

Transformers from First Principles

From one word to designing your own model

Ayush Singh

October 2026

How to read these notes

Why does a transformer look the way it does? You could memorise the diagram. I would rather you walk the road the researchers walked and arrive at it yourself, one failure at a time. Every piece of the transformer (attention, positions, the MLP, the residual stream, normalisation, the KV cache) exists because something simpler broke. So in every chapter we first build the simple thing, watch it break *with actual numbers*, and only then introduce the fix. If at the end of a section you feel “I could have invented this myself”, the section has done its job. Could you have invented attention in 2014, given what people knew then? By Chapter 3 I think you will say yes.

Two readers at once. These notes are written for someone who left school decades ago and for a PhD student, at the same time. Every idea is climbed in three rungs:

- **PICTURE IT:** an everyday picture you can hold in your head. No symbols.
- **SAY IT EXACTLY:** the same idea in mathematics, with every symbol defined and every shape written.
- **WHY IT HAD TO BE THIS WAY:** the general principle underneath, the part that transfers to problems that have nothing to do with language.

If a rung feels too easy, skip it. If it feels too hard, go one rung down and come back. You never need anything you were not given on an earlier page.

How to use the boxes. Each chapter opens with a **map** and a short **Before you start** check of the maths it needs, with a refresher right there; do it, even if it looks familiar. When you see **Predict first**, stop and write down a guess before reading on. A wrong guess is useful: it shows you exactly which idea to look at again. **Where this picture breaks** tells you the limit of an analogy, so the picture does not mislead you later. **A tempting wrong idea** names a mistake many people make, says why it feels right, and shows why it is wrong. **Check yourself** questions have their answers at the end of the chapter; answer first, then look. For a wrong option, the answer says which wrong idea it came from. **Looking back** brings back questions from earlier chapters, because recalling something after a gap is how it stays.

Why so many questions? Because a guess you wrote down, right or wrong, is what makes the answer stick.

Four rules hold throughout.

- (i) **One running problem.** We start from a real message, **waste of time**, and a real job: reply to it well. The same tokens and the same tiny weights ($d = 4$) follow us through every chapter, small enough to compute with a pencil. Later the same steps are replayed on real models, GPT-2 and Qwen3-0.6B.
- (ii) **Every number is real.** No number in these notes was typed by hand. Each one is produced by a script in the companion repository and cross-checked (Appendix B lists them). Where we make a claim about the world, such as “counting word sequences runs out of data”, we measure it on a real corpus.
- (iii) **Real questions.** Quotes marked S1 to S9 come from recordings of my live sessions on these topics, transcribed word for word, so they read like speech. They show where real learners got stuck.
- (iv) **Shapes on every equation.** Most confusion about transformers is shape confusion, not idea confusion. Every equation carries its tensor shape in grey on the right.

What you will be able to do. By the end, without looking anything up, you should be able to (a) run a full forward pass of a small transformer by hand, for a whole batch of sentences; (b) explain, as a picture of arrows turning and stretching, what training does to every weight; (c) read any model’s `config.json` and compute its parameter count, training cost and KV-cache memory; (d) design a transformer for a given compute, memory and latency budget, and defend every choice.

Contents.

0	Notation	7	From scores to words: temperature, sampling
1	The problem, and 70 years of attempts	8	Learning: how the arrows get nudged
2	Tokens, and words as points in space	9	Inference: the KV cache and its bill
3	Attention: letting tokens talk	10	The toy scaled up: GPT-2, then Qwen3
4	Order: from sinusoids to rotations (RoPE)	11	Designing your own transformer
5	Many heads, the MLP, the residual stream	A	The calculus, checked numerically
6	Normalisation, and why depth is hard	B	Reproducing every number

0 Notation

We fix these symbols once and never change their meaning. A vector is a row of numbers; a list of vectors is stacked, one per row, into a matrix. Shapes are written rows $\times$ columns. Come back to this page whenever a symbol looks unfamiliar; each one is also explained in words where it first appears.

Symbol	Meaning	Shape
B	number of sentences processed together (the <i>batch</i>)	scalar
T	number of tokens in each sentence	scalar
V	number of distinct tokens the model knows (the <i>vocabulary</i>)	scalar
d	model width: how many numbers describe one token	scalar ($d = 4$ in the toy)
d_k	width of one query, key or value vector	scalar ($d_k = 4$ in the toy)
h	number of attention heads in one layer	scalar
d_{ff}	width of the hidden layer inside the MLP	scalar
L	number of layers (blocks) stacked one after another	scalar
$\mathbf{x}_t$	the vector for token t (one row of X)	$1 \times d$
X	all token vectors of one sentence, stacked	$T \times d$
E	the embedding table: one row per token in the vocabulary	$V \times d$
W_Q, W_K, W_V	learned query, key, value matrices	$d \times d_k$
Q, K, V	queries, keys, values: XW_Q, XW_K, XW_V	$T \times d_k$
$\mathbf{a} \cdot \mathbf{b}$	dot product: $\sum_i a_i b_i$	scalar
$\ \mathbf{a}\ $	length of an arrow: $\sqrt{\mathbf{a} \cdot \mathbf{a}}$	scalar
$M^\top$	transpose: rows of M become columns	$(n \times m) \rightarrow (m \times n)$
S	match scores $QK^\top$	$T \times T$
A	attention weights (every row sums to 1)	$T \times T$
O	attention output AV	$T \times d_k$
θ	an angle; in Chapter 4, the angle by which RoPE turns a query or key at a given position	scalar
KV cache	the keys and values of earlier tokens, kept in memory during generation so they are not recomputed	$L \times 2 \times T \times d$ per sentence
$P(w \mid \text{context})$	probability of token w coming next, given what came before	scalar
N	number of learned parameters (numbers) in the model	scalar
D	number of training tokens the model reads	scalar
C	training compute, counted in floating-point operations ($C \approx 6ND$, Chapter 11)	scalar

When a whole batch is processed, every per-sentence shape gets a B in front: X becomes $B \times T \times d$, S becomes $B \times T \times T$, and so on. Nothing else changes.

Try it now. If a sentence has $T = 3$ tokens and $d_k = 4$, what shape is Q ? What shape is $S = QK^\top$? And with $B = 2$ such sentences, what shape is S ? (Answers: 3×4 ; 3×3 ; $2 \times 3 \times 3$.) Did any of these surprise you? If so, keep this page open while you read Chapter 3.

1 The problem, and seventy years of attempts

Map of this chapter. *The problem:* a customer writes **waste of time**, and we want a machine to write a good reply. *The steps:* three ways to automate the reply, and why writing it one word at a time wins (§1.1–1.2); then seventy years of attempts at the one function that needs, each with the failure that pushed the next one (§1.3–1.7); then a real case (§1.8) and the whole machine at a glance (§1.9). *Where it ends:* with four requirements that the transformer was built to meet, and a picture of how one pass handles a whole batch of sentences.

Before you start. Five quick questions. Answer each, then read the refresher under it.

1. A bag holds 3 red counters and 1 blue counter. You pick one without looking. What is the chance it is red?
Refresher: a probability is “favourable cases divided by all cases”, always between 0 and 1. Here $3/4 = 0.75$.
2. You toss a fair coin twice. What is the chance of two heads?
Refresher: for two separate events, “this and then that” means multiply: $\frac{1}{2} \times \frac{1}{2} = 0.25$.
3. What is $0.9 \times 0.9 \times 0.9$?
Refresher: multiplying a number by itself is written as a power: $0.9^3 = 0.729$. A number below 1 gets smaller every time you multiply by it.
4. A friend’s calculation says the chance of rain tomorrow is 1.2. What went wrong?
Refresher: nothing can be more certain than certain (1) or less than impossible (0). Any probability outside 0 to 1 means a mistake earlier in the working: a good thing to notice, because it tells you where to look.
5. What is 30% of 200?
Refresher: “per cent” means “out of 100”, so 30% of 200 is $0.30 \times 200 = 60$.

Before any architecture, we need a problem. Nobody wakes up wanting a transformer. We want something done, and the transformer is what fell out after many people tried many simpler things. So let us start where they started: with a job.

1.1 A real job

A software company sends every customer a message after a support chat: *How was your experience?* Thousands of answers come back every day. Most say *thanks*. Some say *great*. And some say

waste of time

What should the company say back? Tell me: if you were the support agent, what would you write?

A good agent writes something like *Sorry, this has not been useful*, and does not argue, does not paste a refund policy, and does not reply “Glad we could help!”. Our job: get a machine to write that reply.

How did you, the human, do it? You read the whole message. You noticed that **waste** turns **time** into a complaint. You wrote the reply one word after another, each word fitting the words before it. Keep those three things in mind: *read everything*, *notice which words change the meaning of which*, *write one word at a time*. The whole transformer is a machine for doing these three things.

In one sentence: Our running problem is to reply well to waste of time, which needs reading everything, noticing which words change which, and writing one word at a time.

1.2 Three ways to automate the reply

Attempt 1: rules. *If the message contains “waste of time”, reply with an apology.* Will this survive a week? On Monday it works. On Tuesday someone writes *total waste, useless, not worth my evening*, and the rule misses all three. On Wednesday someone writes *not a waste of time at all, loved it!* and the rule apologises to a happy customer. Rules break because language has endlessly many ways to say the same thing, and small words like *not* flip the meaning of everything around them.

Attempt 2: pick a reply from a list. Write ten good replies and train a model to choose one (“apology”, “thanks”, “escalate”, ...). This learns from examples instead of rules, which is better. But what happens with the eleventh situation, a customer who is angry *and* asks a question? There is no reply for it in the list. The model can only pick from what we imagined in advance.

Attempt 3: write the reply, one word at a time. Instead of choosing a finished reply, the machine builds one. And here is the trick: *writing a reply is guessing the next word, over and over.*

PICTURE IT. When a teacher says “Twinkle, twinkle, little —”, a room of children says *star*. They are not reciting a rule. They have heard so much language that, given the words so far, one word is far more likely than all others. Play that game again on *Twinkle, twinkle, little star, how I —*, and again, and again, and a whole verse comes out, one guess at a time.

WHERE THIS PICTURE BREAKS. The children answer with one word. The model answers with a probability for *every* word it knows, and something else (Chapter 7) picks one. The children also know what a star is; the model has only ever seen text.

SAY IT EXACTLY. Let a piece of text be a sequence of tokens $w_1, w_2, \dots, w_T$ (for now, a token is a word; Chapter 2 refines this). The *chain rule of probability*, which is “first this, then that given this”, says

$$P(w_1, w_2, \dots, w_T) = P(w_1) P(w_2 | w_1) P(w_3 | w_1, w_2) \cdots P(w_T | w_1, \dots, w_{T-1}) = \prod_{t=1}^T P(w_t | w_{<t}).$$

In words: the chance of the whole text is the chance of the first word, times the chance of the second word given the first, and so on. It is the coin question from *Before you start*, with each step allowed to depend on the earlier ones. So if a machine answers one question well (*given the words so far, how likely is each possible next word?*), it can write anything. Our reply

waste of time → Sorry → , → this → has → not → been → useful
given

is seven guesses, each made with everything to its left in view. Figure 1 shows the loop.

Predict first. Let’s assume each of the seven guesses is right with probability 0.9, and that one wrong word spoils the reply. Before computing: is the chance that the whole reply comes out right closer to 90%, 70% or 50%?

