-Inf) = 0` exactly, which is cleaner than any large negative number. `float("inf")` is the right initial value for a running minimum. And a log-probability of `-Inf` for an impossible event is correct, not a bug, as long as nothing downstream subtracts two of them.

The rule

A NaN is never the problem; it is the problem's last symptom. Test with `isnan`, never with `==`. Print the five numbers, walk upstream, and find the log of zero, the root of a negative, the division by nothing, or the step whose norm doubled three times in a row.

BEFORE YOU MOVE ON

A training loss becomes NaN at step 4,210. The gradient norm was rising steeply from step 4,195 and reached 10^7 at step 4,208. Where should the search for the cause begin?

A. At step 4,210, in the loss function, since that is where the NaN first appears.

B. At step 4,195, where the norm began to run away: a high learning rate or a missing clip.

C. In the data loader, since a corrupt batch at step 4,210 is the usual cause of a sudden NaN.

D. In the random seed, since a different seed would likely avoid the batch that triggered it.

CASE STUDY · MODULE 8

One channel, ninety times the rest

An agri-tech company in Nagpur shipping a cotton-pest identification model to farmers' phones, where two thirds of users have devices with three gigabytes of memory.

The model identified fourteen cotton pests and disorders from a photograph of a leaf or a boll. It had 300 million parameters and worked well: 0.91 accuracy overall on a held-out set of nine thousand field photographs, and 0.81 on the four rare classes that mattered most, because those were the ones an extension officer would not recognise on sight.

In half precision the model is 600 megabytes. The company's user research said 62 per cent of their farmers had phones with three gigabytes of memory, on which the practical budget for an app of this kind is about 350 megabytes. At int8 the model is 300 megabytes and fits; at int4 it is 150 and fits comfortably.

The alternative was to keep the model on a server and send photographs to it. That is the easy engineering and it fails on the ground: the photograph has to be taken at the plant, and 44 per cent of the villages the company works in have no usable data connection in the field. A model that needs the network is a model that works in the office.

So the decision was between shipping a quantised model to the phone and shipping a better model that most users could not reach.

The first int4 attempt was a failure of a specific and instructive kind. Overall accuracy fell from 0.91 to 0.86, which the product manager could live with. Accuracy on the four rare classes fell from 0.81 to 0.42, which she could not, because those four are the reason the app exists.

The engineer looked at the tensors rather than at the accuracy, and found the cause in two channels. Quantisation stores each weight as a rounded multiple of a scale, and the scale is set by the largest value in the group so that the largest value lands on the top integer. Two channels in the later blocks carried activations about ninety times larger than the rest of the tensor. With one

scale for the whole tensor, those two channels set a step so coarse that almost every other weight in the tensor rounded to zero or to a single step. One number had flattened a million.

The fix was arithmetic rather than retraining: a separate scale for each block of 128 weights, so that the outlier sets the step only for its own block of 128. That costs sixteen extra bits per block, which is an eighth of a bit per weight, taking the model from 150 to about 155 megabytes. Rare-class accuracy came back to 0.78.

During the calibration pass the engineer also hit a NaN, and the trace is worth recording. A normalisation statistic was computed as the mean of the squares minus the square of the mean, on activations whose mean was around three thousand. Both terms were near nine million, both were known to about seven significant digits, and their difference was supposed to be a variance of about two — smaller than the error in either term. In one batch it came out negative, the square root returned NaN, and the NaN propagated through every later layer of the calibration. Subtracting the mean before squaring, so that the numbers being squared were near one rather than near three thousand, removed it.

That left the last decision, which was not technical. At int4 with block scales the model is 0.78 on the four rare classes against 0.81 for the full-precision model on a server. Three points, on the classes that matter most, in exchange for reaching every farmer rather than 56 per cent of them.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They shipped the int4 model with per-block scales. On the four rare classes the app does not give a confident answer below a threshold measured on field photographs; instead it saves the picture and queues it, and when the phone

next has signal the full-precision server model reviews the queue and sends back a corrected answer with a notification. About one identification in nine goes to that queue, and 71 per cent of queued photographs are answered within four hours. The three-point gap on rare classes is therefore paid only by farmers who never regain signal, rather than by the 44 per cent who would have had no app at all. The engineer's two additions to the release checklist are a per-block scale by default and a rule that no variance is ever computed as the mean of squares minus the square of the mean.

Why does one activation channel ninety times larger than the rest destroy the whole tensor at int8 or int4?

The scale is chosen so that the largest absolute value in the group lands on the top integer, so the step size is set by that largest value. If one value is ninety times the typical one, the step becomes about ninety times coarser than the rest of the tensor needs, and every ordinary weight rounds to zero or to one step. The error is half a step for everyone, which is fine for the outlier and catastrophic for the values a hundredth its size. Confining the scale to a block of 128 confines the damage to that block.

The variance came out negative and produced a NaN. What was the mechanism, and why did subtracting the mean first fix it?

The mean of the squares and the square of the mean were both about nine million and each was known to about seven significant digits, so each carried an error of roughly a unit. Their difference was supposed to be about two, which is smaller than the error in either term, so the leading digits cancelled and what remained was rounding error, which can come out negative. Subtracting the mean before squaring makes the quantities being squared small, so there is nothing large to cancel and the result is computed from digits that are actually known.

Why did the company measure rare-class accuracy separately rather than trusting the overall figure?

Because the overall figure is dominated by the common classes and hides the classes the product exists for. Quantisation error is not spread evenly: it is larger for small weights, larger in the layers nearest the output, and larger for classes with fewer redundant paths supporting them. Overall accuracy fell five points while rare-class accuracy fell thirty-nine, and a single number would have reported the

first and concealed the second. Anyone claiming int4 is free has not measured it on the part of their own task that matters.

MODULE 8 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A float32 running total of processed tokens stops changing at 16,777,216. What is happening?

A. The variable overflowed and wrapped round to its most negative value

B. The spacing between neighbouring float32 numbers there is 2, so adding 1 rounds back

C. The counter reached the largest number a float32 can hold

D. The addition returned NaN and the value froze there

2 A variance computed as the mean of the squares minus the square of the mean returns a negative number on data centred near 10,000. Why?

A. The data contain negative values, which the formula cannot handle

B. The formula is valid for populations only, not for samples

C. Both terms are near ten to the eighth and known to seven digits, so their difference is smaller than their error

D. The mean was computed on a different subset from the squares

3 Training keeps a float32 copy of every weight even though the forward pass runs in bf16. What breaks without it?

A. Small updates vanish, because the gap between bf16 numbers near one is 0.0078

B. The gradients overflow, since bf16 cannot represent large values

C. The model cannot be saved to disk in half precision

D. The optimiser cannot compute a moving average in half precision

4 You initialise every weight with a spread of 0.05 in layers of width 1,024 with ReLU between them. What happens over twenty layers?

A. Nothing: the spread is small, so activations stay near their input scale

B. The activations vanish, since 0.05 is far below the right scale

C. The variance is multiplied by about 1.28 a layer, so activations end up over a hundred times larger

D. The variance stays constant, because ReLU normalises its own output

5 A training run produces NaN in the loss at step 900. Where should you look first?

A. At the loss function, since that is where the NaN appeared

B. At the optimiser, which is the only thing that writes to the weights

C. At the random seed, since a different seed usually avoids it

D. Upstream and earlier: a NaN is the last symptom, and the gradient norms around that step tell the story

6 One feature ranges from 0 to 1 and another from 0 to a million, and a least-squares fit in float32 returns a meaningless coefficient on the small one. What is the fix?

A. Drop the small feature, since it clearly carries no signal

B. Standardise both columns, which brings the condition number down from about ten to the twelfth

C. Compute the matrix inverse explicitly rather than using a solver

D. Add more rows, which reduces the condition number in proportion to n

EXAMINATION

The Maths You Actually Need — final examination

This paper covers all eight modules: vectors and the space they live in, matrices and what a layer does, derivatives and gradient descent, probability and distributions, logarithms and where losses come from, estimation from samples, the arithmetic of cost and scale, and how numbers behave inside a machine. Every question tests a mechanism from a lesson rather than a definition, so the arithmetic you did by hand is the arithmetic that is examined. Twenty-five questions are drawn at random from a larger pool, and the pass mark is 70 per cent. There is no timer: read as slowly as you like, in whatever language you think in. If you do not pass, you may retake after thirty minutes and the paper will be drawn fresh, so a retake is a different set of questions rather than the same one again. A teacher can open the next course for any learner at any time regardless of this paper, and that is recorded as a teacher's decision rather than as a pass.

On the website a sitting is 25 questions drawn from this pool and the pass mark is 70%. There is no time limit, there and here. Passing opens the next course in the track.

1 1. Vectors, and the space they live in

Every vector in an index has been normalised to length one. A team benchmarks ranking by cosine against ranking by Euclidean distance. What will they find?

A. Cosine ranks better, because Euclidean distance ignores direction

B. Euclidean ranks better on long documents, where cosine saturates

C. The two orders agree only in fewer than about a hundred dimensions

D. The two produce the identical ordering, since squared distance is two minus twice the cosine

2 1. Vectors, and the space they live in

What is an embedding lookup, in terms of the matrix arithmetic it replaces?

A. A one-hot vector multiplied by the embedding matrix, optimised into an array index

B. A dot product between the token's vector and every row of the table

C. A projection of the token onto the first 768 principal components of the corpus

D. A hash of the token's characters, rescaled to the width of the model

3 1. Vectors, and the space they live in

On one embedding model, unrelated sentence pairs average a cosine of 0.6 and matching pairs average 0.78. What follows for the threshold you should use?

A. The model is broken, since unrelated pairs should sit near zero

B. Use 0.5, the midpoint of the range a cosine can take

C. The signal is the variation around 0.6, so the cut-off must be read off labelled pairs from this model

D. Subtract 0.6 from every score and then apply the usual 0.8 threshold

4 1. Vectors, and the space they live in

Why must the mean and spread used to standardise features be computed on the training rows alone?

A. Because test data usually has a different distribution, which would bias the scaling

B. Because computing them over everything lets the test rows inform the model, which is a leak

C. Because the test set is smaller and would dominate the averages

D. Because standardisation is defined only for data the model has already seen

5 1. Vectors, and the space they live in

If distances concentrate in high dimensions, why does embedding search return sensible results every day?

A. Because search libraries correct for concentration with a scaling factor
B. Because concentration only sets in above about ten thousand dimensions
C. Because trained embeddings occupy a low-dimensional structure inside the big space rather than filling it
D. Because cosine is immune to concentration while Euclidean distance is not

6 1. Vectors, and the space they live in

Why is BM25 hard to beat on queries containing a part number or an error code?

A. It scores rare exact terms highly and matches the string rather than representing it
B. It was tuned on technical documentation, unlike general embedding models
C. It converts every query into a dense vector before matching
D. It assigns common words a high inverse document frequency

7 1. Vectors, and the space they live in

Averaging the sentence embeddings of a paragraph gives a usable paragraph vector. What does that summary discard?

A. The length of the paragraph, which the average preserves only as a scale
B. Word order entirely, so the dog bit the man and the man bit the dog land in the same place
C. Nothing: averaging is a lossless summary of a set of vectors
D. The direction, keeping only the magnitude

8 2. Matrices, and what a layer really does

A tensor of shape (32, 128, 768) is multiplied by a weight matrix of shape (768, 3072). What comes out, and why?