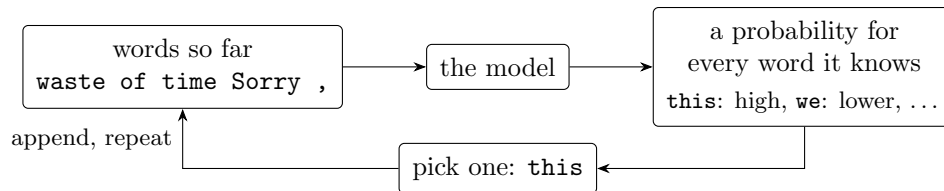


Figure 1: Writing as repeated guessing. Chapter 7 shows how the probabilities are computed. Everything in Chapters 2–6 lives inside the box called “the model”.

The chain rule says multiply: $0.9^7 = 0.478$, about 48%. For a 20-word reply it would be $0.9^{20} = 0.122$.

Sanity check. Every factor is below 1, so the product must fall as the reply grows, and it does.

WHY IT HAD TO BE THIS WAY. Two consequences fall straight out of this loop, and you should hold on to both.

- (i) *Every word is written in the light of all earlier words, including the model’s own.* If one guess goes wrong, every later guess builds on the mistake, and the worked example shows how fast that adds up. A large part of what people call *hallucination* comes from here.
- (ii) *The problem labels itself.* Any text is already a pile of solved exercises: hide the next word, and the text itself is the answer key. My eight lecture transcripts (183,152 words) are 183,152 such exercises, minus one, and nobody had to label a single one.

A tempting wrong idea (M1.3). *Hallucination is a bug someone forgot to fix.* It is tempting because software errors usually are bugs. But look at consequence (i): the same loop that lets the model write a fluent reply to a message it has never seen also lets one wrong guess carry the rest of the reply with it. As I put it in one session, “it’s not something of problem ... it’s a consequence of the process”. Systems built on these models reduce it with checks around the model; they do not delete a line of code.

A tempting wrong idea (M1.4). *Training data needs a human to label every example.* Every model you may have built before (a churn model, a fraud model, a release-triage model) needed a labelled X and Y , so this feels like a law. For the next-word game the label is the next word, already sitting in the text. Attempt 2 needed people to label replies; Attempt 3 needs only text.

A tempting wrong idea (M1.1). *Next-word prediction is shallow autocomplete, so it cannot involve understanding.* Phone keyboards also predict the next word, and they are not clever, so the comparison is natural. But ask what it takes to guess the next word *well* after **waste of time**: you need to know it is a complaint, who is speaking, and what a polite company says. The game is simple to state and hard to play; the better a model plays it, the more it has to capture about the world the text describes.

A tempting wrong idea (M1.2). *When a model says “I am 90% sure”, that is a calibrated probability.* It looks like one, and the model does compute probabilities inside. But those are probabilities of *next tokens*. The words “90% sure” are themselves next tokens, chosen because they fit the text, not a measurement of how often the model is right. Calibrated confidence has to be measured from outside, by checking many answers. Chapter 7 returns to this.

Check yourself 1.1. A model writes a 10-word reply, and each word is right with probability 0.95. Which statement is true?

- (a) The reply is right with probability 0.95, because each word is.
- (b) The reply is right with probability 0.95^{10} , which is less than 0.95.
- (c) If the reply is wrong, the cause is a bug in the code.
- (d) The model's reply "I am 95% sure" would tell us the same number.

Abstraction — turn the task into a game the data already plays. The strongest move in machine learning is often not a better model but a better question. "Choose a reply from my list" needs labelled data and caps the system at what you imagined. "Guess the next word" needs no labels and caps nothing: translation (*English: waste of time. French:*), classification (*Review: waste of time. Sentiment:*) and question answering all become "continue the text". When you design an AI system (an agent, a memory layer, an evaluator), ask first: what question can my data answer by itself?

So the problem is fixed for the rest of these notes:

Build a function that reads $w_1, \dots, w_{t-1}$ and returns $P(w_t = w \mid w_1, \dots, w_{t-1})$ for every word w in the vocabulary.

In one sentence: Writing is repeated next-word guessing, which turns every text into free, self-labelled practice, and also lets an early mistake carry forward.

1.3 Attempt: count what came next (1948 onwards)

PICTURE IT. Keep a giant notebook. Every time you read two words, write down the word that came after them. To guess what follows *waste of*, open the notebook at the page *waste of* and pick whatever was written there most often.

WHERE THIS PICTURE BREAKS. A real notebook has a page for every pair you might meet. This one only has pages for pairs it has already read, and, as we are about to measure, that is a tiny fraction of the pairs that exist.

SAY IT EXACTLY. This is the *n-gram model*. Claude Shannon introduced the idea in 1948, generating English-like text from counts of letter and word sequences, and in 1951 he measured how well *people* play the next-letter game. From the 1970s, speech-recognition groups, notably Frederick Jelinek's at IBM, built practical systems on word trigrams:

$$P(w_t \mid w_{t-2}, w_{t-1}) \approx \frac{\text{count}(w_{t-2}, w_{t-1}, w_t)}{\text{count}(w_{t-2}, w_{t-1})}.$$

In words: of all the times you saw the two previous words, in what fraction did this word come next?

Predict first. Take the transcripts of my first eight lecture sessions (183,152 words, 5,409 distinct words) as the notebook, and the ninth session (61,963 words) as new text. Before looking at the figure: what fraction of the three-word sequences in session 9 do you think never appear in sessions 1–8? A tenth? A quarter? Half? Write your guess down.

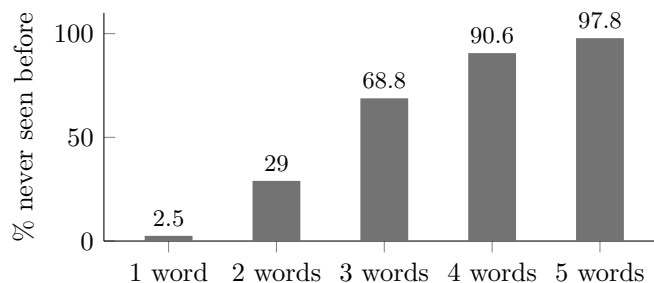


Figure 2: Sparsity, measured. Sequences in session 9 that never occur in sessions 1–8. Single words are almost always known; by three words, 68.8% are new.

Single words are almost always known (2.5% unseen). But 68.8% of three-word sequences, and 97.8% of five-word sequences, were never seen. And our running phrase? In the first eight sessions the page *waste of* is completely blank: it occurs 0 times. In session 9 the phrase **waste of time** occurs 38 times. A counting model built from 183,152 words has nothing to say about it.

Sanity check. Why so empty? With 5,409 words there are about 158 billion possible three-word sequences; the notebook has seen 131,165 of them, that is 0.00008% of the space. More text helps, but the space grows as V^n , much faster than any library.

WHY IT HAD TO BE THIS WAY. Clever patches (smoothing, such as Kneser and Ney, 1995) move a little probability onto unseen pages, but they cannot fix the real problem: *to a counter, every word is a name and nothing more*. Having seen *waste of money* a thousand times tells it nothing about *waste of time*, because it has no idea that *money* and *time* are alike.

A tempting wrong idea (M1.5). *A language model looks up sentences it has already seen.* It was trained on a huge amount of text, so storing and retrieving sounds plausible. The figure says otherwise: even inside one person’s lectures on one topic, most three-word sequences in a new session were never seen before. A model that only looked things up would be silent most of the time. Whatever a useful model does, it has to work on text it has never seen.

A tempting wrong idea (M1.8). *A counting model knows that time and money are alike.* You know it, and the model has seen both words many times, so it feels like it should. But the notebook stores counts of exact sequences. Nothing in it connects the page *waste of money* to the page *waste of time*. That connection is what §1.5 adds.

Check yourself 1.2. A three-word counting model is built from sessions 1–8 and is given **waste of**. What does it predict next?

- (a) **time**, because **waste of time** is a common English phrase.
- (b) Nothing: **waste of** never occurs in those sessions, so the count is zero.
- (c) **time**, because it has seen **waste of money** and knows *money* and *time* are alike.
- (d) **time**, because **waste of time** occurs 38 times in session 9.

Requirement 1. Words must have a notion of *similarity*, so that what is learned about one word transfers to similar words.

In one sentence: Counting what came next fails because almost every sequence is new, and a counter

cannot transfer what it knows to similar words.

1.4 Attempt: forget the order, count the words

A tempting shortcut, long used in search engines and spam filters: throw the words into a bag and only count them. A bag of *waste, of, time* is never sparse.

Predict first. Do these two customer replies give the same bag of words?

not useful, very good and very useful, not good

They contain exactly the same words the same number of times, so their bags are identical (our script checks this). They mean opposite things. Order is not decoration; it carries meaning.

A tempting wrong idea (M1.6). *If the right words are present, their order hardly matters.* Keyword search works well in daily life, so this is easy to believe. The pair above has the right words and the opposite meaning. Order matters most for exactly the small words, like *not*, that the rules of §1.2 tripped on.

Check yourself 1.3. Which pair of replies would a bag-of-words model treat as identical?

- (a) good, not cheap and cheap, not good
- (b) waste of time and waste of money
- (c) not useful and useless
- (d) None: a bag of words keeps the order of the first and last word.

Requirement 2. The model must know *which word came where*.

In one sentence: Counting words without their order removes sparsity but also removes meaning, so order must be kept.

1.5 Attempt: words as points in space (2003, 2013)

PICTURE IT. Imagine a map of a city where every word lives at an address. *Time* lives next door to *money* and *effort* (things you can waste); *star* lives across town. If you learn something about one house, you have learned a little about its neighbours.

WHERE THIS PICTURE BREAKS. A city map has two directions. Real word maps have hundreds, so a word can be near *money* in one direction (things you spend) and near *clock* in another (things you measure). No flat map can show that.

SAY IT EXACTLY. In the counting model, each word is effectively a *one-hot* vector: a list of V numbers, all zero except a single 1 at the word's own position. Any two different one-hot vectors are exactly the same distance apart ($\sqrt{2} \approx 1.414$), so every word is an equal stranger to every other. Bengio and colleagues (2003) replaced this with a short learned vector of d numbers per word, an *embedding*, and trained a small neural network to predict the next word from the embeddings of the previous few. Because similar words end up with similar vectors, knowledge spreads between them. Mikolov and colleagues' *word2vec* (2013) made such vectors cheap to learn from billions of words.

WHY IT HAD TO BE THIS WAY. This meets Requirement 1, and it gives us the most important idea in these notes: *a word is an arrow in space, and meaning is direction*. Chapter 2 builds it properly; Chapter 3 uses it on every page. But Bengio’s model still looked at a fixed window of the last few words. What about the word fifty words back?

In one sentence: Giving each word a learned position in space lets knowledge spread between similar words, but a fixed window still cannot reach far back.

1.6 Attempt: read word by word, keep a running summary (1990 to 2014)

PICTURE IT. Read a story one word at a time, and after each word update a single sticky note that summarises everything so far. To guess the next word, look only at the sticky note.

WHERE THIS PICTURE BREAKS. A person can reread the story; this reader cannot. Once a detail is rubbed off the note, it is gone.

SAY IT EXACTLY. This is the *recurrent neural network* (RNN; Elman, 1990; used as a language model by Mikolov and colleagues, 2010). It keeps a hidden state $\mathbf{h}_t$ and updates it with each new word:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t), \quad P(w_{t+1} \mid w_{\leq t}) = g(\mathbf{h}_t).$$

It meets Requirement 2 for free, since the note is written in reading order, and in principle it can remember forever. In practice it breaks in three ways.

Break (i): one sticky note. Whatever the sentence (three words or three thousand), it must fit into the same fixed-size $\mathbf{h}_t$. Detail is squeezed out.

Break (ii): memories fade or explode. Predict first. Take a single number carried on the note, and let’s assume each update multiplies it by 0.9. How much of it is left after 50 words? More or less than a quarter?

After 10 words 0.349 of it is left, after 50 words 0.00515, after 100 words 2.66×10^{-5} . If the factor is 1.1, it grows to 2.59, then 117, then 1.38×10^4 . Real networks multiply by matrices, not numbers, but the effect is the same, and it hits learning as hard as remembering: the signal that should teach the network “the word fifty steps back mattered” has to travel back through the same fifty multiplications (Bengio, Simard and Frasconi, 1994). The LSTM (Hochreiter and Schmidhuber, 1997) added *gates*, learned switches that let the note keep some numbers untouched, which eased this a lot but did not remove breaks (i) or (iii).

Break (iii): it cannot be done in parallel. You cannot write note 1000 before note 999, and you cannot write 999 before 998. A sentence of 1,000 tokens is 1,000 steps that must happen one after another. A GPU is a machine with thousands of small workers; give it a chain like this and almost all of them wait.

Requirement 3. Any word must be able to reach any earlier word *directly*, not through a chain of summaries.

Requirement 4. The work for all positions must be doable *at the same time*.

Check yourself 1.4. An RNN carries a signal that is multiplied by 0.9 at every word. Which statement is true?

- (a) After 50 words, the signal is about half its size.
- (b) After 50 words, less than 1% of the signal is left, and the lesson “this early word mattered” fades the same way.
- (c) An LSTM’s gates make the RNN process all words at the same time.
- (d) The fading only affects remembering, not learning.

In one sentence: An RNN reads in order, but squeezes everything into one note, lets old signals fade, and cannot work on all positions at once.

1.7 The turn: let the reader look back (2014 to 2017)

In 2014, neural translation systems (Sutskever, Vinyals and Le; Cho and colleagues) read an English sentence with one RNN into a single vector and wrote the French with another. Break (i) was plain to see: long sentences came out worse. Bahdanau, Cho and Bengio (2014) proposed a fix: when writing each French word, let the writer *look back at every English word* and decide how much each one matters right now. They called these weights *attention*. Requirement 3 was met. But there was still an RNN underneath, so Requirement 4 was not.

In 2017, Vaswani and colleagues asked: *what if attention is all you need?* Drop the RNN entirely. Let every word look at every earlier word directly, and do it for all words at once. Figure 3 shows the difference. Their base translation model trained in about twelve hours on eight GPUs. That architecture is the *transformer*; GPT (2018), GPT-2 (2019) and every model in Chapter 10 are transformers trained on the next-word game of §1.2.

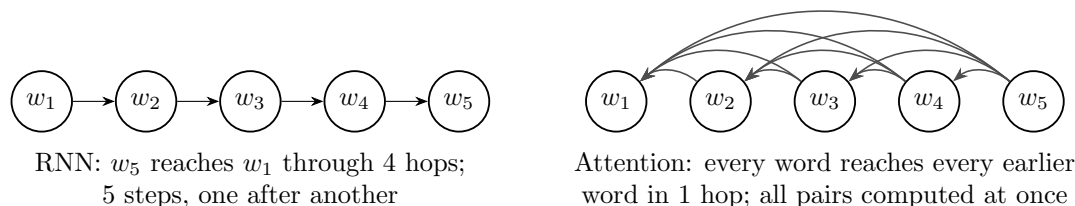


Figure 3: The two shapes of computation. For a sentence of 1,000 tokens, the RNN needs 1,000 dependent steps and 999 hops from last to first; attention computes all 1,000,000 pairings in one parallel step and connects any two words in one hop.

A tempting wrong idea (M1.7). *Transformers replaced RNNs mainly because they are more accurate.* Better results are the headline of every paper, so this is the natural reading. Look at the two breaks the transformer removed: the long chain between distant words (Requirement 3) and the one-step-at-a-time work (Requirement 4). The second one decided the history: a model that uses every GPU worker at once can be trained on far more text in the same time, and the accuracy followed from the scale.

Attempt	What it fixed	Where it broke
Rules	nothing to learn	endless paraphrases, negation
n -gram counting (1948 on)	learns from text, no labels	68.8% of 3-word sequences unseen
Bag of words	no sparsity	order lost: opposite meanings, same bag
Embeddings (2003, 2013)	similarity between words	fixed window of recent words
RNN, LSTM (1990, 1997)	order; unlimited history in principle	one note; fading; strictly sequential
RNN + attention (2014)	direct look-back	still sequential underneath
Transformer (2017)	direct look-back, all at once	<i>order is lost again</i> : see §3.11

In one sentence: Attention gave every word a direct line to every earlier word, and dropping the RNN let all of that run at once, which is what made training at scale possible.

1.8 A real case: Google Translate, 2016

For a decade, Google Translate worked phrase by phrase: cut the sentence into short chunks, look each chunk up in a huge table of translations counted from text, then glue and reorder the pieces. That is the counting notebook of §1.3, applied to translation, and it had the same weakness: anything outside a chunk was invisible, so whole sentences came out stiff. In 2016 Google replaced it with a neural system (Wu and colleagues, 2016): LSTMs that read the whole sentence, with attention letting the writer look back at every source word. That is §1.6 plus §1.7, about a year before the transformer. A year later the transformer paper reported better translation quality than such systems at a fraction of the training cost.

Tell me: which of the four requirements does each step in this story meet? The phrase table meets none of 1, 3 and 4. The 2016 system meets 1, 2 and 3. The transformer meets 1, 3 and 4, and loses 2 for a while (Chapter 4).

In one sentence: A product used by millions went through the same sequence of fixes as these notes, for the same reasons.

1.9 The whole machine at a glance

Before we open the box, here is the whole forward pass, the route every batch of text takes from characters to probabilities. Do not try to understand each stage yet; look at the shape of the road.

PICTURE IT. An assembly line. Sentences come in at one end as text. They are cut into pieces (tokens), each piece gets its starting arrow (embedding), and then the arrows pass through a row of identical workstations. At each station two things happen: the tokens *talk to each other* (attention), then each token *thinks on its own* about what it heard (the MLP). At the end of the line, every token's arrow is turned into a guess about the word that comes after it.

WHERE THIS PICTURE BREAKS. On an assembly line each item passes each station once and the stations never look back. Here every station sees all the tokens of a sentence together, and what one token becomes depends on the others.

Drawing the transformer, block by block. Here is the whole machine as a frame of empty blocks. Every chapter from now on fills in one or two of them, and opens with this same picture, so you always know where you are: solid blocks are built, the highlighted block is the one that chapter builds, and dashed blocks are still ahead. For now only the top is filled in: text comes in.

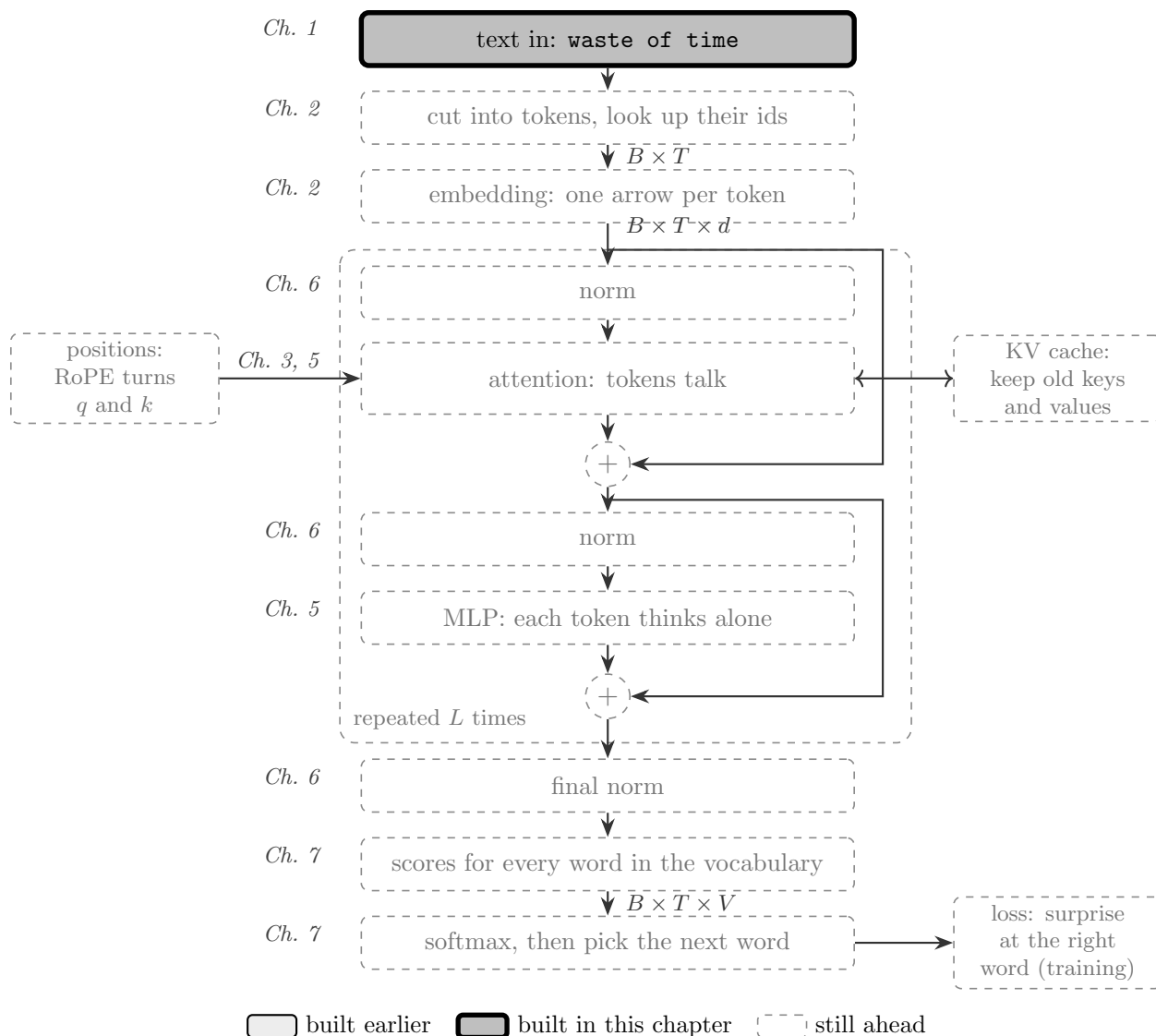


Figure 4: Where we are in the machine. The frame we will fill. Text enters at the top; each dashed block is one chapter’s job. The middle part, inside the large rounded box, is one *layer*; a real model stacks L of them. At the bottom, the model gives a score to every word it knows, of shape $B \times T \times V$: a guess at *every* position of *every* sentence in one pass, not only at the last one.

	what the model reads				the word it must guess next		
	1	2	3		1	2	3
sentence 1	waste	of	time	sentence 1	of	time	Sorry
sentence 2	not	very	useful	sentence 2	very	useful	.
	$B \times T = 2 \times 3$				$B \times T = 2 \times 3$ guesses		

Look at the two tables above. That is the quiet reason transformers train so fast. A sentence of T tokens is not one exercise but T exercises: after **waste**, guess **of**; after **waste of**, guess **time**; after **waste of time**, guess **Sorry**. The transformer answers all of them in the same pass, for all B sentences in the batch. How can it do that without cheating, without the guess for position 2

peeking at the answer in position 3? That is one of the things Chapter 3 has to explain.

Check yourself 1.5. A batch holds $B = 8$ sentences of $T = 16$ tokens each. How many next-word guesses does one forward pass produce?

- (a) 8, one per sentence.
- (b) 16, one per position.
- (c) 128, one per position of every sentence.
- (d) It depends on how long the reply is.

In one sentence: One pass turns B sentences of T tokens into $B \times T$ next-word guesses, through tokens, embeddings, L talk-then-think stations and a final scoring step.