A. (32, 128, 3072): the product acts on the last two dimensions and treats the rest as batch

B. (32, 768, 3072), because the middle dimension is the one contracted

C. (128, 768), because batching is applied after the multiplication

D. An error, because a three-dimensional tensor cannot multiply a matrix

9 2. Matrices, and what a layer really does

Splitting a tensor of shape (batch, seq, 768) into 12 heads of width 64 needs a reshape and then a transpose. What happens if you reshape straight to (batch, 12, seq, 64)?

A. It raises an error, because the element count does not match

B. It produces identical numbers, since reshape and transpose commute

C. It halves the memory, which is why some implementations prefer it

D. It is legal arithmetic that slices the sequence across heads instead of the features, and nothing errors

10 2. Matrices, and what a layer really does

To solve a linear system you should call a solver rather than forming the matrix inverse. Why?

A. Because the inverse does not exist for most square matrices

B. Because the solver uses the determinant, which is more stable

C. Because forming the inverse costs about three times the work and adds rounding errors of its own

D. Because the inverse is defined only for symmetric matrices

11 2. Matrices, and what a layer really does

Two features are 0.98 correlated and a linear fit gives one a large positive coefficient and the other a large negative one. What is happening, and are the predictions wrong?

A. The model is broken and the predictions are unusable

B. The matrix is nearly singular, so the coefficients are unstable while the predictions can be fine

C. The features have opposite effects, which the signs correctly report

D. One feature must be dropped before any prediction is valid

12 2. Matrices, and what a layer really does

A recurrent weight matrix has a largest eigenvalue of 1.4. What happens to a signal passed through it over fifty time steps?

A. It decays towards zero, since eigenvalues shrink at each application

B. It stays the same size, because eigenvectors are not rotated

C. It rotates without changing length, since 1.4 is a real number

D. It grows by about 1.4 to the fiftieth, roughly five hundred million

13 2. Matrices, and what a layer really does

Truncated singular value decomposition is the best rank-k approximation of a matrix in squared error. What does that let you decide before compressing anything?

A. That any compression will preserve your task metric to within the same error

B. That the error is the sum of the squares of the discarded singular values, and no cleverer low-rank method exists

C. That the matrix can always be replaced by two thin ones without measurable loss

D. That the condition number will improve after truncation

14 2. Matrices, and what a layer really does

Adding a column of random numbers to a regression raises its R-squared. Why, and what should be reported instead?

A. Because R-squared never falls when a feature is added; report it on data the model did not fit

B. Because random noise genuinely helps the fit; report the noise level alongside it

C. Because R-squared is a correlation, and correlations rise with more variables by definition

D. Because the residuals are recomputed on a larger matrix; report the determinant instead

15 3. Change, slopes and the walk downhill

Checking a hand-written gradient, you shrink the step h from 1e-4 to 1e-12 and the answer gets worse. Why?

A. The function became non-differentiable at that scale

B. Truncation error grows as h shrinks, and it is the dominant term

C. The two values agree in their leading digits, so subtracting leaves rounding error over a tiny number

D. The framework cached the earlier result and returned it again

16 3. Change, slopes and the walk downhill

A blog post gives 3e-4 for training a transformer, and it makes a fine-tune diverge within forty steps. What is the reason?

A. Fine-tuning uses a different optimiser, which needs a different scale

B. That figure is for training from scratch; adjusting a good solution needs ten to a hundred times less

C. Fine-tuning has fewer parameters, so each one gets a larger share of the step

D. The figure was for SGD, which always needs a larger rate than Adam does

17 3. Change, slopes and the walk downhill

Why does almost every large training run raise the learning rate from near zero over the first few hundred steps?

A. To let the data loader fill its buffers before real steps begin
B. Because the loss surface is flat at initialisation and small steps explore it better
C. Because the schedule requires a symmetric shape at both ends
D. Because Adam's moment estimates are built from very few samples at the start and are badly noisy

18 3. Change, slopes and the walk downhill

Adam stores two extra numbers per parameter. For a seven-billion-parameter model in 32-bit, what does that add?

A. About 56 GB, on top of 28 GB of weights and 28 GB of gradients
B. About 14 GB, since the moments are kept in half precision
C. Nothing measurable, since the two moments are scalars
D. About 7 GB, one byte per parameter for each moment

19 3. Change, slopes and the walk downhill

Two runs of the same network on the same data with different seeds reach the same accuracy but behave differently on edge cases. Why is that expected?

A. Because the data loader shuffles, which changes the effective dataset
B. Because the loss is not convex, so different starts reach genuinely different solutions
C. Because floating-point arithmetic is random
D. Because accuracy is the only thing training optimises, so everything else is free to drift

20 3. Change, slopes and the walk downhill

The loss falls and then plateaus while the gradient norm stays high. Which fix addresses the mechanism?

A. Raise the learning rate, since progress has stalled
B. Add more data, since the model has run out of signal
C. Decay the learning rate, since the optimiser is oscillating across a narrow valley
D. Switch to a larger model, since capacity is now the limit

21 3. Change, slopes and the walk downhill

Gradient clipping rescales the whole gradient vector when its norm exceeds a threshold. What does it preserve?

A. The direction, while capping the magnitude
B. The magnitude, while rotating the direction towards the steepest axis
C. The sign of each parameter's gradient, zeroing everything else
D. Nothing: it replaces the gradient with a fixed unit step

22 4. Probability you can rely on

A model with a 50,000-token vocabulary generating a twenty-token sentence has about ten to the ninety-fourth possible outputs. Which conclusion follows?

A. The model has memorised a large fraction of that space
B. Beam search with eight candidates covers a meaningful share of it
C. Exhaustive scoring of continuations is impossible, so decoding must be greedy or sampled
D. The vocabulary should be reduced until the space becomes enumerable

23 4. Probability you can rely on

A doctor says thirty per cent chance of dengue and it turns out to be dengue. What does that single case tell you about her numbers?

A. That she was wrong, since she gave the outcome less than half
B. Nothing on its own: you check whether about three in ten of her thirty per cent cases turn out that way
C. That she is underconfident and should raise her estimates
D. That the true probability was one hundred per cent all along

24 4. Probability you can rely on

The probability a language model assigns to a sentence is a product of one factor per token. Why is it computed as a sum of logs?

A. Because log probabilities are easier for a person to read
B. Because taking the log makes rare tokens count for less
C. Because the chain rule of probability only holds in log space
D. Because multiplying five hundred numbers below one underflows to exactly zero in any float format

25 4. Probability you can rely on

A Monte Carlo estimate from 200,000 trials gives a probability to three reliable digits. What does a fourth digit cost?

A. A hundred times the trials, because the error falls with the square root of the count
B. Twice the trials, since precision doubles with the sample size
C. Ten times the trials, in proportion to the digit gained
D. Nothing: the fourth digit is already correct and simply not printed

26 4. Probability you can rely on

Why do so many methods assume a diagonal covariance rather than a full one?

A. Because off-diagonal terms are almost always zero in practice
B. Because a full 768-dimensional covariance has 295,296 free parameters to estimate
C. Because a diagonal matrix is the only kind that can be inverted
D. Because correlations between features are not meaningful for continuous data

27 4. Probability you can rely on

A vendor sells forty bowls on sixty per cent of days and fifteen on the rest, an expectation of thirty. What does thirty describe?

A. The most likely single day's sales, since sixty per cent is the larger share
B. The number of bowls he can expect to sell on a typical trading day
C. A weighted average that never actually occurs on any day
D. The midpoint between his two possible daily outcomes

28 4. Probability you can rely on

Of a hundred tickets, sixty are billing and forty technical; forty-five billing and twenty technical are resolved on the first reply. What is the probability that a resolved ticket was billing?

A. 0.75, which is 45 out of 60

B. 0.45, which is 45 out of 100

C. 0.60, the share of all tickets that are billing

D. 0.69, which is 45 out of 65

29 5. Logarithms, information and the shape of a loss

A language model reports a loss of 3.0 nats per token. What does exponentiating it give you?

A. The number of parameters needed to reach that loss

B. About 20, the effective number of equally likely options at each token

C. The compression ratio against the raw text

D. About 8, since three doublings reach eight

30 5. Logarithms, information and the shape of a loss

A model scores 2.0 nats per token on a corpus of a billion tokens. What does that number equal in the compression reading?

A. The size of the model, expressed in bits

B. The number of tokens the model has memorised

C. The number of yes-or-no questions needed to specify the model's weights

D. About 361 MB, the size the model would compress that text to

31 5. Logarithms, information and the shape of a loss

Four outcomes. Which distribution has the largest entropy, and what is it?

A. (0.97, 0.01, 0.01, 0.01), at 2 bits, since rare events are surprising

B. (0.5, 0.25, 0.125, 0.125), at 1.75 bits, since it is the most spread out

C. (0.25, 0.25, 0.25, 0.25), at 2 bits, since equal likelihood is maximally uncertain

D. (1, 0, 0, 0), at 4 bits, since certainty carries the most information

32 5. Logarithms, information and the shape of a loss

Dividing every logit by a temperature of 0.5 before softmax does what to the output?

A. Sharpens it, because the gaps between logits are doubled
B. Flattens it, since dividing makes the numbers smaller
C. Leaves it unchanged, since softmax is invariant to scaling
D. Shifts every probability down by 0.5 and renormalises

33 5. Logarithms, information and the shape of a loss

Why does training against exact one-hot targets push a model towards ever larger logit gaps, and what stops it?

A. Nothing stops it, which is why trained models eventually produce infinite logits
B. The loss reaches zero only at a probability of one, and label smoothing gives a finite place to stop
C. Weight decay stops it by shrinking every logit towards zero
D. Nothing pushes it: cross-entropy is minimised at a probability of 0.5

34 5. Logarithms, information and the shape of a loss

A twelve-class classifier trains, but its loss stalls just under 2.48 nats. What is the likeliest cause?

A. The learning rate is too small, so it has not yet converged
B. The classes are balanced, and 2.48 is the best achievable loss
C. A softmax was applied before a loss function that expected raw logits
D. The batch size is too large, which flattens the gradient

35 5. Logarithms, information and the shape of a loss

Beam search holds a log probability of minus 700 and another of minus 701, and needs the log of their sum. What does it compute?

A. Minus 700 plus the log of one plus e to the minus one, which is minus 699.69
B. The sum of the two logs, minus 1401
C. The larger of the two, minus 700, since the smaller is negligible
D. The log of e to the minus 700 plus e to the minus 701, computed directly

36 6. Estimation: from a sample to a number you can trust

A model scores 85 per cent on 200 items. What is the rough 95 per cent range?

A. 84.9 to 85.1, since the score is precise to one decimal place
B. 80 to 90, because the standard error is about 2.5 points
C. 75 to 95, which is two standard deviations of the data
D. 83 to 87, because the standard error is about one point

37 6. Estimation: from a sample to a number you can trust

Two models score 85 and 88 on the same 200 items and disagree on 30 of them, the second winning 18 to 12. Why is that comparison sharper than two separate sets?

A. Because the same items were used, so the sample size doubled to 400
B. Because a paired test always halves the standard error
C. Because the scores are now correlated, which removes the uncertainty
D. Because the 170 items both got right or both got wrong add nothing to the noise of the difference

38 6. Estimation: from a sample to a number you can trust

You want a 95 per cent chance of observing at least one failure of a fault that occurs once in a thousand requests. How many trials do you need?

A. About three thousand
B. About a thousand, giving one expected failure
C. About three hundred, since three over n is the bound
D. About a million, the square of the rate

39 6. Estimation: from a sample to a number you can trust

One item has one click from one view; another has forty clicks from a hundred views. With a Beta(2, 8) prior, which ranks higher?