Looking back. These look back at Chapter 0, the notation page. (1) What do B , T and d stand for, in words? (2) A matrix has shape 3×4 . How many rows does it have, and what does each row hold in these notes? (3) Which symbol will stand for the number of learned parameters?

Exercises

Exercise 1.1. A notebook model looks at the previous 4 words, and the vocabulary has 50,000 words. How many pages could the notebook have? Is that more or less than the number of words ever written?

Exercise 1.2. Using the bag-of-words idea, find two more replies a support team might receive that have identical bags but opposite meanings.

Exercise 1.3. An RNN multiplies a signal by 0.95 per word. After how many words is less than 1% of it left?

Exercise 1.4. Write the chain rule for the reply **Sorry , this has not been useful given waste of time**: how many factors are there, and what does the third factor condition on?

Answers to Chapter 1

Check yourself 1.1. (b). A chain of guesses multiplies, so the whole is less likely than each part: $0.95^{10} \approx 0.60$. (a) treats the whole reply as one guess. (c) is the wrong idea M1.3: wrong words come from the guessing process itself. (d) is M1.2: “I am 95% sure” is more text, not a measurement.

Check yourself 1.2. (b). The page *waste of* is blank: 0 occurrences. (a) is M1.5: the model does not look things up in “English”; it only has its notebook. (c) is M1.8: counts do not know which words are alike. (d) uses data the model never trained on, the same leakage your fraud model must avoid.

Check yourself 1.3. (a). Same words, same counts, opposite meanings. (b) differ in one word. (c) have different words. (d) is M1.6 in disguise: a bag keeps no order at all.

Check yourself 1.4. (b). $0.9^{50} = 0.00515$. (a) underestimates how fast repeated multiplication shrinks a number. (c) is wrong: gates help memory, not parallelism (M1.7 is the belief that accuracy, not parallelism, was the deciding issue). (d) misses that learning signals travel back through the same steps.

Check yourself 1.5. (c). $8 \times 16 = 128$: every position of every sentence predicts its next token. (a) and (b) count one dimension only. (d) confuses training on given text with writing a new reply.

Looking back. (1) Batch size, tokens per sentence, numbers per token. (2) Three rows, one per token. (3) N .

Exercise 1.1. $50,000^4 = 6.25 \times 10^{18}$ possible four-word contexts; far more than all words ever written, so almost every page stays empty.

Exercise 1.2. For example `good, not cheap` (good but expensive) versus `cheap, not good`.

Exercise 1.3. We need $0.95^n < 0.01$, that is $n > \ln 0.01 / \ln 0.95 \approx 89.8$, so after 90 words.

Exercise 1.4. Seven factors, one per reply token. The third is $P(\text{this} \mid \text{waste of time Sorry ,})$: it conditions on the message and the two reply tokens before it.

What you now know

- Why writing a reply one word at a time beats rules and fixed reply lists, and why that game labels itself.
- How the chain rule turns a sentence into a product of next-word probabilities, and why errors compound.
- Why counting, bag of words, word vectors, RNNs and RNN-plus-attention each fell short, measured where possible, and the four requirements those failures left behind.
- The shape of the whole forward pass, and why one pass gives $B \times T$ guesses.

2 Tokens, and words as points in space

Map of this chapter. A model cannot read. It receives numbers. This chapter answers two questions: how do we cut text into pieces the model can count (*tokens*), and how does each piece become an arrow of d numbers (an *embedding*)? We try whole words, then single letters, watch both break on real text, and arrive at byte-pair encoding. Then we turn each token into an arrow, first by plain lookup and then by asking where good arrows come from. It ends with Requirement 1 of Chapter 1 met: similar words sit near each other, and Chapter 3 can start moving them.

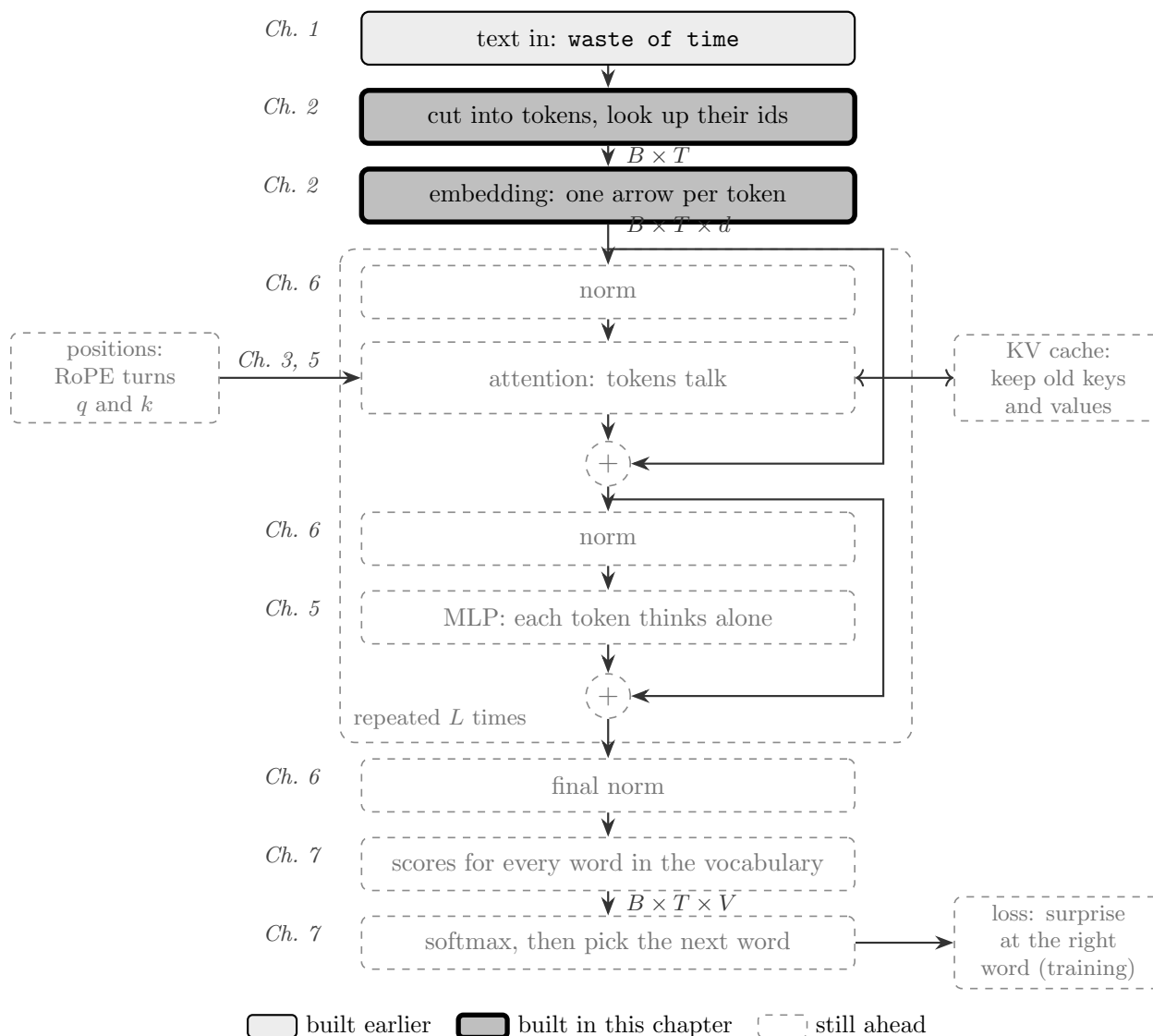


Figure 5: Where we are in the machine. You are here: Chapter 2 builds the first two blocks, cutting text into tokens and looking up one arrow per token. Every later block works on these arrows.

Before you start. (1) In a list of 1000 words, 25 have never been seen before. What percentage is that? (2) Count the pairs of letters in the word **banana**, reading left to right. How many times does **an** appear? (3) A table has 12 rows and 4 columns. How many numbers does it hold? (4) If two arrows

point at right angles, what is their dot product?

Refreshers. (1) $25/1000 = 0.025$, which is 2.5%. (2) The pairs are **ba**, **an**, **na**, **an**, **na**: **an** appears twice. Overlapping pairs count separately. (3) $12 \times 4 = 48$. (4) Zero (Chapter 3, §3.2): no shadow, nothing in common.

2.1 What does the model receive?

Predict first. When you type **waste of time** into a chat model, what arrives at the first layer of the model: letters, words, or something else? Write your guess down before reading on.

The answer is a short list of whole numbers, one per piece of text, such as (2) for the toy word **time**. Everything the model will ever know about your message has to travel through those numbers. So the first design decision in any language model is a quiet one: *what counts as one piece?* Let us try the two obvious answers and watch them break, with numbers from my own lecture transcripts, the same corpus as Chapter 1.

In one sentence: the model never sees text, only a list of token numbers, so we must decide what one token is.

2.2 Two simple choices, and how each breaks

Choice 1: one token per word. This is how you read, so it feels natural. In sessions 1–8 there are 183,152 words but only 5,409 different ones, so the table of words is small.

How it breaks. Language keeps inventing words. In session 9, 2.5% of the words (789 different ones) never appeared in sessions 1–8, and a word-level model has no row for any of them. What would you do with a word you have no row for? In the session where we built this, the answers in the room were all reasonable guesses:

| S9: “it will try to go to the nearest token or maybe it will split further or maybe it will write unknown.”

Older systems did the last one: every new word became one special token, **<unk>**, and all of them looked the same to the model. A customer who writes a product name the model has never seen would be answered as if they had written nothing. And the long tail is not small: 36% of the different words in sessions 1–8 (1,934 of them) appear exactly once.

Choice 2: one token per letter. There are only 27 different letters in the transcripts, so nothing is ever unknown.

How it breaks. The text gets long. An average word here is 4.14 letters, plus a marker for the end of the word, about 5.12 tokens per word. Chapter 3 compares every token with every earlier token, so the work grows with the *square* of the length: roughly $5.12^2 \approx 26$ times more comparisons than one token per word. And a single letter carries almost no meaning, so the model must spend layers re-assembling words before it can think about them.

Check yourself 2.1. You are building a model for a support team whose customers often type new product names. Which choice breaks for them first, and why?

- (a) One token per word, because every new product name has no row in the table.
- (b) One token per letter, because product names are long.
- (c) Neither, because the model reads the name the way a person does.
- (d) One token per word, because product names are longer than other words.

In one sentence: whole words leave new words with nowhere to go; single letters make every sentence about five times longer and attention about 26 times more expensive.

2.3 The middle path: byte-pair encoding

Predict first. Suppose you could give a few hundred common letter-groups their own token, and spell everything else out of letters. Which groups would you pick first?

PICTURE IT. Think about how people take fast notes. Nobody invents a shorthand for a word used once, but a phrase you write twenty times in a meeting soon gets an abbreviation. Byte-pair encoding does exactly that, automatically: it looks for the pair of symbols that sits side by side most often in the text, and gives that pair its own symbol. Then it does it again, and again.