A. The first, at 100 per cent, since its observed rate is higher
B. They tie, since the prior adds the same amount to both
C. The second: smoothing gives 3 over 11 against 42 over 110, so 0.27 against 0.38
D. The first, because a prior cannot change an ordering

40 6. Estimation: from a sample to a number you can trust

What can you say about a 95 per cent Bayesian credible interval that you cannot say about a 95 per cent confidence interval?

A. That it is narrower, because a prior adds information
B. That the true value lies inside it with 95 per cent probability, given the prior and the data
C. That it depends on no modelling assumption at all
D. That it will contain the estimate, which a confidence interval need not

41 6. Estimation: from a sample to a number you can trust

Why does stopping training early act like an L2 penalty on the weights?

A. Because it reduces the number of parameters that were ever updated
B. Because unfinished training leaves noise in the weights, which regularises
C. Because low-curvature directions have barely moved from zero, which is where an L2 penalty holds them
D. Because the validation loss is what chooses the stopping point

42 6. Estimation: from a sample to a number you can trust

Dividing the sample variance by n is biased low, yet it can be the better estimator. How?

A. It cannot be: an unbiased estimator is always preferable
B. Because bias disappears entirely once the sample exceeds thirty
C. Because dividing by n is faster to compute on large samples
D. Because bias and variance both contribute to error, and it has the lower variance

43 7. Counting the cost: complexity and the arithmetic of scale

Which growth rate stops working at around a million items however fast the hardware is?

A. n log n, because the logarithm grows without bound
B. log n, because thirty steps is already too many
C. n, because a single pass over a million items is expensive
D. n squared, because a million squared is ten to the twelfth operations

44 7. Counting the cost: complexity and the arithmetic of scale

Why does a forward pass cost about two floating-point operations per parameter per token?

A. Because each parameter is read twice, once for input and once for output
B. Because each weight is used for one multiplication and one addition
C. Because half precision splits each operation into two
D. Because the attention scores double the cost of the linear layers

45 7. Counting the cost: complexity and the arithmetic of scale

Why does a hosted model charge more for a long context even when the reply is short?

A. Because long prompts are harder for the model to understand
B. Because the tokeniser slows down on long inputs
C. Because the key and value vectors of every token in every layer are held in memory
D. Because the number of parameters grows with the context length

46 7. Counting the cost: complexity and the arithmetic of scale

A Bloom filter at ten bits per item with seven hash functions has a false-positive rate under one per cent. What can it never do?

A. List what it contains
B. Say that an item is absent when it is in fact present
C. Answer a membership query in constant time
D. Hold more than about a million items

47 7. Counting the cost: complexity and the arithmetic of scale

Ordinary hashing scatters similar inputs far apart. Why does deduplication want the opposite?

A. Because scattering makes the hash table slower to search
B. Because near-duplicates must land in the same bucket if all-pairs comparison is to be avoided
C. Because similar documents have similar lengths, which the hash must preserve
D. Because a scattered hash cannot be inverted to recover the document

48 7. Counting the cost: complexity and the arithmetic of scale

Repeatedly multiplying a probability vector by a graph's transition matrix converges to a fixed vector. What is that vector?

A. The row of the matrix with the largest sum
B. The average of every node's degree, normalised
C. The eigenvector with eigenvalue one: the share of time a long random walk spends at each node
D. The determinant of the transition matrix, spread across the nodes

49 7. Counting the cost: complexity and the arithmetic of scale

You have 80,000 embeddings of 768 dimensions and a single user querying them. What should you build?

A. Nothing: brute force is about a millisecond and returns the true nearest every time
B. An HNSW graph, since approximate search is always faster
C. A product-quantised index, since 80,000 vectors will not fit in memory
D. A clustered index with 280 centres, to keep the query cost constant

50 8. Numbers inside a machine

Why does 0.1 plus 0.2 equal 0.3 in float32 but not in float64?

A. float32 stores decimals exactly while float64 approximates them
B. float64 uses a different rounding mode
C. Neither format stores a tenth exactly, and in float32 both sides happen to round to the same value
D. float32 rounds to two decimal places before any comparison

51 8. Numbers inside a machine

Adding ten million values of 0.1 in a float32 Python loop gives 1,087,937, while NumPy's sum gives 1,000,000. Why?

A. NumPy adds neighbours first, so each addition combines numbers of similar size
B. NumPy uses float64 internally for every operation
C. NumPy rounds its answer to the nearest round number
D. The Python loop adds a different value, since 0.1 is stored differently there

52 8. Numbers inside a machine

Two runs with the same seed on the same GPU produce slightly different models. What causes it?

A. The seed does not fix the dropout mask on a GPU
B. The GPU generates its own random numbers and ignores the seed
C. The framework jitters the learning rate to escape saddle points
D. Floating-point addition is not associative, and the threads' reduction order varies

53 8. Numbers inside a machine

A dataset index built as int32 produces negative positions past 2.1 billion tokens, and nothing crashes. Why not?

A. Negative indices raise a warning that the script suppressed
B. The value wraps to the most negative integer, and negative indices legally count from the end
C. Python integers grow without limit, so the values are still correct
D. The tokens are stored as int64, which masks the error

54 8. Numbers inside a machine

After a ReLU an activation is never negative. Why does asymmetric quantisation pick a zero-point that maps a real zero onto an exact integer?

A. Because integers cannot represent a zero otherwise
B. Because padding and ReLU produce exact zeros in quantity, and rounding them adds a bias everywhere
C. Because the scale becomes undefined if zero is not representable
D. Because negative integers are unavailable in unsigned formats

55 8. Numbers inside a machine

A gradient check in float64 shows twenty parameters at a relative error near ten to the minus eight and three near 0.5. What does that pattern mean?

A. The gradient is wrong, since any disagreement at all is a failure

B. The step size is too large, but only for those three parameters

C. Those three nudge a unit across a ReLU's kink, where the numerical derivative averages two slopes

D. Those three parameters are unused and their gradients are undefined

56 8. Numbers inside a machine

You shuffle with a fixed seed and take the first 80 per cent as training. New rows arrive. What breaks?

A. Nothing, because the seed makes the shuffle reproducible

B. The shuffle fails, because the seed was chosen for the old row count

C. The split ratio drifts away from 80 per cent as rows are added

D. Every row's position shifts, so yesterday's test examples move into training

Answers

Each entry gives the correct option, then what each wrong one reveals about how somebody was thinking. The second part is worth more than the first: a wrong answer here is a named misreading, not a mistake.

Lesson checks

1. Everything a model sees is numbers — B

A — Identifies a real problem with unscaled columns, but scaling a meaningless ordering just produces a meaningless ordering between 0 and 1.

C — Universal approximation says a network can fit a function of the input it is given; it does not conjure back the adjacency information the encoding destroyed.

D — Describes leakage, which is a different failure; here the encoding is wrong even with a clean split.

2. A vector is just a list — B

A — Treats similarity as growing with size, when the ranking only ever looked at angle.

C — Assumes a vector's position is what matters, when only its direction from the origin does.

D — Reads length as intensity, so a longer vector must mean the thing more strongly.

3. What a dot product measures — C

A — Assumes the score reflects real relevance rather than an artefact of vector length.

B — Thinks a dot product is term overlap, so more words mechanically means a higher score.

D — Blames the model when the scoring function, not the model, is doing the damage.

4. How long, and how far — D

A — Confuses normalisation with dimensionality; the arithmetic still runs, which is precisely why the bug survives.

B — This is actually right for a single query, and the trap is that it feels wrong; the real damage appears when scores are compared across queries or against a fixed threshold.

C — Multiplying by a positive constant cannot reverse an order; nothing here flips signs.

5. Cosine or distance, and why it often does not matter — B

A — States a real property of cosine, but it cannot produce a difference here: with unit vectors, magnitude has already been removed from both measures.

C — Squared distance is a monotone function of distance, so omitting the square root leaves every ranking exactly as it was.

D — Recall@10 is computed from a ranking; both measures produce a ranking, so they are perfectly comparable.

6. Projection, and the art of removing a direction — C

A — The projection can be arithmetically perfect and this still happens; zero dot product with one vector is not zero information about the concept.

B — Possible in principle, but the standard finding here is measured on held-out data, and the mechanism is about the geometry rather than the fitting.

D — Overstates the case: the projection does change the vectors and does remove some signal, it just does not remove all of it.

7. The same vector, written two different ways — B

A — PCA to fewer dimensions is lossy too; the difference is which information it chooses to lose, not whether it loses any.

C — Both operations change vector lengths, and both are usually followed by re-normalisation; this is not the deciding issue.

D — Directionally true but it misses the mechanism: what makes the fixed operation bad here is specifically that the basis is arbitrary.

8. Why your 3-D intuition is wrong in 768 dimensions — C

A — HNSW is used at this width in production constantly; the dimension is not what broke the benchmark.

B — Configuration matters, but no setting rescues an index whose whole premise — exploitable local structure — is absent from the data.

D — More uniformly random points make concentration worse, not better; the problem is the distribution, not the count.

9. Sparse and dense, and the memory each costs — C

A — Truncation happens with long documents, but a single short code is well within any model's limit; the failure is about representation, not length.

B — Normalisation is real but is not the cause; even un-normalised, the model has no distinctive representation for a token it barely saw.

D — Cosine returns 1.0 for identical vectors and handles exact matches fine; the problem is upstream, in what the model encoded.

10. Doing a matrix multiplication yourself, once — A

B — Picks the wrong axis: the rule is that the last axis of the input must match the first axis of the weight, not the second-to-last.

C — Only true for a square matrix; this one is 768 in and 3072 out, so the width changes.

D — Nothing here transposes the batch and sequence axes; they pass through untouched.

11. A matrix is a stack of questions — A

B — Explains it as an optimisation failure, when it is an algebraic identity — the second layer could not help even if trained perfectly.

C — Assumes more parameters always means more expressive power, so the disappointment must be statistical.

D — Pictures the redundancy as duplicated weights rather than as composition collapsing into one map.

12. Shapes, batches, and the broadcasting rules — B

A — Scale alone would not damage accuracy this badly, and it is not what these shapes cause.

C — `.mean()` with no axis averages everything, so no row is excluded; the problem is that too many entries exist, not too few.

D — They are compatible under the broadcasting rules, which is exactly why no error appears and the result is not zero.

13. Undoing a matrix, and why you rarely should — D

A — Scaling changes coefficient magnitudes but does not produce the huge equal-and-opposite pair, and ridge does not rescale features.

B — Ridge shrinks coefficients rather than removing them, and the equal-and-opposite signature points at conditioning, not capacity.

C — A real situation with its own remedy, but the described symptom is the correlated-pair one and can happen with plenty of rows.

14. The determinant, and what it says about collapse — A

B — Rotations do have determinant 1, but so do many non-rotations; the condition is necessary, not sufficient.

C — This is the common and costly mistake: a matrix with singular values 100 and 0.01 has determinant 1 and is badly conditioned.

D — That would require an orthogonal matrix; determinant 1 says total volume is preserved, not individual lengths.

15. The best fit, and what "best" was defined to mean — B

A — True only when the model includes an intercept column; with no intercept the residuals need not sum to zero at all.

C — Normality is an assumption used for confidence intervals and tests, not something the fitting procedure produces or guarantees.

D — Least squares minimises the sum of squares, not every individual residual; another fit can beat it on any particular point.

16. Rank, and why a huge matrix can be two small ones — B

A — The base model's rank does not limit the update's rank; LoRA adds to the weights rather than living inside them.

C — The adapter changes the effective weights and so changes the function; freezing the base does not cap what the sum can express.

D — Conditioning of the factors is not the mechanism here, and LoRA at rank 4 trains stably in practice.

17. The directions a matrix does not turn — A

B — Reverses the direction: values above 1 amplify, values below 1 damp.