The numbers. We ran it from scratch on sessions 1–8 (the script is `code/ch2_bpe.py`). Each word starts as its letters followed by an end-of-word marker `_`, so the word `time` starts as `t i m e _`. The first merges, with how often each pair appeared:

merge	pair	times seen	merge	pair	times seen
1	e+ <code>_</code>	35,241	5	i+n	13,221
2	t+ <code>_</code>	24,560	6	y+ <code>_</code>	11,498
3	t+h	21,348	7	o+ <code>_</code>	11,313
4	s+ <code>_</code>	21,177	8	a+n	11,111

Did your prediction include `e_`, a word ending in “e”? Very few people guess it, and it is the most common pair of all. After 1,000 merges the vocabulary has 1,028 symbols, and our running words split like this:

word	0 merges	100 merges	1,000 merges
waste	w a s t e <code>_</code>	w as t e <code>_</code>	w as te <code>_</code>
of	o f <code>_</code>	of <code>_</code>	of <code>_</code>
time	t i m e <code>_</code>	ti me <code>_</code>	time <code>_</code>

Common words become one token. Rarer words become a few familiar pieces. And now the real test: the word `tokenizer` appears 24 times in session 9 and never in sessions 1–8. A word-level model would have nothing. Byte-pair encoding writes it as `t ok en iz er _`. Of the 61,963 words in session 9, the number that cannot be written at all is 0, and the average word costs 1.37 tokens, against 5.12 for single letters.

SAY IT EXACTLY. Byte-pair encoding (Gage, 1994, as a compression method; Sennrich, Haddow and Birch, 2016, for translation) builds the vocabulary in a loop:

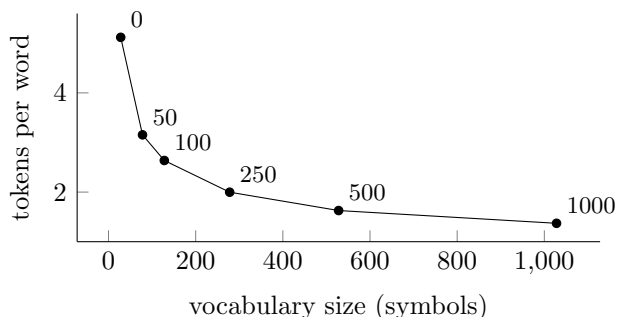


Figure 6: Measured on session 9: the more merges (numbers on the points), the fewer tokens each word costs. The curve flattens, so the vocabulary size is a trade-off between table size and sequence length.

1. Start with every word written as single symbols plus an end marker. The vocabulary is the set of single symbols.
2. Count every pair of neighbouring symbols across the whole text, weighted by how often each word appears.
3. Add the most frequent pair to the vocabulary as one new symbol, and rewrite the text with it.
4. Repeat until the vocabulary has the size you chose.

To cut new text into tokens, apply the learned merges in the order they were learned. GPT-2 (Radford and colleagues, 2019) starts from the 256 possible *bytes* instead of letters, so any text in any language, including emoji and code, can always be written. Real tokenizers are this same loop run on far more text: Qwen3 uses a vocabulary of 151,936 entries, as published in the Qwen3 technical report (2025; we will check this against the model’s `config.json` in Chapter 10). In session 9 we quoted the tokenizer’s own count, 151,669; the embedding table is padded to a slightly larger size.

WHERE THIS PICTURE BREAKS. Shorthand notes are written by one person for a reason they understand. Byte-pair encoding has no idea what a pair means; it only counts. So `e_` becomes a token because it is frequent, not because it is a unit of meaning. The model has to learn meaning later.

WHY IT HAD TO BE THIS WAY. Two pressures pull in opposite directions: a small vocabulary keeps the table small and leaves no word unknown; a large vocabulary keeps sequences short. Frequency is a fair referee between them, because the things you write often are the things worth a dedicated symbol.

A tempting wrong idea (M2.2). “A word the model has never seen makes it fail or print *unknown*.” With byte-pair encoding there is always a way to spell the word, down to single letters or bytes. The model never fails to *read* a new word. Whether it *understands* it is a separate question: `tokenizer` arrives as pieces, and the model has to build the meaning from `ok`, `en`, `iz`, `er_` and the words around them.

A tempting wrong idea (M2.3). “A token is a word.” In session 9 we described the tokenizer as “tokenizing it into the words”, and for `of` and `time` that is true. But `waste` became three tokens in our tokenizer. Measured on session 9, one of our tokens is on average 0.73 of a word, and a rare word is several tokens.

Check yourself 2.2. After 1,000 merges, **time** is one token but **waste** is three. What best explains this?

- (a) **time** is more important for meaning than **waste**.
- (b) In sessions 1–8, the pieces of **time** were merged often enough to become one symbol, and **waste** was rarer.
- (c) The tokenizer splits every word into its syllables.
- (d) **waste** has more letters, and long words are always split.

In one sentence: byte-pair encoding gives frequent letter-groups their own token and spells everything else from smaller pieces, so nothing is ever unknown and common words stay short.

2.4 From a token number to an arrow: the embedding table

Predict first. The token **time** is number 2 in our toy vocabulary and **of** is number 1. Does that mean **time** and **of** are similar, because 1 and 2 are close?

PICTURE IT. A token number is a locker number. Locker 2 is next to locker 1, but that tells you nothing about what is inside. To get the contents you walk to the locker and open it. The embedding table is the row of lockers, and each locker holds an arrow of d numbers.

The numbers. Our toy model knows 12 tokens, each with an arrow of $d = 4$ numbers. First, the text becomes token numbers, one per token; a batch of sentences becomes a table of numbers:

token ids of waste of time				a batch of token ids			
waste	of	time		sentence 1			
0	1	2		0	1	2	
				3	4	5	
$1 \times T = 1 \times 3$				$B \times T = 2 \times 3$			

Each number opens one row of the embedding table E . Rows 0–2 are the arrows Chapter 3 uses; the highlighted row is **time**:

	E			
	1	2	3	4
waste	1	0	1	0
of	0	1	0	0
time	0	0	1	1
not	0	1	0	1
very	1	0	0	0
useful	0	0	1	0
Sorry	1	1	0	0
,	0	0	0	1
this	1	0	0	1
has	0	1	1	0
been	1	1	1	0
.	0	1	1	1

$V \times d = 12 \times 4$

To fetch **time** with a multiplication, write its number as a *one-hot* row: twelve zeros with a single 1 in position 2 (counting from 0). Multiplying that row by E picks out exactly one row, because every other row is multiplied by 0:

	one-hot(time)		E		$\mathbf{x}_{\text{time}}$
	0 1 2 3 4 5 6 7 8 9 10 11		1 2 3 4		1 2 3 4
time	0 0 1 0 0 0 0 0 0 0 0 0	$\times$	1 0 1 0 0 1 0 0 0 0 1 1 0 1 0 1 1 0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1	$=$	0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1×12		12×4		1×4

Figure 7: The lookup as a matrix product, $(1 \times V)(V \times d) = 1 \times d$. The one-hot row meets column 4 of E : only the term from row 2 survives. The same happens for every column, so the result is row 2 of E , the arrow of **time**. Our script checks this for every token.

SAY IT EXACTLY. For a vocabulary of V tokens and width d , the embedding table is $E \in \mathbb{R}^{V \times d}$ and the arrow for token i is $\mathbf{x} = \mathbf{e}_i E$, where $\mathbf{e}_i$ is the one-hot row for i . In code nobody multiplies

by a row of zeros; it is written $E[i]$. The two are the same thing, and the one-hot form is what lets training treat the lookup as an ordinary matrix product. The toy table holds $12 \times 4 = 48$ numbers. Qwen3’s, with the published $V = 151,936$ and $d = 1024$, holds 155,582,464 numbers, about 155.6 million.

For a batch, the $B \times T$ table of token numbers becomes a $B \times T \times d$ block of arrows: every number is replaced by its row of E . Two sentences, three tokens each, four numbers per token:

		0	1	0	1	← sentence 2
waste	1	0	1	0	0	
of	0	1	0	0	0	
time	0	0	1	1		← sentence 1

$X: B \times T \times d = 2 \times 3 \times 4$

A tempting wrong idea (M2.4). “Token 1 and token 2 are close, so their meanings are close.” The number only says which row to open. In our table, row 1 (**of**) and row 2 (**time**) are at right angles: their dot product is 0. Meaning lives in the arrows, never in the locker numbers.

A tempting wrong idea (M2.5). “The vector of a word changes with the sentence.” In session 9 we stopped on exactly this point: “embeddings are same, assume that embeddings are same.” The row for **time** is the same in *time flies* and in *waste of time*. What changes with context is the arrow *after* attention has mixed in the other tokens (Chapter 3). The table gives every token the same starting point; the layers do the rest.

Check yourself 2.3. What does multiplying the one-hot row for **useful** (a 1 in position 5) by E give?

- (a) $(0, 0, 1, 0)$, row 5 of the table.
- (b) $5 \times (1, 1, 1, 1)$, because the 1 sits in position 5.
- (c) The sum of all twelve rows, since every row is multiplied.
- (d) A number between 0 and 1 saying how useful the word is.

In one sentence: a token number opens one row of the embedding table, and that row, an arrow of d numbers, is where the model’s knowledge about the token starts.

2.5 Where do good arrows come from?

Our toy arrows were chosen by hand. Requirement 1 of Chapter 1 asks for more: *similar words must get similar arrows*, so that what is learned about one transfers to the other. How could a machine know that **money** and **amount** are alike, if nobody tells it?

PICTURE IT. You learn what kind of person someone is by talking with them and seeing who they spend time with. A word is the same. The linguist J. R. Firth put it in 1957: “You shall know a word by the company it keeps.” Words that keep the same company are used for the same things.

The numbers. Let us build arrows from company alone. We took the 3,000 most frequent words of sessions 1–8 and counted, for each pair of words, how often they appear within 3 words of each other. Then we kept only the pairs that occur together more often than chance would predict, and compressed each word’s row of counts to 64 numbers (the method is in `code/ch2_bpe.py`; Levy and Goldberg, 2014, showed it is closely related to the popular *word2vec* of Mikolov and colleagues, 2013). Nobody told the program anything about meaning. Here are the nearest arrows it found, with their cosine (the agreement $\cos \theta$ from Chapter 3):

word	nearest words in sessions 1–8 (cosine)
money	amount (0.58), costs (0.50), businesses (0.46), spending (0.46)
model	tuned (0.71), models (0.68), tune (0.67), quen (0.65)
question	fundamental (0.69), another (0.69), second (0.67), ask (0.64)
time	solves (0.61), same (0.59), at (0.59), point (0.55)

money lands next to **amount** and **costs**; **model** next to **models**, **tuned** and **quen** (how the transcript spells Qwen). Now look at **time**. Its neighbours are **same**, **at**, **point**: in a lecture, “time” mostly lives in *at the same time* and *at this point in time*. The arrows learn how a word is *used in this text*, not what a dictionary says. The cosine between **time** and **money** is 0.28, while any two different one-hot rows have cosine 0.00: the counting notebook of Chapter 1 had no way to see any resemblance at all.

SAY IT EXACTLY. An embedding is a map from tokens to $\mathbb{R}^d$ chosen so that the geometry carries information: tokens used in similar contexts get a small angle between them. In this section the map came from counting. In a transformer the table E is learned by the next-token game itself (Chapter 8): if two tokens tend to be followed by the same words, the nudges that improve one also improve the other, and their rows drift together.

WHERE THIS PICTURE BREAKS. The “company” picture says nothing about *which* company. **good** and **bad** keep almost the same company (“a _ answer”, “very _”), so counting puts them close together even though they are opposites. A model that needs to tell them apart has to learn it from the next-word game, where they *are* followed by different words.

A tempting wrong idea (M2.6). “Each of the 64 numbers means something you can name, like sentiment.” In the churn model of session 3, each clue had a name: number of logins, days since last purchase. Embedding coordinates do not. The compression is free to spread one idea over many coordinates and to pack many ideas into one. Directions can carry meaning; single coordinates rarely do.

Check yourself 2.4. Our arrows put **time** near **same** and **point**. What does that tell you?

- (a) The method failed, because **time** should be next to **day**.
- (b) In these transcripts, **time** is mostly used in phrases like *at the same time*, and the arrows record usage.
- (c) Someone labelled **time** and **same** as synonyms.
- (d) The fourth coordinate of each arrow stores the word’s meaning, and they share it.

In one sentence: arrows built only from which words appear together already place similar words close together, and a transformer learns its own arrows the same way, from the next-word game.

2.6 Requirement 1 is met. What is still missing?

We now have what Chapter 1 asked for: a way to read any text (byte-pair encoding) and a starting arrow for every piece, with similar pieces near each other. But look again at §2.4: the row for **time** is the same in every sentence. A support reply to *waste of time* and a calendar reminder about *on time* start from the same arrow. The arrow has to change once the token has seen its neighbours. That is exactly the job of attention, and it is where Chapter 3 begins.

Abstraction — give frequent things a short name and rare things a way to be spelled. Byte-pair encoding is a compression rule: short codes for what is common, a guaranteed fallback for everything else. The same rule appears in good systems everywhere: a cache keeps hot items close and goes to disk for the rest; an agent’s tool list gives common actions a named tool and leaves a general fallback (“run this command”) for the rest. And the embedding idea, *put similar things close together and search by angle*, is the whole basis of retrieval in an agent’s memory system: a memory is stored as an arrow, and recall means finding the stored arrows with the smallest angle to the question.

Looking back. (a) Chapter 1 measured that 68.8% of three-word sequences in session 9 were never seen before. Does byte-pair encoding fix that problem? (b) Which of the four requirements from Chapter 1 does this chapter meet? (c) In Chapter 3, which matrix was the first to act on the arrows that come out of the embedding table?

Exercises

Exercise 2.1. Run one step of byte-pair encoding by hand on the text **waste waste wasted time** (each word followed by `_`). Which pair, or pairs, are most frequent, and how many times does each appear? What must a real tokenizer do when there is a tie?

Exercise 2.2. Compute the cosine between the toy arrows of **waste** and **time**. Then compute it between the one-hot rows of the same two tokens. Which one could ever tell you the words are related?

Exercise 2.3. A model has $V = 50,000$ tokens and $d = 768$. How many numbers are in its embedding table? If you double the vocabulary, what happens to the table, and what (from §2.2) do you expect to happen to the length of a typical sentence?

Exercise 2.4. Explain to a friend who has never programmed why a model with a byte-level vocabulary can read a word it has never seen, but may still not understand it. Use the locker picture and say where it stops working.

Answers to Chapter 2

Check yourself 2.1. (a). Each new product name has no row. (b) is wrong: letters never run out; long names cost length, not failure. (c) reflects M2.1: the model does not read like a person, it only gets token numbers. (d) confuses length with novelty.

Check yourself 2.2. (b). Merges follow frequency in the training text. (a) reflects a belief that the tokenizer judges meaning; it only counts. (c): byte-pair pieces need not be syllables (**as**, **te_**).

(d): length matters only through frequency; a long common word can be one token.

Check yourself 2.3. (a). The one-hot row selects row 5. (b) treats the token number as a quantity (M2.4). (c) forgets that eleven of the twelve multipliers are zero. (d) treats the arrow's numbers as nameable scores (M2.6).

Check yourself 2.4. (b). The arrows record how a word is used here. (a) expects dictionary meaning; (c) is M2.7, since nobody labelled anything; (d) is M2.6.

Looking back. (a) No. Byte-pair encoding makes every *word* readable, but a counting model still needs to have seen the same *sequence*; that problem is solved by arrows plus attention. (b) Requirement 1, similarity. (c) W_Q , W_K and W_V act on the arrows.

Exercise 2.1. 5 pairs tie, each seen 3 times: **as**, **e_**, **st**, **te**, **wa**. A tokenizer needs a fixed tie-breaking rule (for example, alphabetical order) so that the same text always gives the same tokens.

Exercise 2.2. The toy arrows $(1, 0, 1, 0)$ and $(0, 0, 1, 1)$ have dot product 1 and lengths $\sqrt{2}$ each, so the cosine is $1/2 = 0.50$. The one-hot rows have cosine 0.00, as any two different one-hot rows do. Only the arrows can express a relation.

Exercise 2.3. $50,000 \times 768 = 38,400,000$ numbers. Doubling the vocabulary doubles the table. Sentences get shorter, since more letter-groups have their own token, but by less each time: the curve in §2.3 flattens.

Exercise 2.4. Any word can be written as bytes, so it always gets a list of lockers to open. But each locker holds the arrow of a *piece*, and the meaning of the whole word has to be assembled by later layers from the pieces and the context. The locker picture stops working there: lockers are independent, while a word's meaning depends on its neighbours.

What you now know.

- Why neither whole words nor single letters work, with the numbers that show it.
- How byte-pair encoding builds a vocabulary by merging the most frequent pair, again and again.
- That an embedding is one row of a table, fetched by a one-hot row times a matrix.
- That token numbers carry no meaning, and that arrows built from word company place similar words close together.

3 Attention: letting tokens talk

Map of this chapter. *The problem:* the token **time** must learn, from the tokens before it, that it is time *wasted*. *The steps:* the plain average and why it fails (§3.1); three tools from geometry: the dot product, the matrix, the transpose (§3.2–3.4); queries, keys and values (§3.5); a full worked example (§3.6); why softmax and why $\sqrt{d_k}$ (§3.7–3.8); a whole batch at once (§3.9); learning as turning arrows (§3.10). *Where it ends:* with a working attention layer, meeting Requirements 3 and 4 of Chapter 1, and a new failure: it has forgotten word order.

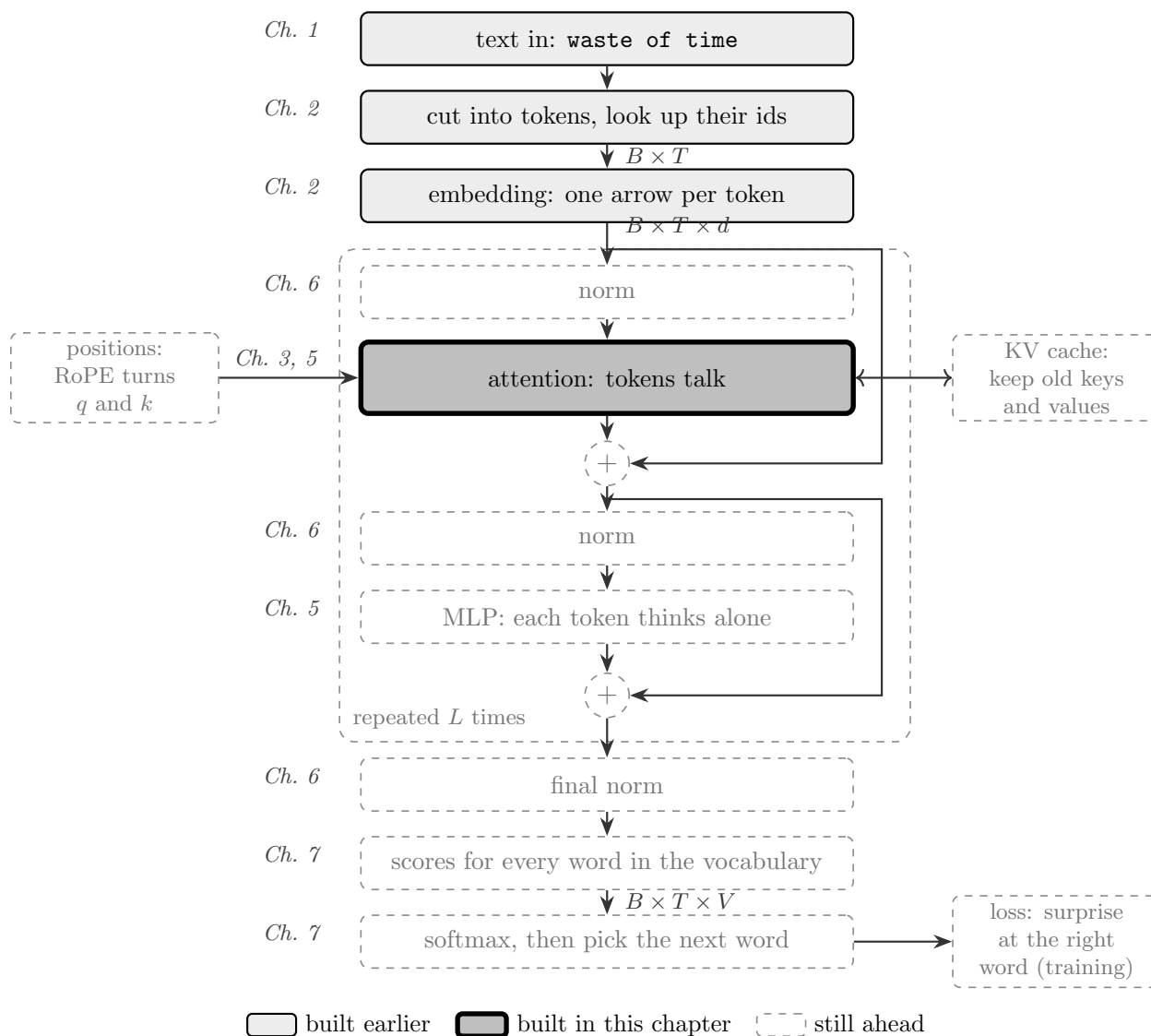


Figure 8: Where we are in the machine. You are here: Chapter 3 adds *attention*, the block where tokens talk to each other. The embedding of Chapter 2 feeds it; everything below it is still ahead.

Before you start.

1. On squared paper, draw the arrow (3,1): start at a corner, go 3 squares right and 1 square up.
Refresher: the first number is “across”, the second “up”. Negative numbers go left or down.

2. What is $2 \times (-1) + 3 \times 4$?

Refresher: multiply first, then add. A positive times a negative is negative: $-2 + 12 = 10$.

3. What are $\sqrt{16}$ and $\sqrt{256}$?

Refresher: the square root asks “which number times itself gives this?” $4 \times 4 = 16$ and $16 \times 16 = 256$.

4. What is e , and what are e^0 and e^2 ?

Refresher: $e \approx 2.718$ is a fixed number, like π . Any number to the power 0 is 1, and $e^2 = e \times e \approx 7.389$. Powers of e are always positive.

5. What is the average of 2, 4 and 9?

Refresher: add them and divide by how many there are: $15/3 = 5$.

Chapter 1 left us with four requirements. Chapter 2 met the first one: every token becomes an arrow of d numbers, a row of the embedding table, and similar tokens get similar arrows. This chapter meets Requirements 3 and 4: every token must reach every earlier token *directly*, and all of that must happen *at once*. We use the three tokens of our problem. Their rows of the toy embedding table, stacked one row per token, form the matrix X :

	X			
	1	2	3	4
waste	1	0	1	0
of	0	1	0	0
time	0	0	1	1

$T \times d = 3 \times 4$

These numbers are chosen by hand so you can follow every multiplication with a pencil. Real ones are learned and much longer: Qwen3-0.6B uses 1024 numbers per token (Chapter 10).

3.1 The simple thing, and how it breaks

Look at the token **time**. On its own, its arrow is the same in every sentence: *time flies, on time, waste of time*. But the reply we want depends on *wasted* time, time in a complaint. You learn what kind of person someone is by talking to them, not by reading their name; a token learns what it means here by listening to the tokens around it. So **time** must pull in information from the tokens before it. What is the simplest possible way? Take the average.

PICTURE IT. A class decides what to name the class pet, and every vote counts the same, whether the voter loves the pet or has never seen it.