C — Repeated application is precisely what eigenvalues describe well, since each step multiplies the component by the same factor.

D — The direction does converge to the top eigenvector, but nothing normalises the state inside the recurrence, so the magnitude is not bounded.

18. Singular values, and the honest measure of importance — C

A — Slow decay says the opposite of compressible, and conditioning is about the ratio of largest to smallest, not the decay profile.

B — Centring matters for PCA on data; a weight matrix is not data and is not centred before its SVD.

D — Randomised SVD is an approximation of the same decomposition and is faster, never more accurate.

19. Slope, before anyone says the word derivative — A

B — Doubles go down to roughly 1e-308; 1e-15 is perfectly representable on its own, and the problem is the addition, not the number.

C — Assumes infinite precision; a correct, varying function still shows no change when the step falls below the spacing of representable values near x.

D — Finite differences approximate a local slope and do not care about convexity at all.

20. Slopes, and why training is walking downhill — B

A — Reads the steepness of a slope as a distance to the bottom, when a slope only describes the spot you are standing on.

C — Turns a local sensitivity into a claim about the whole model, when the same weight may be flat almost everywhere else.

D — Confuses the gradient, which is a property of the loss surface, with the step size, which you chose yourself.

21. Many knobs, one derivative each — B

A — At a minimum the gradient norm would be small, not large; these two observations contradict each other.

C — A large gradient guarantees a large change only for an infinitesimal step; at a finite learning rate, overshoot can cancel the progress.

D — A too-small learning rate gives slow but steady descent, whereas this pattern is flat loss with persistent large gradients.

22. The chain rule, which is the whole trick — C

A — The gradient still traverses every block; what changes is the factor contributed, not the count.

B — That describes what a normalisation layer does; a residual connection adds an identity path and does not rescale anything.

D — Residual networks are not convex; they are easier to optimise empirically, which is a different and weaker claim.

23. Backpropagation, and where the memory goes — C

A — Adam's moments are per parameter, not per example, so they do not change with batch size at all.

B — Per-example gradients are averaged into a single gradient of the same size as the weights; the gradient buffer does not scale with the batch.

D — Weights are shared across the batch and stored once; nothing duplicates them.

24. Convexity, and why nobody promises anything about deep networks — C

A — ReLU networks are piecewise linear in their inputs but not in their parameters, and the loss surface retains plenty of structure.

B — Bounded below says nothing about how many distinct flat points exist between the start and the bottom.

D — Adding parameters does not create convexity; the permutation symmetries alone guarantee it cannot be convex.

25. Curvature, and what the optimisers are actually doing — B

A — That is closer to what the first moment does; the second moment is a scale, not a smoother.

C — No adaptive method guarantees monotone decrease; Adam frequently increases the loss on individual steps.

D — The rescaling is relative to each parameter's own history, so a uniformly vanishing gradient stays vanishing.

26. Reading a training failure from the numbers — C

A — Larger batches give less noisy gradients, not more, so this reasoning runs backwards.

B — Dead ReLUs suppress gradients towards zero; they cannot produce a norm of 400.

D — A larger step in a region that is already overshooting makes divergence certain rather than curing it.

27. The one hyperparameter worth your afternoon — B

A — Seeing more data per step does not increase overfitting; if anything the gradient estimate is better.

C — Averaging gradients is the intended behaviour, and cancellation of conflicting gradients is not a capacity effect.

D — Batch-norm statistics get more reliable with larger batches, not less, and the described model need not use batch norm at all.

28. Counting, and the size of the spaces involved — C

A — Counts values rather than pairs; the first collision is expected far earlier than the space is full.

B — The tiny-fraction reasoning is exactly what the birthday argument overturns, and 5 billion is already past the threshold.

D — Even a perfectly uniform hash collides at this rate; uniformity is what the birthday calculation assumes.

29. Probability is a degree of belief — C

A — Treats a high probability as a promise about a single event, so anything short of certainty is a broken promise.

B — Uses a true general fact to justify a conclusion that one outcome cannot support.

D — Assumes a probability needs a repeatable event, so one-off forecasts are beyond judgement.

30. Probability once you know something — A

B — Accuracy is a different quantity again, and knowing it would still not give the probability that a flag is correct.

C — They come from the same matrix but divide by different totals: recall divides by actual positives, the flag-correctness rate by predicted positives.

D — The threshold does affect recall, but supplying it alone still would not convert recall into the probability a flag is correct.

31. Bayes, done slowly, with real numbers — B

A — Reads the test's error rates as the answer, which is exactly the swap between P(positive|disease) and P(disease|positive).

C — Equal error rates say nothing about the balance of outcomes when the two groups differ in size by a factor of ten thousand.

D — The proportion is independent of the number screened; only the base rate and the two error rates are needed.

32. Correlation, dependence, and the gap between them — C

A — Redundancy affects a model's use of a feature, but Pearson is computed pairwise with the target and is not suppressed by another feature's presence.

B — Pearson is invariant to linear rescaling of either variable, so units cannot be the cause.

D — Missing values reduce the sample used, which widens uncertainty rather than systematically pushing the estimate to zero.

33. When you cannot solve it, simulate it — C

A — Assumes linear improvement; the standard error falls with the square root, so ten times the trials gives only about three times the precision.

B — Doubling the trials divides the error by the square root of two, roughly 1.41, not by 2.

D — There is no such floor at these magnitudes; the error keeps falling with the square root of the count.

34. Distributions, and why the normal one keeps showing up — B

A — Treats a wrong shape as a sampling problem, which more data will not fix.

C — Argues about the one-sided versus two-sided convention instead of the assumption, and still leaves the 8% unexplained.

D — Keeps the normal assumption alive by blaming drift, which would push the alert rate both up and down rather than only up.

35. Six distributions, and the process each one describes — B

A — Poisson requires mean and variance to be equal; 12 against 140 is a factor of twelve apart.

C — Gets the direction wrong: variance above the mean is overdispersion, and it makes Poisson intervals too narrow rather than too wide.

D — Five hundred observations estimate a variance perfectly adequately; a twelvefold gap is far beyond sampling noise.

36. Expectation and variance in plain terms — C

A — Confuses the value an estimate is centred on, which does not change, with how much it wobbles, which does.

B — Assumes the error shrinks linearly with sample size rather than with its square root.

D — Mixes up how big your steps are with how noisy the direction of those steps is.

37. Covariance, and the shape of a cloud of points — B

A — Estimation error grows with dimension, and 400 samples for a 768x768 matrix is far below what is needed.

C — Whitening is defined for any data with a covariance; non-normality makes it less interpretable, not invalid.

D — A reasonable preprocessing choice, but it does not address the fundamental shortage of samples relative to dimensions.

38. Heavy tails, and the average that describes nobody — B

A — A window change can help, but averaging within any window still hides the tail that users experience.

C — Plausible as a measurement gap, but it does not explain why a correct average would still misrepresent the experience.

D — Dismisses the report rather than examining the statistic, and the tail explanation accounts for the complaint directly.

39. What a log is, and why the axis got logged — B

A — Reads a log axis as if it were linear; equal spacing on a log axis is equal ratio, and zero is not on the axis at all.

C — A power law is straight on log-log axes; with only the y-axis logged, a straight line is an exponential.

D — A log axis expands the small values rather than compressing them; a straight line on it is a real and informative shape.

40. Exponentials, decay, and the memory of a moving average — A

B — Treats an exponential average as a moving window; the weights decay by a factor of β per step and never reach exactly zero.

C — A β near 1 lengthens the memory but does not make it uniform; a gradient 5,000 steps back carries weight 0.999^5000, under one per cent.

D — Confuses β2 with β1 = 0.9; the window is 1/(1−β), and 1/(1−0.999) is a thousand, not ten.

41. Logs, and why losses are logged — C

A — Believes log loss pays you to be vague, when it penalises underconfidence as well as overconfidence.

B — Reads an ordinary difference in probability quality as evidence of leakage.

D — Assumes accuracy fully determines quality, so any remaining gap has to be a units artefact.

42. Surprise, bits, and why prediction is compression — C

A — Confuses the count of alternatives with the log of the probability; eight equally likely alternatives would cost log2(8) = 3 bits, not 8.

B — Uses the probability directly instead of its negative log; a cost that fell as the probability fell would reward bad predictions.

D — Surprise depends only on the probability the model assigned to the outcome that occurred; the vocabulary size affects the uniform baseline, not this cost.

43. Entropy: the average surprise, computed by hand — B

A — Uses the balanced-label baseline for an unbalanced label; the relevant floor is the entropy of a 10/90 split, 0.325 nats.

C — Reads the loss as an absolute quality score; a cross-entropy only means something against the entropy of the labels themselves.

D — The entropy of the labels is enough to judge this: a loss above it means the model is doing worse than a constant, whatever the accuracy.

44. Cross-entropy and KL: the cost of believing the wrong distribution — D

A — Ignores the floor: no model can score below the entropy of the labels, so there is at most 0.05 nats left to gain.

B — A bouncing optimiser shows as a noisy or rising curve, not a flat one sitting a few hundredths above the theoretical floor.

C — Both are given in nats here; cross-entropy equals entropy plus KL, so they are directly comparable and the difference is the KL.

45. Loss functions are not arbitrary: maximum likelihood — C

A — Confuses the distribution of the targets with the distribution of the errors; squared error assumes nothing about how prices are spread, only about how predictions miss.

B — Squared error carries no assumption about the model's shape; it would pull just as hard on a deep network's fit.

D — Independence may also fail, but it is not what makes a few large errors dominate; the Gaussian's thin tails are what treat a 10× error as nearly impossible.

46. Softmax, its gradient, and the shift it cannot see — A

B — Reads a logit as an absolute score; the shared 50 cancels between numerator and denominator, and the ratio stays $e^2 \approx 7.4$.

C — e^53 is about 10^23, far below the float32 limit near 10^38; overflow needs logits above roughly 88, and the max-subtraction trick removes even that.

D — Invents a mechanism; a constant shift multiplies every exponential by the same factor, which the normalisation removes.

47. The log-sum-exp trick, and why exp(1000) is not a number — D

A — The function expects raw logits, not log-probabilities; it applies log-softmax itself, and the fix is to remove the softmax rather than add a log.

B — Sigmoid is for independent binary outputs; the problem here is a duplicated normalisation, not the choice of activation.

C — Diagnoses a plateau as an optimisation setting; the ceiling comes from feeding numbers in [0,1] into a second softmax, which flattens every prediction.

48. Power laws, and the straight line on a log-log plot — A

B — Reads the slope as a linear change in the error rather than an exponent; on log-log axes the slope multiplies, it does not subtract.

C — Applies the growth factor directly; the factor is raised to the exponent, and 4 to the power −0.5 is one half, not one quarter.

D — The intercept sets the absolute error, but the ratio between two sizes depends only on the slope.

49. An estimator is a recipe, and why the variance divides by n − 1 — A

B — Reverses the defaults: NumPy's ddof is 0 and pandas' is 1, which is exactly why the same column yields two numbers.

C — A ratio of exactly 10/9 is not rounding; it is the n versus n − 1 denominator, and both libraries are correct by their own definition.

D — Pandas does skip NaNs, but a column with no missing values still gives the n − 1 result; the denominator differs by design, not by data.

50. The law of large numbers, and how fast it works — C

A — Ignores the correlation entirely; variances of correlated draws do not simply add, and the $\sigma/\sqrt{n}$ formula assumes they do.

B — Too pessimistic: a correlation of 0.5 leaves half of each extra answer's information intact, so the effective n sits well above the user count.

D — Applies the correlation as a simple discount; the inflation grows with cluster size, $1 + (m - 1)\rho$, and with 25 per cluster it is 13, not 2.

51. Why averages look normal, and when they refuse to — B

A — Applies a rule of thumb for mild skew to a heavy tail; the convergence depends on the shape, and 50 skewed latencies can be far from a bell.

C — Confuses the data with the sampling distribution of the average; the theorem exists precisely because non-normal data can have a normal mean.

D — A log transform changes what you are averaging; the mean of the logs may be well behaved, but it is a different quantity from the mean latency.

52. Standard error for a mean and a proportion, by hand — D

A — Uses a single estimate's error as if it were the error of the difference; the uncertainty of a gap is larger than either side's alone.

B — Correct for separate test sets, but these models were scored on the same items, where shared difficulty cancels and this figure overstates the noise.

C — Confuses the smallest expressible difference with the noise level; the standard error, not the granularity, sets what can be detected.

53. What a 95 per cent interval promises, and where the textbook one breaks — B

A — Fifty items cannot establish certainty; the zero width comes from plugging the estimate into the error formula, not from the evidence.

C — Small samples give wide intervals, not undefined ones; the Wilson recipe returns a perfectly usable range from fifty items.

D — The t-correction addresses a noisy estimate of σ for a mean; for a proportion at p = 1 the standard error is zero under either constant, so this changes nothing.

54. Zero failures in 300 tries, and what that does not prove — B

A — Treats the point estimate as the truth; zero observed failures is consistent with any rate small enough that 200 trials would likely miss it.

C — Uses 1/n as the bound; the 95 per cent bound is about 3/n, because rates up to that level still leave a reasonable chance of a clean run.

D — Zero is informative: it rules out high rates, and the rule of three says exactly how high.

55. Updating a rate as evidence arrives: the Beta distribution — D

A — Ignores the prior entirely; the whole point of the pseudo-counts is that one view cannot establish a rate.

B — A prior of ten pseudo-views is easily outweighed by a hundred real ones; it dominates only where the data are thin, as for A.

C — Backwards: fewer observations mean a wider posterior, and the prior pulls a thin estimate toward its own mean.

56. Regularisation is a prior: L2, L1 and the diamond — A

B — Strength is not the mechanism; a large L2 λ shrinks weights aggressively and still never reaches zero, because its gradient is proportional to the weight.

C — Backwards: squaring makes small weights cheaper, not dearer, which is exactly why L2 stops caring about them near zero.

D — Nothing is rounded afterwards; the zeros arise during optimisation because a constant pull toward zero beats a data pull weaker than λ.

57. Regression to the mean, and the winner's curse in model selection — C

A — The model never trained on the validation items; the optimism comes from choosing among noisy scores, not from memorisation.

B — Assumes a bias with no evidence; a 2.5-point drop is exactly what the expected maximum of 100 draws at a 1-point standard error predicts.

D — Tuning anything on the test set would make its score optimistic in the same way; the test number is trustworthy precisely because nothing was chosen on it.

58. Big-O, and the four growth rates you will meet — B

A — Assumes linear growth; doubling the input quadrupled the time, which is the signature of n^2, not n.

C — Big-O drops constants but the two timings supply them; the shape plus one measured point predicts the scale to within a small factor.

D — Treats a fourfold jump as overhead rather than as evidence of the growth rate; the ratio between the two runs is the whole signal.

59. Counting the floating-point operations in a layer and a training run — B

A — Counts only the forward pass; training also runs the backward pass, which costs about twice the forward, giving six per parameter per token.

C — Underestimates the backward pass, which computes two gradients per layer and costs about twice the forward, not the same as it.

D — The parameter count already summarises the architecture; the FLOP rule needs only parameters and tokens, with attention as a modest correction.

60. Anything pairwise is quadratic: distances, attention and deduplication — B

A — Attention cost per token grows with L while the linear layers do not, so the fraction cannot be fixed.

C — The absolute cost of attention does rise about 25-fold, but the linear layers stay put, so the share rises too; the crossover is at L = 6d.

D — Each token scores against every other, so a longer context means more work per token, not less.

61. Bytes per parameter, and the arithmetic of what fits on a card — D

A — Data streams through in batches; the footprint that breaks the card is per-parameter state and activations, not the dataset.

B — Gradients add the size of the weights, but the master copy and the two Adam moments are the larger part; the total is eight times the fp16 weights, not two.

C — Mixed precision keeps an fp32 master copy alongside fp16 weights rather than replacing them, and that copy is only a quarter of the training footprint.

62. Why a GPU idles: memory-bound versus compute-bound — B

A — The FLOP count per token is unchanged, since every parameter still does one multiply-add; the speed-up comes from the other ceiling.

C — 3.5 GB fits no cache; the weights still stream from main memory each token, and it is the byte count over that link that fell.

D — Invents a parallelism that quantisation does not provide; the batch is still one, and the gain is purely fewer bytes per token.

63. Hashing, collisions, and the square-root law behind them — D

A — Eviction causes recomputation, not wrong answers; returning another request's response is the signature of two keys mapping to one slot.

B — Even a perfectly uniform 32-bit hash collides by the birthday bound; the fault is the key size, not the function's quality.

C — Confuses the number of possible values with the number of draws before a repeat; the latter is the square root, tens of thousands, not billions.

64. A graph is a matrix, and its paths are its powers — C

A — Powers of the matrix are not powers of the entries; each multiplication composes a step, so A^3 is about three-step walks.

B — That would be a row sum over a reachability matrix; a single entry concerns one destination, node 5, not a count of nodes.

D — Probabilities come from the transition matrix, rows divided by degree; the raw adjacency matrix counts walks rather than weighting them.

65. Finding the closest vector among ten million: the arithmetic of approximate search — A

B — 40,000 dot products of length 768 is about 60 million FLOPs, well under a millisecond on a laptop; the quadratic fear does not apply to a single query against a store.

C — 40,000 × 768 × 4 bytes is about 120 MB, which fits comfortably; memory becomes the constraint around ten million vectors, not forty thousand.

D — HNSW is approximate; it trades a measured recall below 100 per cent for speed, which is a bad trade when brute force was already fast.

66. Estimating anything to within a factor of three — D

A — Assumes every error points the same way; in reality they partly cancel, and the total grows with the square root of the count.

B — Overstates the cancellation; errors do not vanish, they add in log space like a random walk, giving $\sqrt{6}$ steps rather than zero.

C — Adds the factors instead of compounding them; uncertainty multiplies, and the correct compounding uses $\sqrt{6} \times \ln 3$ in the exponent.

67. What a float32 actually stores, and where its seven digits run out — D

A — Float32 reaches 3.4 × 10^38 before overflow, and overflow gives infinity rather than a wrap; the stall is a precision limit, not a range limit.

B — It represents every integer up to 2^{24} exactly and many beyond; what it loses above that point is the odd ones, not integers as such.

C — The suspicious value is exactly 2^{24}, which is the number at which float32 loses the ability to represent odd integers; that coincidence is the diagnosis.

68. bf16, fp16, and why training keeps a float32 copy — A

B — A decayed schedule would be visible in the config; the described symptom appears whenever updates fall below bf16's gap, which happens well before the rate reaches zero.

C — bf16 shares float32's exponent range, so underflow is not its weakness; the gradients are stated to be non-zero.

D — Accumulation inside the multiply is done in float32 by the hardware; the loss of updates happens at the moment of adding a tiny step to a coarse weight.

69. Catastrophic cancellation, and why a sum depends on its order — D

A — Bessel's correction changes the estimate by a factor near 1 and cannot flip its sign; the sign flip is catastrophic cancellation.

B — A variance is a sum of squares and cannot be negative for any real data; a negative result can only come from the arithmetic, not the inputs.

C — Float32 reaches 3.4 × 10^38; the squares are nowhere near overflow. The problem is precision at that magnitude, not range.

70. Integers, overflow, and the token id that would not fit — C

A — Noise would produce small values near zero, not bright ones; the bright artefacts are the signature of wrapping, which turns a small negative into a large positive.

B — NumPy does not promote uint8 minus uint8; it stays uint8 and wraps, which is exactly what produces the artefacts.

D — Misalignment would show structured edges, not speckles across static regions; the pattern here follows the arithmetic, not the geometry.

71. Quantisation is rounding with a scale: int8, int4 and the outlier problem — B

A — That figure assumed the scale was set by 0.02; with the outlier the scale is 0.039, two hundred times coarser.

C — Clipping happens only if the scale is chosen below the maximum; with the scale set from the maximum, the outlier is exact and everything else pays.

D — Rounding w / scale never exceeds 127 when the scale is max|w| / 127; the damage is loss of the small values, not overflow of the large one.

72. The condition number, and how many digits a problem throws away — D

A — Collinearity is a different route to ill-conditioning; here the cause is a scale ratio of 10^9 between columns, which happens even when the features are unrelated.

B — Gradient descent sees the same Hessian and the same condition number, and would take on the order of κ steps; rescaling the feature is the fix under either solver.

C — An outlier moves the fit, but a well-conditioned fit moves it by a bounded amount; wild swings from a single ordinary edit are the mark of a condition number that has consumed every digit.

73. How variance travels through a layer, and the initialisation that follows — A

B — The symptom is present before any update; it is the forward pass at initialisation that has produced saturated logits.

C — Vanishing would give a loss near ln(classes) and tiny gradients; a huge loss with gradients of ±1 is the saturated, exploded case, and 0.05 exceeds the $\sqrt{(2/2048)} \approx 0.031$ that would keep variance level.

D — The halving is real but is exactly what the factor of 2 in the He rule accounts for; it does not by itself produce a huge loss.

74. Random numbers that are not, and what a seed does and does not fix — C

A — Different initial weights would produce entirely different models, not ones agreeing to two decimal places; near-agreement is the mark of tiny arithmetic differences compounding.

B — Periods of 2^128 or more cannot be exhausted; and a wrapped sequence would still be identical between two seeded runs.

D — Dropout masks come from the seeded generator and are identical across the two runs; the nondeterminism is in the summation, not the sampling.

75. Checking a gradient with finite differences, and the step size that lies — D

A — At h = 1e-6 in float32 the rounding term ε/h is itself about 0.1, so the check cannot distinguish a correct gradient from a wrong one.

B — Kinks affect a handful of parameters whose set changes with h or input, not all of them uniformly; a uniform 0.1 is the signature of precision, not of kinks.

C — Smaller h makes the rounding term ε/h larger still; the optimum is $\varepsilon^{(1/3)}$, about 5e-3 in float32, and even there only five digits survive.

76. NaN, Inf, and the five numbers to print about any tensor — B

`A` — The NaN is the last symptom; the gradient norm shows the blow-up starting fifteen steps earlier, and the loss only inherited it.

`C` — A corrupt batch produces a NaN with no warning in the preceding steps; a norm that grows for fifteen steps is an optimisation instability, not a bad row.

`D` — Changing the seed hides the symptom without finding the cause; the runaway norm will recur at some other step because the dynamics, not the sample, are unstable.

Module test — 1. Vectors, and the space they live in

1. A

The dot product equals the length of one vector times the length of the other times the cosine of the angle, so it mixes agreement with size. A six-thousand-word page has a long vector and scores high against every query, whatever it says. Dividing by both lengths cancels the magnitudes and leaves the cosine, which depends on angle alone.

2. B

The spread of the cosine between two random vectors is about one over the square root of the dimension: 0.051 at 384 dimensions and 0.026 at 1,536. A cosine of 0.35 is therefore about seven spreads out in the first space and about thirteen in the second, so the same number is a much stricter demand. The rule gives the direction; the threshold itself still has to be measured on labelled pairs from the model you are actually using.