SAY IT EXACTLY.

$$\mathbf{u}_{\text{time}} = \frac{1}{3} (\mathbf{x}_{\text{waste}} + \mathbf{x}_{\text{of}} + \mathbf{x}_{\text{time}}) = (0.333, 0.333, 0.667, 0.333) \quad [1 \times d]$$

This says **of** matters to **time** exactly as much as **waste** does. That is wrong, and no amount of training can fix it, because there is nothing to train: the weights $\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ are fixed by the length of the sentence.

WHERE THIS PICTURE BREAKS. In a class vote, every voter at least chose to vote. Here **of** gets a full third of the say without having anything to say about **time**.

WHY IT HAD TO BE THIS WAY. *We need the weights themselves to depend on the content: time should be able to ask which earlier tokens matter to it, and get a different answer in every sentence. To build “asking” and “matching” out of numbers, we need three tools from geometry. They are short, and they are the whole of the mathematics in this chapter.*

A tempting wrong idea (M3.1). *A token can use its own vector to predict the next word; mixing in other tokens is optional.* The vector already encodes the word’s meaning, so why mix? In one session the question came up as “why we cannot [...] use `u3` ... and learn to predict next tokens?”. Because `time`’s own vector is identical after *on* and after *waste of*, and the right next word is different. Only the neighbours can tell them apart.

Check yourself 3.1. Why can the token `time` not predict `Sorry` from its own arrow alone?

- (a) It can: its arrow already encodes what `time` means.
- (b) Its arrow is the same after *on* and after *waste of*, but the right next word differs.
- (c) It needs the plain average of all tokens, which settles the question.
- (d) Because its arrow does not know its position in the sentence.

In one sentence: A token’s own arrow is the same in every sentence, and a plain average cannot decide who matters, so the mixing weights must depend on content.

3.2 Tool 1: how much two arrows agree

Start from something you may have built already: a churn score. For each customer you have some clues (days since last login, support tickets this month, purchases this month) and for each clue an importance. The score does what I described in one session: “for every clue how important it is and then combine everything together”.

Predict first. A customer has clues (4, 2, 1) and the importances are (0.5, 1, −2). What is the score? Is it positive or negative?

Multiply each clue by its importance and add: $(4 \times 0.5) + (2 \times 1) + (1 \times (-2)) = 2$. Drawn as a row of clues meeting a column of importances:

$$\begin{array}{c}
 \text{clues} \\
 \text{days} \quad \text{tickets} \quad \text{buys} \\
 \text{customer} \quad \boxed{\begin{array}{|c|c|c|} \hline 4 & 2 & 1 \\ \hline \end{array}} \\
 1 \times 3
 \end{array}
 \times
 \begin{array}{c}
 \text{importance} \\
 \boxed{\begin{array}{|c|} \hline 0.5 \\ \hline 1.0 \\ \hline -2.0 \\ \hline \end{array}} \\
 3 \times 1
 \end{array}
 =
 \begin{array}{c}
 \text{score} \\
 \boxed{\begin{array}{|c|} \hline 2 \\ \hline \end{array}} \\
 1 \times 1
 \end{array}$$

highlighted row (`customer`) · highlighted column (`column 1`) = $(4)(0.5) + (2)(1.0) + (1)(-2.0) = 2$

That operation, multiply matching entries and add, is the *dot product*. Now let us see it as geometry.

PICTURE IT. An arrow is a set of directions: “3 steps east, 1 step north” is the arrow $(3, 1)$. Two friends walk out of school, one along $\mathbf{a} = (3, 1)$ and one along $\mathbf{b} = (1, 2)$. How much are they going

the same way? Shine a torch from straight above $\mathbf{a}$'s path and look at the shadow that $\mathbf{b}$ casts on it. A long shadow means they mostly agree. No shadow means they are walking at right angles. A shadow pointing backwards means they are walking apart.

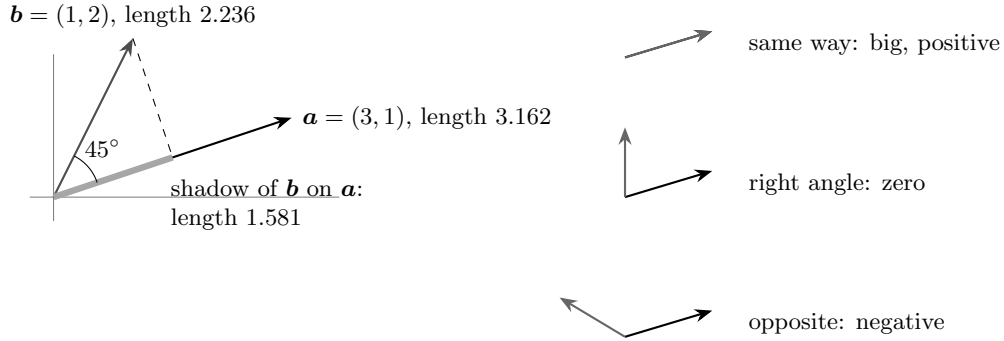


Figure 9: The dot product as a shadow. Left: our two arrows, at 45° . Right: the three cases.

Predict first. Before computing $(3, 1) \cdot (1, 2)$: from the picture, will it be positive, zero or negative?

SAY IT EXACTLY. The dot product multiplies matching entries and adds:

$$\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + \cdots + a_db_d.$$

As a picture, the row $\mathbf{a}$ meets the column $\mathbf{b}$ (we write $\mathbf{b}$ standing up, as $\mathbf{b}^\top$):

$$\begin{array}{c}
 \mathbf{a} \\
 \boxed{\begin{array}{|c|c|} \hline 3 & 1 \\ \hline \end{array}} \\
 1 \times 2
 \end{array}
 \times
 \begin{array}{c}
 \mathbf{b}^\top \\
 \mathbf{b} \\
 \boxed{\begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline \end{array}} \\
 2 \times 1
 \end{array}
 =
 \begin{array}{c}
 \mathbf{a} \cdot \mathbf{b} \\
 \mathbf{b} \\
 \boxed{5} \\
 1 \times 1
 \end{array}$$

highlighted row (a) $\cdot$ highlighted column (b) = $(3)(1) + (1)(2) = 5$

The surprising fact is that this sum measures agreement:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta,$$

where θ is the angle between the arrows. Check it: $3.162 \times 2.236 \times \cos 45^\circ = 5$. Why should multiply-and-add know about angles? Draw the third side of the triangle, $\mathbf{a} - \mathbf{b}$. Expanding its squared length entry by entry gives $\|\mathbf{a} - \mathbf{b}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - 2\mathbf{a} \cdot \mathbf{b}$, while the law of cosines says the same squared length is $\|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - 2\|\mathbf{a}\|\|\mathbf{b}\|\cos \theta$. Compare the two, and the formula falls out. Nothing in the argument cares whether there are 2 entries or 1024, so the picture holds in any number of dimensions.

Sanity check. The churn customer is an arrow too: three clues, three directions. Its score of 2 is the shadow of the customer arrow on the importance arrow, times the importance arrow's length.

WHERE THIS PICTURE BREAKS. You can draw shadows in two or three dimensions. In 1024 dimensions you cannot, but the arithmetic still works and still means “how much do these agree”. Trust the formula where the drawing stops.

WHY IT HAD TO BE THIS WAY. The dot product has exactly two knobs: *direction* ($\cos \theta$, how much the arrows agree) and *length* ($\|\mathbf{a}\| \|\mathbf{b}\|$, how strongly they say it). Keep these two knobs in mind; at the end of the chapter, learning will turn out to be nothing but turning them.

A tempting wrong idea (M3.8). *A bigger dot product always means more similar.* Dot products are often introduced as similarity scores, so bigger feels like closer. But a long arrow pointing a little off can beat a short arrow pointing exactly the same way. Similarity of direction is $\cos \theta$; the dot product also multiplies in both lengths.

Check yourself 3.2. Take $\mathbf{a} = (1, 0)$, $\mathbf{b} = (3, 3)$ and $\mathbf{c} = (1, 0)$. Which arrow points most nearly the same way as $\mathbf{a}$?

- (a) $\mathbf{b}$, because $\mathbf{a} \cdot \mathbf{b} = 3$ is bigger than $\mathbf{a} \cdot \mathbf{c} = 1$.
- (b) $\mathbf{c}$: the angle is 0° , against 45° for $\mathbf{b}$.
- (c) They agree equally, since both dot products are positive.
- (d) You cannot tell without drawing them.

In one sentence: Multiply-and-add measures how much two arrows agree in direction, scaled by how long they are.

3.3 Tool 2: a matrix is a machine that moves arrows

PICTURE IT. A matrix is a set of recipe cards. Each card says “take this much of your first number, this much of your second, $\dots$, and add”. An arrow goes in; each card produces one number of the new arrow. In my sessions I called a matrix with 64 columns “64 recipes”: 64 different ways of looking at the same token.

SAY IT EXACTLY. Multiplying a $1 \times d$ arrow by a $d \times d_k$ matrix W gives a $1 \times d_k$ arrow whose j -th entry is the dot product of the input with column j of W (card j).

Predict first. Feed **waste**, $(1, 0, 1, 0)$, into the key matrix below. Before multiplying: which rows of W_K will end up mattering?

$$\begin{array}{c}
 \mathbf{x}_{\text{waste}} \\
 \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \end{array} \\
 1 \times 4
 \end{array}
 \times
 \begin{array}{c}
 W_K \\
 \begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ \hline \end{array} \\
 4 \times 4
 \end{array}
 =
 \begin{array}{c}
 \mathbf{k}_{\text{waste}} \\
 \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 2 & 2 \\ \hline \end{array} \\
 1 \times 4
 \end{array}$$

$$\text{highlighted row (waste)} \cdot \text{highlighted column (3)} = (1)(1) + (0)(0) + (1)(1) + (0)(0) = 2$$

Each output number is the input row meeting one column (one recipe card). The highlighted column 3 gives 2; the four columns give $(0, 0, 2, 2)$. There is a second way to see the same multiplication. Because $\mathbf{x}_{\text{waste}}$ has a 1 in positions 1 and 3, the result is *row 1 plus row 3* of W_K , namely $(0, 0, 1, 1) +$

$(0, 0, 1, 1)$. A matrix sends each input direction somewhere; it can rotate arrows, stretch them, squash them, or flatten some directions to nothing.

WHERE THIS PICTURE BREAKS. Recipe cards suggest each card is separate. A matrix also has a single geometric effect on the whole space (a turn, a stretch, a squash) that no single card shows.

WHY IT HAD TO BE THIS WAY. A matrix lets the *same* token be seen in different ways. The arrow for **waste** is fixed, but W_K produces one view of it and W_Q another. If the model needs a new view, it does not change the word; it learns a new machine.

In one sentence: A matrix turns one arrow into another, one recipe (column) per output number, so one token can be viewed in many learned ways.

3.4 Tool 3: the transpose, and the table of all meetings

PICTURE IT. At the start of a tournament every player shakes hands with every other player. The organiser draws a table: players down the side, players across the top, and in each cell writes how friendly that handshake was.

SAY IT EXACTLY. Stack all the queries as rows of Q ($T \times d_k$) and all the keys as rows of K ($T \times d_k$). We want the table S with $S_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$. Matrix multiplication always pairs a *row of the left* with a *column of the right*. Our queries are already rows. Our keys are rows too, so we flip them into columns: that flip is the *transpose* $K^\top$ ($d_k \times T$). Now

$$S = QK^\top \quad [(T \times d_k)(d_k \times T) = T \times T],$$

and the cell in row i , column j is row i of Q meeting column j of $K^\top$, which is $\mathbf{q}_i \cdot \mathbf{k}_j$. There is a deeper way to see it. A column of $K^\top$ is a key, and dot-producting with an arrow measures the shadow along it. So $K^\top$ is a set of T *rulers*, one per key, and multiplying a query by $K^\top$ measures that query along every key at once.