3. C

L2 of A is the square root of 100, which is 10, and L2 of B is the square root of 36, which is 6, so L2 calls A larger. L1 gives 10 against 12 and calls B larger. Squaring says one deviation of ten is worse than four of three, which is exactly why squared-error loss is dragged around by a single wild data point and absolute-error loss mostly ignores it.

4. B

Projection removes precisely the component along the direction you named, and afterwards every vector's dot product with that direction is zero. If the attribute is spread across a subspace of many correlated directions, or is encoded non-linearly, everything outside that one direction survives untouched. The arithmetic did what was asked; the ask was too small.

5. C

Rotate every embedding by an orthogonal matrix and rotate the next layer's weights back by its inverse, and every dot product, distance and output is identical, so the loss cannot prefer one rotation over another. Training therefore lands on an arbitrary basis, different for every seed. Coordinates are relative to a basis; distances and neighbourhoods are not, so build what you rely on out of the second kind.

6. D

An embedding is a lookup of learned rows. A code the tokeniser has to split into fragments it has seen almost nothing about produces a direction close to noise, whatever the model's quality. A sparse index does not represent the code at all: it matches it. The two systems fail in complementary ways, which is the argument for running both and merging their rankings by position.

Module test — 2. Matrices, and what a layer really does

1. A

Applying A and then B is exactly applying BA, because matrix multiplication is the operation of composing linear maps. So a hundred stacked matrices are one matrix, for every possible input, and all those parameters buy nothing. Depth starts paying only when something a matrix cannot do is wedged between the layers, which is what an activation function is.

2. B

Broadcasting pads the shorter shape on the left, so (4,) becomes (1, 4), and a dimension of size one is stretched to match its neighbour. (1, 4) against (4, 1) gives (4, 4): sixteen differences where four were wanted. Nothing raises an error, the loss decreases, and the model learns slowly and wrongly. One assertion that the two shapes are equal, placed before the loss, removes the whole class of bug.

3. C

The determinant is the factor by which a transformation multiplies volume, so zero means the space was squashed onto something of lower dimension. Two different inputs then produce the same output, the map has no inverse, and the information is gone. Nothing downstream can restore a dimension that was destroyed, which is why determinant zero, no inverse and information lost are three statements of one fact.

4. D

The product of a 4096 by 8 and an 8 by 4096 matrix has rank at most 8, so whatever goes in, what comes out lies in an 8-dimensional subspace of a 4096-dimensional space. That is fine when the change a task needs really is low rank, which is the assumption behind low-rank fine-tuning. When such a fine-tune underperforms, raising the rank tests the assumption directly rather than guessing at hyperparameters.

5. C

Squared singular values sum to the total squared magnitude, so needing 620 of 768 directions to reach 90 per cent means the matrix is spread almost evenly across its width. Truncating the singular value decomposition is provably the best rank-k approximation that exists, so if it is not good enough, no low-rank method is. The check is four lines and belongs before the attempt, not after it disappoints.

6. A

As the coefficients vary, Xw traces out the column space of X: a flat surface sitting inside the data's space. The closest point on a flat surface to a point outside it is the foot of a perpendicular, so the best Xw is the projection of y and the residual must be perpendicular to every column. Rearranging that single condition gives the normal equations directly, and the word normal there means perpendicular rather than typical.

Module test — 3. Change, slopes and the walk downhill

1. B

The derivative of the loss with respect to a first-layer weight is a product of one factor per layer and one per activation, about eighty in all. Factors of 0.25 give 0.25 to the fortieth, which is indistinguishable from zero in float32, so no signal arrives. ReLU contributes a factor of exactly one for positive inputs and a residual connection provides a path with factor one whatever the layer does, which is why both exist.

2. C

Backpropagation caches on the way forward and reuses on the way back: computing a layer's weight gradient needs that layer's input, which was produced long before. So peak memory scales with depth times batch size times activation size, on top of the weights and optimiser state. That is why reducing the batch is the first answer to an out-of-memory error, and why gradient checkpointing trades about a third more compute for a large fall in memory.

3. B

A gradient norm of exactly zero is not a small signal but no signal: either the loss is not connected to the parameters through the graph, or every unit in some layer outputs zero for every example and so has zero derivative. A rate that is too high gives a large and rising norm followed by a NaN, and noise from a small batch makes the norm fluctuate rather than vanish. The three cases are distinguished by that one logged number.

4. C

For a flat point to be a local minimum, the surface must curve upward along every one of the Hessian's eigen-directions, which is a million conditions holding at once. A point that curves up in some directions and down in others is a saddle, and gradient descent simply follows a downward direction out of it. This is the best current explanation rather than a proof: deep networks are not convex and carry no convergence guarantee at all.

5. B

Along the narrow direction successive gradients point opposite ways, so terms in the running average cancel and the oscillation is damped. Along the long direction they point the same way at every step, so they accumulate towards one over one minus beta, which at 0.9 is ten times a single gradient. Dividing by a gradient history is a different correction: that is what Adam's second moment does.

6. D

The steepest part of the descent is where the rate is doing the most work, while the minimum of the range-test curve sits at the edge of instability. A run that survives two hundred steps there frequently fails at twenty thousand, because the curvature the optimiser meets grows sharper as the loss falls. Warmup and clipping reduce the risk; they do not license taking the edge.

Module test — 4. Probability you can rely on

1. B

Out of 100,000 transactions there are 100 fraudulent ones, of which 95 are caught, and 99,900 legitimate ones, of which one per cent, 999, are wrongly flagged. That is 1,094 flags with 95 real, about 9 per cent. The small error rate acts on the far larger group, so it dominates the total, and nobody made a mistake to produce it.

2. C

Pearson measures only how well a straight line describes a relationship. For y equal to x squared with x spread symmetrically about zero, the best straight line is flat and the correlation is exactly zero, though x determines y precisely. Spearman catches monotone curves, mutual information catches any dependence at all, and a scatter plot catches every one of them in five seconds.

3. D

The Poisson's distinguishing property is that its variance equals its mean, which gives you a free diagnostic on any count data. A variance six times the mean means the independence assumption is broken, most often because events arrive in bursts. This is overdispersion, it is extremely common in real systems, and ignoring it produces alert thresholds that fire constantly.

4. C

The chance that all twenty are fast is 0.99 to the twentieth, which is 0.818, so the chance at least one is slow is 18.2 per cent. A one per cent tail at the service level is an eighteen per cent tail at the page level. That is why serious latency work targets the 99th and 99.9th percentiles, and why an average latency on a dashboard is close to useless as a measure of experience.

5. B

Variances of independent draws add, so the variance of an average of n falls as one over n and the spread as one over the square root of n. Four times the batch halves the noise rather than quartering it. That square root is why very large batches stop helping long before they stop costing, and it is the same arithmetic that governs how many test items an accuracy figure needs.

6. C

The quoted figure conditions on fraud and counts flags; the number you act on conditions on a flag and counts fraud. Swapping them is the prosecutor's fallacy, and it appears constantly in reasoning about model outputs. Converting between the two needs the base rate, and when the thing being detected is rare, the second number is far smaller than the first.

Module test — 5. Logarithms, information and the shape of a loss

1. B

Adding a constant to every logit leaves softmax unchanged, because the shared exponential factor cancels between the numerator and the denominator. Subtracting the maximum makes the largest exponent exactly zero, so the largest exponential is exactly one and nothing can overflow; the smallest underflow to zero, which contributes nothing anyway. On logits of 1000 and 999 it is the difference between the right answer and NaN.

2. C

The entropy of the label column is the cross-entropy of a model that knows nothing but the frequencies: minus 0.1 times log 0.1 minus 0.9 times log 0.9, which is 0.325 nats. Any loss above that is worse than a constant predictor. Compute the number before training starts and draw it on the plot, because a curve without it can look like progress while being none.

3. B

Cross-entropy equals the data's entropy plus the KL divergence between the data and the model. Training against fixed data cannot move the first term, so it can only shrink the second, and the loss can never go below the data's own entropy. KL is never negative, so nothing cancels the entropy, and a plateau above zero may be the floor rather than a failure.

4. B

If the model gave the correct class 0.01, the push on its logit is 0.99, nearly the largest possible; if it gave 0.999, the push is 0.001. The size of the step is the size of the mistake. That is the mechanism behind the usual shape of a loss curve: fast early, when everything is wrong, and slow late, when the remaining errors are few and each carries a small gradient.

5. C

Every standard loss is minus the log-likelihood under a stated noise assumption, and squared error is the Gaussian one. Under a Gaussian an error ten times the typical size has a probability of order e to the minus fifty, so the fit distorts itself to avoid it. Absolute error corresponds to a Laplace, whose fatter tails expect the occasional large error, which is why it is robust to outliers rather than merely different.

6. A

An exponential average with factor beta weights a value k steps back by beta to the k, and the effective window is about one over one minus beta: ten steps at 0.9 and a thousand at 0.999. The second moment estimates a variance, and a variance from ten samples is noisy enough to make the step size lurch, so it is given a longer memory. The same arithmetic explains why both moments need a bias correction at the start.

Module test — 6. Estimation: from a sample to a number you can trust

1. B

The sample mean is by construction the point minimising the sum of squared distances to the sample, so distances measured from it are systematically smaller than distances from the true mean. One degree of freedom was spent estimating the mean. With two-point samples drawn from the values 0 and 2, dividing by n gives estimates averaging 0.5 against a true variance of 1, while dividing by n minus one averages to exactly 1.

2. C

The design effect is one plus the cluster size minus one, times the correlation: one plus 24 times 0.5 is 13, and 2,000 divided by 13 is about 154. Any interval computed as if n were 2,000 is roughly three and a half times too narrow. Counting only the 80 users is too harsh, because answers within a user still carry some independent information.

3. D

Zero failures in n independent trials bounds the rate at roughly three over n with 95 per cent confidence, so 300 clean runs are consistent with one failure in a hundred. On a system handling a thousand requests a day that is ten failures a day the test could not have seen. Zero is real evidence, and the honest report is the bound rather than the point estimate.

4. C

The recipe estimates the standard error as the square root of p times one minus p over n, and at p equal to one that is exactly zero, so the interval collapses to a point. Near the boundary the estimate is a poor stand-in for the truth. Wilson solves instead for the true rates under which fifty out of fifty would be plausible, and gives 0.929 to 1.000, which is the honest answer.

5. A

The derivative of the L2 penalty is lambda times w, which vanishes as w approaches zero, so the penalty stops pulling exactly when it would need to finish. The derivative of the L1 penalty is lambda times the sign of w, a constant force regardless of size, so a weight whose data pull is weaker than lambda cannot leave zero. The closed form is soft thresholding: shrink by lambda, and stop at zero if that would cross it.

6. C

The expected maximum of a hundred draws sits about two and a half standard errors above the truth, so roughly two and a half points of the 91 were selection on favourable noise. Nothing happened to the model; the luck simply does not repeat on new items. The only unbiased number comes from a set nothing was selected on, and the count of candidates is part of the claim rather than a footnote.

Module test — 7. Counting the cost: complexity and the arithmetic of scale

1. B

Checking membership in a list walks it from the start, so inside a loop over n items that is n scans of up to n elements and the work grows with the square. A set hashes the item and looks in one bucket, which does not depend on how many items are stored, so the loop grows linearly. Big-O tells you whether an approach survives scale, and no constant factor rescues a quadratic at a million.

2. A

A training step costs about six FLOPs per parameter per token: two for the forward pass and about four for the backward, which computes an input gradient and a weight gradient per layer. Six times thirteen billion times two trillion is 1.56 times ten to the twenty-third. The forward-only figure would be a third of that, and for ordinary context lengths the architecture only matters at the twenty-per-cent level.

3. D

Generating one token multiplies every weight matrix by a single vector: two FLOPs for every two bytes read, an arithmetic intensity of about one, far below any modern chip's ridge point. The time is the weight bytes divided by the bandwidth, so a quarter of the bytes is a quarter of the time whether or not the arithmetic units are faster. For large-batch throughput, which is compute-bound, the gain depends on whether integer matrix units exist.

4. C

The number of pairs is about n squared over two and each collides with probability one over m, so the expected count is n squared over 2m: a million squared over 8.6 billion, roughly 116. Collisions become likely once n reaches the square root of the space, about 77,000 items for 32 bits, not four billion. A 64-bit digest puts the same corpus at a collision probability of about three in a hundred million.

5. B

Set four L d equal to 24 d squared and L comes out at six d, which for a width of 4,096 is 24,576 tokens. Below that attention is the smaller cost; at 4,096 tokens it is about a sixth of the linear layers. The memory is worse than the FLOPs, since the score matrix is L by L per head, which at 32,768 tokens in half precision is 2.1 GB per head per layer and never fits.

6. C

In logs each guess is a step of size log three in a random direction, and k random steps travel about the square root of k steps in total, so the product is uncertain by roughly three to the power of the square root of six, about 15. Six careless guesses still land within about an order of magnitude, which is usually enough to decide, and the estimate names the factor worth going away to measure.

Module test — 8. Numbers inside a machine

1. B

A float32 carries 24 bits of mantissa, so it represents integers exactly only up to two to the 24, which is 16,777,216. Past that the gap between representable values is 2, and adding 1 rounds straight back down. Nothing overflowed: the largest float32 is about 3.4 times ten to the 38, and the format has simply run out of resolution. Keep counters as integers.

2. C

Subtracting two nearly equal floats cancels the digits they agree on and promotes their rounding error to the front. Both terms here are about ten to the eighth and each is known to about eight units, while the answer is supposed to be around one. Subtract the mean first and then square, so the numbers being squared are near one and there is nothing large left to cancel.

3. A

bf16 keeps float32's range and pays with about two decimal digits, so near one the spacing between representable numbers is 0.0078. An update of 0.0001 added there disappears entirely, and small updates are the normal case late in training when the learning rate has decayed. The master copy holds the truth at a spacing near ten to the minus seven, and the half-precision weights are a working copy rounded from it.

4. C

A unit's output variance is fan_in times the weight variance times the input variance: 1,024 times 0.0025 is 2.56, and the ReLU roughly halves it, giving about 1.28 a layer. Twenty layers multiply the variance by 1.28 to the twentieth, about 139, and sixty layers by a million. The value that keeps it level is the square root of two over fan_in, which here is 0.044, and the small difference compounds.

5. D

NaN propagates through everything it touches, so where it appears is downstream of where it was made. The usual origins are a log of zero, a square root of a small negative left by a cancelled variance, a division by a zero norm, and an overflow after several steps of a growing gradient. The first non-finite loss normally arrives a few steps after the first non-finite gradient, so that history is the evidence.

6. B

Solving a system with condition number kappa loses about log base ten of kappa digits, and float32 has about seven. The matrix built from these two columns has one direction stretched by roughly ten to the twelfth relative to the other, so nothing survives. Subtracting a mean and dividing by a spread takes the condition number under about ten. The small feature may carry real signal: it was the arithmetic destroying it, not the data.

The Maths You Actually Need — final examination

1. D ***1. Vectors, and the space they live in***

For unit vectors the squared distance between a and b expands to two minus twice their dot product, and the dot product of two unit vectors is the cosine of the angle. Squared distance is therefore a strictly decreasing function of cosine, so sorting by cosine descending and by distance ascending gives the same order, not a similar one. The choice only matters when the lengths are not one.

2. A ***1. Vectors, and the space they live in***

Multiplying a one-hot vector by a 50,000 by 768 matrix selects exactly one row and discards the rest, so the whole operation is a row lookup that somebody replaced with an array index. That is why the rows can be learned, why related words end up with related rows, and why a word costs 768 numbers rather than 50,000.

3. C ***1. Vectors, and the space they live in***

Many trained models push their vectors into a narrow cone, so the average cosine between unrelated texts sits well above zero. Nothing is broken: the offset carries no information and the spread around it carries all of it. A threshold is a claim about one model's output distribution, and the only defensible way to set one is to score a few hundred labelled pairs with that model and read the cut-off from the two distributions.

4. B ***1. Vectors, and the space they live in***

The scaling parameters are part of the fitted model. Computing them across the whole dataset means every test row has quietly contributed to a number the model uses, so the test score stops being an estimate of performance on data the model has never met. Fit the scaler on the training rows and apply it unchanged to everything else.

5. C ***1. Vectors, and the space they live in***

Distance concentration is a statement about points spread through the whole space. A trained model maps its inputs onto a curved surface whose effective dimension is often in the tens, so its points are not spread through the whole space. That is also why approximate indexes work on real data and degrade towards brute force on random vectors, which is why a synthetic benchmark tells you very little.

6. A ***1. Vectors, and the space they live in***

The inverse document frequency of a term is the log of the number of documents over the number containing it, so a term in one document in a thousand gets a weight near 6.9 while a term in every document gets zero. A code is exactly that kind of rare term, and a sparse index matches it without ever having learned it. A dense model must have learned the token to represent it at all.

7. B ***1. Vectors, and the space they live in***

Addition is commutative, so the average of a set of vectors does not depend on the order they were added in. That makes the summary cheap and useful and blind to arrangement, which is a specific and known limitation rather than a general unreliability. It is used in production more than anyone admits, and knowing exactly what it throws away is what makes that defensible.

8. A ***2. Matrices, and what a layer really does***

Matrix multiplication in NumPy and PyTorch acts on the last two dimensions and treats everything in front as batch, so the same 768 by 3072 matrix is applied to each of the 32 times 128 individual vectors with no loop written anywhere. The inner dimensions must match and vanish; the outer ones survive.

9. D ***2. Matrices, and what a layer really does***

Reshape reinterprets the same numbers in the same memory order under a new shape; transpose genuinely moves numbers between positions. Both target shapes hold the same element count, so the wrong one is legal and silent. It puts different tokens where different features should be, and the model still trains, just towards a worse function with nothing to chase.

10. C ***2. Matrices, and what a layer really does***

A solver factorises the matrix and works with the factors, never building the inverse at all. Forming the inverse costs roughly three times the work of solving directly, and it produces an intermediate object carrying its own rounding errors before that object is used. Library documentation steers you away from inv for exactly these two reasons.

11. B ***2. Matrices, and what a layer really does***

When two columns are nearly copies, the matrix built from them cannot tell them apart, its condition number rises without limit, and small changes to the data swing the pair of coefficients wildly while they nearly cancel. The fitted surface can still predict well. What is not available is an interpretation of either coefficient, and a ridge penalty bounds the condition number by pushing the smallest singular values away from zero.

12. D ***2. Matrices, and what a layer really does***

Write any vector as a mixture of eigenvectors: applying the matrix n times scales each part by its eigenvalue to the n, so the largest eigenvalue takes over completely. A spectral radius above one blows the signal up and below one shrinks it to nothing, and the difficulty of holding it near one over long sequences is a large part of why LSTMs and later transformers exist.

13. B ***2. Matrices, and what a layer really does***

The error of a rank-k truncation is computable in advance as the sum of the squares of the singular values you discard, which is unusual and useful. It also settles the question of alternatives: if truncated SVD is not good enough, no low-rank approximation is. What it does not tell you is the task metric, since fidelity to the numbers and usefulness for your problem are different quantities.

14. A ***2. Matrices, and what a layer really does***

R-squared compares the residual sum of squares with the variation around the mean of the target, computed on the very data the model was fitted to. An extra column can only give the fit more room, so the residuals cannot get worse and the statistic cannot fall. Adjusted R-squared penalises the count imperfectly; the version worth quoting is computed on data the model has not seen.

15. C ***3. Change, slopes and the walk downhill***

Two errors pull against each other. Truncation error, from using a finite gap, falls as h shrinks. Rounding error, from subtracting two nearly equal numbers and dividing by twice h, grows as machine epsilon over h. They balance near the cube root of epsilon, about six in a million for float64 and five in a thousand for float32, and past that smaller is worse.

16. B ***3. Change, slopes and the walk downhill***

Pre-training searches for a solution; a fine-tune adjusts one that already exists, and a large step destroys what the pre-training built. Typical figures are 1e-4 to 1e-3 from scratch with AdamW and 1e-5 to 5e-5 for a full fine-tune. Adapters sit higher again, near 1e-4 to 1e-3, because they start from zero and carry their own scale.

17. D ***3. Change, slopes and the walk downhill***

At step one Adam's second moment is built from a single squared gradient, so the per-parameter step sizes rest on almost no evidence. Bias correction removes the systematic part of that error but not the variance. Warmup avoids large steps while the estimates are still noise, and for a pre-trained model it also protects the weights from the large gradients a freshly initialised head produces.

18. A ***3. Change, slopes and the walk downhill***

Two moments at four bytes each is eight bytes per parameter, which for seven billion parameters is 56 GB. With 28 GB of weights and 28 GB of gradients that is 112 GB before a single activation is stored. This is the arithmetic behind memory-efficient variants: eight-bit Adam quantises the two states, and SGD with momentum keeps only one of them.

19. B ***3. Change, slopes and the walk downhill***

Swap two hidden units along with their weights and the network computes the same function from different parameters, so a layer of n units has n factorial equivalent solutions, and a non-linear activation removes convexity entirely. Different seeds land in different basins with genuinely different behaviour at the edges. That is also why reporting one run is reporting one draw.

20. C ***3. Change, slopes and the walk downhill***

A high gradient norm at a plateau means the surface is not flat: the steps are crossing a narrow direction and cancelling rather than advancing along the valley floor. Decaying the rate shortens the crossing steps and lets the long direction make progress, which is the main reason cosine schedules are the default. A plateau with a norm near zero would be an entirely different problem.

21. A ***3. Change, slopes and the walk downhill***

Clipping divides every component by the same factor when the norm is above the threshold, so the vector points exactly where it did and is merely shorter. That is what makes it safe: it removes the size of an exploding step without changing which way the step goes. Clipping each component separately would change the direction, and is a different operation with different behaviour.

22. C ***4. Probability you can rely on***

Training data of order ten to the thirteenth tokens is a vanishing fraction of ten to the ninety-fourth, so no model has looked its answer up, and it cannot score all continuations either. Decoding therefore proceeds one token at a time, which is why the sequence a model produces is generally not the highest-probability sequence available to it.

23. B ***4. Probability you can rely on***

A probability is a stated degree of belief, and thirty per cent things happen thirty per cent of the time, so one outcome cannot refute one number. The test is in bulk: gather every case she called thirty per cent and count how many came true. The same test applies to a model unchanged, and the question worth asking of any system reporting probabilities is whether anyone has plotted the buckets.

24. D ***4. Probability you can rely on***

Each factor is a probability below one, and five hundred of them multiply to something near ten to the minus five hundred, far below the smallest number float32 or even float64 can hold: the value becomes exactly zero and every comparison after it is meaningless. Logs turn the product into a sum of five hundred moderate numbers, which a float handles without difficulty.

25. A ***4. Probability you can rely on***

The estimate is a proportion from independent trials, so its standard error is the square root of p times one minus p over n. Shrinking that by a factor of ten needs a hundred times the trials. Cheap to be roughly right and expensive to be very precise is the whole cost model of simulation, and it is the same square root that governs how many test items an accuracy needs.

26. B ***4. Probability you can rely on***

A full covariance in d dimensions has d times d plus one, over two, free parameters, which at 768 dimensions is 295,296 numbers, and estimating them well needs far more than 768 data points. That count, and no deeper principle, is why diagonal covariance is so common: it is the version you can estimate from a realistic sample. Assuming it is a claim that the features are uncorrelated.

27. C ***4. Probability you can rely on***

Expectation is every possible value weighted by how likely it is, and it need not be a value that can occur: he never sells thirty bowls, just as a die never rolls 3.5. It is a summary of a distribution, and summaries are allowed to be things that never happen. Every loss you minimise is an expectation of this kind, estimated from a batch that stands in for all future data.

28. D ***4. Probability you can rely on***

Conditioning shrinks the space to the cases consistent with what you know: sixty-five tickets were resolved on the first reply and forty-five of them were billing, so the answer is 45 over 65, or 0.69. The 0.75 answers the opposite question, the probability of resolution given billing, and swapping the two is the commonest error in applied probability.

29. B ***5. Logarithms, information and the shape of a loss***

Perplexity is e to the loss when the loss is in nats, so 3.0 gives 20.1: the model is about as uncertain as somebody guessing uniformly among twenty options. It also explains why a loss curve that looks flat near the end still delivers real gains, since each unit of loss is the same multiplicative change: 4.0 to 3.0 is 54.6 down to 20.1, and 3.0 to 2.0 is 20.1 down to 7.4.

30. D ***5. Logarithms, information and the shape of a loss***

Two nats is 2.89 bits, and any distribution can be turned into a code where a message of probability p takes about minus log base two of p bits, which is what arithmetic coding does. So 2.89 bits times a billion tokens is 361 megabytes. The honest caveat is that the decompressor needs the model, which is far larger; the claim that stands is that per token, a good predictor and a good compressor are the same object.

31. C ***5. Logarithms, information and the shape of a loss***

Entropy is average surprise, largest when nothing is predictable, so a uniform distribution over k outcomes has entropy log base two of k, which is 2 bits for four outcomes. The 0.5, 0.25, 0.125, 0.125 distribution comes to 1.75 bits and the nearly certain one to 0.242, because its rare outcomes cost 6.6 bits each and almost never happen. Entropy is zero only when one outcome has probability one.

32. A ***5. Logarithms, information and the shape of a loss***

Temperature rescales the gaps between logits, and only the gaps matter, because softmax cannot see a shift shared by every logit. Dividing by 0.5 doubles every gap, so 2, 1, 0 becomes 4, 2, 0 and the output moves from 0.665, 0.245, 0.090 to 0.867, 0.117, 0.016. As the temperature approaches zero the largest logit takes everything; as it grows the output tends to uniform.

33. B ***5. Logarithms, information and the shape of a loss***

With a one-hot target, cross-entropy collapses to minus the log of the probability on the correct class, which is zero only at probability one, and a softmax reaches one only in the limit of an infinite logit gap. So the gradient never quite vanishes and the model keeps pushing. Replacing the target with something like 0.97, 0.01, 0.01, 0.01 gives it a finite optimum, which is what label smoothing is.

34. C ***5. Logarithms, information and the shape of a loss***

The natural log of twelve is 2.4849, the loss of a model whose output is uniform. Probabilities all lie between zero and one, so a loss function treating them as logits applies a second softmax to numbers differing by at most one and flattens the output almost to uniform. The model trains slowly towards a ceiling it cannot pass, and the stall just below the log of the class count is the fingerprint.

35. A ***5. Logarithms, information and the shape of a loss***

Exponentiating either value underflows to exactly zero, so the direct route returns the log of zero. Log-sum-exp factors out the larger term: the answer is the maximum plus the log of the summed exponentials of the differences, here minus 700 plus the log of 1.368, or minus 699.69. Taking the maximum alone would discard a real contribution of 0.31 nats, and this identity is what makes probabilistic computation over long sequences possible.

36. B ***6. Estimation: from a sample to a number you can trust***

The standard error of a proportion is the square root of p times one minus p over n: the square root of 0.85 times 0.15 over 200, which is 0.0252, or 2.5 points. Two of those either side gives 80 to 90, for a number that was reported to one decimal place. The same 85 on 50 items would give 75 to 95.

37. D ***6. Estimation: from a sample to a number you can trust***

Item difficulty is shared when both models see the same items, and the shared part cancels in the difference, so only the items they disagree on carry information about the gap. The standard error of the paired difference is about the square root of the discordant count over n, here root 30 over 200, or 2.7 points, against 3.4 for independent sets. The gap is still inside its own uncertainty, so nothing has been shown.

38. A ***6. Estimation: from a sample to a number you can trust***

The same algebra that bounds a rate after zero failures answers the planning question: to have a 95 per cent chance of seeing at least one event of rate p you need about three over p trials. At one in a thousand that is three thousand. With a budget of a thousand runs you are blind to anything rarer than about one in three hundred, and the honest report says so rather than quoting a zero.

39. C ***6. Estimation: from a sample to a number you can trust***

A Beta prior behaves as pseudo-counts: Beta(2, 8) is ten imaginary views at a twenty per cent rate, and observing k successes and m failures updates it to Beta(2 plus k, 8 plus m). One real view is nearly swamped by ten imaginary ones, while a hundred real views hold their ground and the prior fades. This is additive smoothing, and it is in every search engine and recommender, usually without being called Bayesian.

40. B ***6. Estimation: from a sample to a number you can trust***

A confidence interval's guarantee is about the procedure: 95 per cent of the intervals it produces across repeated samples would contain the truth. The credible interval licenses the sentence everyone wants to say about the single interval in front of them, at the price of stating a prior somebody can ask you to defend. Once the data outweigh the prior, the two usually agree to the second decimal.

41. C ***6. Estimation: from a sample to a number you can trust***

Starting from zero weights, gradient descent moves quickly in directions of high curvature and barely at all in directions of low curvature. Stopping after t steps therefore leaves the low-curvature directions near zero, which is the state an L2 penalty of strength about one over the learning rate times t would produce. The number of epochs is a regularisation strength, which is why it gets tuned on a validation set like one.

42. D ***6. Estimation: from a sample to a number you can trust***

Mean squared error is bias squared plus variance, so a slightly biased estimator with a much smaller variance can be closer to the truth on any single sample. The whole of regularisation is that trade taken deliberately: accept a little bias to buy a large reduction in variance. Unbiasedness is one property among several rather than a definition of a good estimator.

43. D ***7. Counting the cost: complexity and the arithmetic of scale***

Anything up to n log n scales to any dataset you will meet: sorting a billion items is about thirty seconds at a billion operations a second. A quadratic at a million items is ten to the twelfth operations, about seventeen minutes, and at a billion it is thirty-one years. That line is what decides whether an approach survives, and no constant factor moves it.

44. B ***7. Counting the cost: complexity and the arithmetic of scale***

Every entry of a matrix product is a dot product, and a dot product of length k is k multiplications and k additions, so an m by k matrix times a k by n matrix costs two m n k FLOPs. In a linear layer each weight touches each token exactly once for one multiply and one add. Training is about three times that, since the backward pass computes an input gradient and a weight gradient per layer.

45. C ***7. Counting the cost: complexity and the arithmetic of scale***

For each token in the context a transformer keeps a key and a value vector at every layer, so it need not recompute them for each new token. At two vectors times 32 layers times width 4,096 times two bytes that is about half a megabyte a token: a 4,096-token context is 2.1 GB and a 32,000-token context is 17 GB, more than the weights, and every simultaneous conversation needs its own.

46. A ***7. Counting the cost: complexity and the arithmetic of scale***

A Bloom filter stores only bits set by hash functions, not the items themselves, so it can report membership and can never enumerate. It also never says absent wrongly, since a present item always set its bits; it can say present wrongly, at a rate the formula lets you buy exactly. A billion URLs at ten bits each is 1.25 GB, against tens of gigabytes for the set itself.

47. B ***7. Counting the cost: complexity and the arithmetic of scale***

Comparing every document with every other is quadratic and impossible past a few hundred thousand. A locality-sensitive scheme is built so that the chance two items share a hash tracks their similarity: MinHash signatures collide with probability equal to the overlap of their word sets. Bucket by signature bands, compare only within buckets, and the pairs that never shared a bucket were with high probability never near-duplicates.

48. C ***7. Counting the cost: complexity and the arithmetic of scale***

Every other component of the vector decays under repeated multiplication while the component along the eigenvector with eigenvalue one does not, so that direction takes over. It is the stationary distribution of the walk. PageRank is exactly this, with one refinement: with probability 0.15 a step jumps to a random page, which stops the walker being trapped in a page with no outgoing links.

49. A ***7. Counting the cost: complexity and the arithmetic of scale***

Eighty thousand vectors of 768 numbers is 246 MB in float32 and about 1.2 times ten to the eighth operations per query, roughly a millisecond on a laptop. An index at this scale is wasted effort that also costs recall, since every approximate method misses the true nearest some measured fraction of the time. Do the two multiplications, n times d and n times d times four, before choosing a tool.

50. C ***8. Numbers inside a machine***

One tenth is not representable in binary, any more than a third is in decimal, so both formats store an approximation. In float64 the sum lands on 0.30000000000000004 and the comparison is false; in float32 both sides round to 0.300000012 and it is true. The lesson is not about 0.3: equality between computed floats is an accident of rounding, so compare with a tolerance chosen from the format.

51. A ***8. Numbers inside a machine***

Once the running total passes a few million, the spacing between float32 numbers exceeds 0.1, so each increment rounds to whatever the local spacing allows and the error is systematic rather than random. Pairwise summation adds neighbours, then pairs of pairs, so no addition combines a huge number with a tiny one, and the error grows like log n instead of n. Accumulating in float64 is the other fix.

52. D ***8. Numbers inside a machine***

The seed fixes the initialisation and the masks; it does not fix the order in which thousands of threads add their contributions to a gradient. Since addition in floating point depends on order, the gradients differ in their last bits at the first step and the runs separate visibly within a few thousand. Deterministic modes force a fixed reduction order, at a cost in speed.

53. B ***8. Numbers inside a machine***

Fixed-width integers are exact inside their range and wrap silently outside it: an int32 past 2,147,483,647 becomes the most negative value and counting continues. NumPy then treats a negative index as counting from the end of the array, which is legal, so the code reads the wrong tokens with no error at all. Count in int64, and read an exact 2,147,483,647 or 65,535 as a limit rather than a measurement.

54. B ***8. Numbers inside a machine***

With a symmetric scale on values that are never negative, half the integer range is wasted, so the asymmetric form adds an offset making the minimum map to 0 and the maximum to 255. Exact zeros are extremely common, from padding and from every ReLU that fired negative, and if the encoding rounds them to something slightly non-zero, that small error is added in the same direction across an enormous number of values.

55. C ***8. Numbers inside a machine***

A ReLU has no derivative at zero: the slope is 0 on one side and 1 on the other. If a nudge of h moves some unit across that point, the two-sided difference averages the two slopes while the analytic gradient picked a side, so the relative error lands near 0.5. The tell is that the disagreeing parameters change when you change h or the input. If every parameter disagreed, the gradient really would be wrong.

56. D ***8. Numbers inside a machine***

A seed makes one shuffle repeatable for one input; add a row and the whole permutation changes, so rows cross between the two halves and the test set is contaminated by things the model has now trained on. Assign each row by a hash of its own id instead, so its side depends on the row alone: new rows land in either split and old rows never move.