WHERE THIS PICTURE BREAKS. A handshake is the same from both sides. Here it is not: the score of **time** looking at **waste** uses **time**'s query and **waste**'s key, while **waste** looking at **time** uses the other pair. The table is not symmetric.

WHY IT HAD TO BE THIS WAY. This is Requirement 4 in one line. An RNN produces its numbers one step after another; here all $T \times T$ scores come out of a single matrix multiplication, which is the operation a GPU's thousands of workers are built to share.

A tempting wrong idea (M3.9). *The transpose changes the vectors.* The matrix looks different afterwards, so something must have changed. But every key arrow is exactly the same arrow before and after; the transpose only stands it up as a column, so that “row meets column” pairs each query with each key.

Check yourself 3.3. In $QK^\top$ for our toy, why $K^\top$ and not K ?

- (a) Q is 3×4 and K is 3×4 , so QK is not defined; $QK^\top$ is 3×3 , one score per pair of tokens.
- (b) The transpose turns the keys so they match the queries better.
- (c) The transpose rescales the keys to length 1.

(d) Either works; QK gives the same numbers in a different order.

In one sentence: Flipping the keys into columns lets one multiplication produce every query-key score at once, without changing any arrow.

3.5 The fix: three questions per token

Now we can build “asking”. Attention gives every token three different views of itself, each made by its own learned machine:

- **Query $q = xW_Q$:** *what am I looking for?*
- **Key $k = xW_K$:** *what can I be found by?*
- **Value $v = xW_V$:** *what do I hand over if I am chosen?*

PICTURE IT. A library. The query is what you type into the search box. The key is the label on each book’s spine. The value is what is written inside the book. You match your search against the labels, but you take home the contents.

WHERE THIS PICTURE BREAKS. In a library you take home a few whole books. Attention takes home a blend: a little of every book, in proportions set by how well each label matched.

WHY IT HAD TO BE THIS WAY. Why three, and not one arrow for everything? Because what makes you *findable* is not the same as what you *offer*, and neither is the same as what you *need*. **waste** should be found by any token asking “is there a judgement nearby?”, but what it hands over is the content of that judgement, and what **waste** itself is looking for is something else again. One arrow cannot point in three directions.

A tempting wrong idea (M3.4). *Query, key and value are three copies of the same thing; one vector would do.* All three are made from the same token arrow, so they look like copies. The question “why can’t we [...] assign one recipe instead of creating another 64 uh recipes?” came up in a session. The answer is the paragraph above: each is a different machine applied to the same arrow, and each points where its own job needs it.

A tempting wrong idea (M3.6). *Values are the words’ original meanings.* Values are made from the embeddings, so it is tempting to read them as the dictionary meaning. They are a learned *view* made for handing over, built by W_V . In the toy, $x_{\text{waste}} = (1, 0, 1, 0)$ but v_{waste} is different, as you will see below.

In the toy we use the machines in Figure 10 (chosen for pencil-friendliness; real ones are learned). Each picture feeds all three tokens through one machine at once: X (3×4) times W (4×4) gives one new row per token. One row meeting one column is highlighted and written out; check one more cell yourself before reading on.

Check yourself 3.4. Why does attention give each token three arrows instead of one?

- It does not need to: the three are copies, and one would do.
- What a token looks for, what it can be found by and what it hands over are different jobs, so each gets its own machine.

$$\begin{array}{c}
\begin{array}{c} X \\ \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \text{of} & 0 & 1 & 0 & 0 \\ \hline \text{time} & 0 & 0 & 1 & 1 \\ \hline \end{array} \\ 3 \times 4
\end{array}
\times
\begin{array}{c}
\begin{array}{c} W_Q \\ \begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline \end{array} \\ 4 \times 4
\end{array}
=
\begin{array}{c}
\begin{array}{c} Q \\ \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \text{of} & 0 & 1 & 0 & 0 \\ \hline \text{time} & 0 & 0 & 1 & 1 \\ \hline \end{array} \\ 3 \times 4
\end{array}
\end{array}$$

highlighted row (**time**) · highlighted column (**column 4**) = (0)(0) + (0)(0) + (1)(0) + (1)(1) = **1**

$$\begin{array}{c}
\begin{array}{c} X \\ \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \text{of} & 0 & 1 & 0 & 0 \\ \hline \text{time} & 0 & 0 & 1 & 1 \\ \hline \end{array} \\ 3 \times 4
\end{array}
\times
\begin{array}{c}
\begin{array}{c} W_K \\ \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ \hline 0 & 1 & 0 & 0 \\ \hline \end{array} \\ 4 \times 4
\end{array}
=
\begin{array}{c}
\begin{array}{c} K \\ \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 0 & 0 & 2 & 2 \\ \hline \text{of} & 1 & 0 & 0 & 0 \\ \hline \text{time} & 0 & 1 & 1 & 1 \\ \hline \end{array} \\ 3 \times 4
\end{array}
\end{array}$$

highlighted row (**waste**) · highlighted column (**column 4**) = (1)(1) + (0)(0) + (1)(1) + (0)(0) = **2**

$$\begin{array}{c}
\begin{array}{c} X \\ \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \text{of} & 0 & 1 & 0 & 0 \\ \hline \text{time} & 0 & 0 & 1 & 1 \\ \hline \end{array} \\ 3 \times 4
\end{array}
\times
\begin{array}{c}
\begin{array}{c} W_V \\ \begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline \end{array} \\ 4 \times 4
\end{array}
=
\begin{array}{c}
\begin{array}{c} V \\ \text{waste} \quad \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \text{of} & 0 & 0 & 0 & 1 \\ \hline \text{time} & 0 & 1 & 1 & 0 \\ \hline \end{array} \\ 3 \times 4
\end{array}
\end{array}$$

highlighted row (**waste**) · highlighted column (**column 2**) = (1)(0) + (0)(0) + (1)(1) + (0)(0) = **1**

Figure 10: The three machines applied to the whole sentence: $Q = XW_Q$, $K = XW_K$, $V = XW_V$, each of shape $T \times d_k = 3 \times 4$. W_Q is the identity in the toy, so $Q = X$; W_K and W_V move the arrows.

- (c) The values must keep the original meaning, so they cannot be shared with queries.
- (d) Only the current token needs a query; keys and values are for the other tokens.

In one sentence: Each token gets a query (what it looks for), a key (what it is found by) and a value (what it hands over), each from its own learned machine.

3.6 Worked example: what does time attend to?

Predict first. Look at Q and K above. Which earlier token do you expect **time** to attend to most, and roughly what share? Write it down before reading on.

Step 1: match (every handshake at once). $S = QK^\top$: every row of Q meets every column of $K^\top$. The highlighted row **time** meeting the highlighted column **waste** gives the cell circled in S .

$$\begin{array}{c}
\begin{array}{c} Q \\ 3 \times 4 \end{array}
\begin{array}{c} K^\top \\ 4 \times 3 \end{array}
=
\begin{array}{c} S \\ 3 \times 3 \end{array}$$

waste	1	0	1	0
of	0	1	0	0
time	0	0	1	1

 $\times$

waste	0	1	0
of	0	0	1
time	2	0	1

 $=$

waste	2	1	1
of	0	0	1
time	4	0	2

highlighted row (**time**) $\cdot$ highlighted column (**waste**) = $(0)(0) + (0)(0) + (1)(2) + (1)(2) = 4$

Figure 11: $S = QK^\top$ for our sentence, $(3 \times 4)(4 \times 3) = 3 \times 3$. Row **time** reads $(4, 0, 2)$. Grey cells are a token meeting a token that comes *after* it; Step 3 removes them.

Look at row **time** through the geometry of §3.2. Its query is $\mathbf{q}_{\text{time}} = (0, 0, 1, 1)$, of length 1.414. Each score is length $\times$ length $\times$ agreement:

key	$\ \mathbf{k}\ $	angle to $\mathbf{q}_{\text{time}}$	$\cos \theta$	$\ \mathbf{q}\ \ \mathbf{k}\ \cos \theta$	dot product
waste	2.828	0.0°	1.000	4	4
of	1.000	90.0°	0.000	0	0
time	1.732	35.3°	0.816	2	2

time’s question points *exactly* along **waste**’s label (0 degrees), at right angles to **of**’s (nothing in common), and partly along its own. That is what “**waste** is relevant to **time**” means, geometrically.

Step 2: scale. Divide by $\sqrt{d_k} = \sqrt{4} = 2$. Row **time** becomes $(2, 0, 1)$. Why we divide is the subject of §3.8.

Step 3: cover the answers. Remember from §1.9 that one sentence is three exercises: after **waste**, guess the next word; after **waste of**, guess the next; and so on. The exercise “after **waste of**, guess the next word” must not be allowed to see **time**: that is the answer.

PICTURE IT. An exam where all the questions and all the answers are printed on one long sheet, in order. Before each question, you cover everything below it with a ruler. A fraud-detection model is trained the same way: it may never use information recorded *after* the transaction, because in production that information does not exist yet. A model that used it would look brilliant in training and fail in the real world.

WHERE THIS PICTURE BREAKS. On the exam sheet you cover the answers once. Here every row has its own ruler at a different height, and all the rulers are applied at the same moment.

SAY IT EXACTLY. Set every score above the diagonal (the grey cells) to $-\infty$. This is *causal masking*:

$$\begin{array}{c}
S/\sqrt{d_k}, \text{ masked} \\
T \times T
\end{array}$$

waste	1	$-\infty$	$-\infty$
of	0	0	$-\infty$
time	2	0	1

Step 4 will turn $-\infty$ into a weight of exactly zero. So the row for **of** is built only from **waste** and **of**; our script checks that changing the arrow of **time** to anything at all leaves the first two rows of the output untouched. That is why the three exercises can be solved *in the same pass* without cheating.

A tempting wrong idea (M3.7). *The mask exists only to stop cheating.* That is how it is usually introduced, and in a session the answer given was “it might copy from the future tokens.” That is true, and it is half the story. The other half is speed: because each row ignores everything after it, one pass can train all T positions at once. Without the mask you would need T separate passes, one per prefix.

Step 4: turn scores into shares (softmax). For each row: raise e to each score, then divide by the row’s total. For **time**, the three steps are the three rows of this table:

	row time		
	waste	of	time
score	2	0	1
e^{score}	7.389	1.000	2.718
share	0.665	0.090	0.245

total of e^{score} : 11.107; shares add to 1

In symbols, $\mathbf{a}_{\text{time}} = (\frac{7.389}{11.107}, \frac{1.000}{11.107}, \frac{2.718}{11.107}) = (0.665, 0.090, 0.245)$. **time** gives 66.5% of its attention to **waste**, 9.0% to **of** and 24.5% to itself. How close was your prediction? Compare that with the fixed 33.3% each of §3.1: the weights now come from the content.

Sanity check. The three shares are positive and add to 1, and the biggest score got the biggest share. A share above 1 or a negative share would mean an arithmetic slip.

A tempting wrong idea (M3.2). *Attention scores are already percentages.* They come out as one number per pair, like a table of shares, so it is natural to read $(4, 0, 2)$ as “4 parts to **waste**”. In a session the question was put as “is it really saying some sort of percentages?”. No: scores can be negative and do not add to 1. Only after softmax are they shares.

A tempting wrong idea (M3.5). *The current token only needs a query; keys and values are for the other tokens.* The current token is the one asking, so its own key seems pointless; “why does team time need keys and values?” came up in a session. Look at the result: **time** gives 24.5% of its attention to *itself*, through its own key and value. And later tokens, such as **Sorry**, will look back at **time** through the same key and value.

Step 5: collect. The shares are shares *of the values*, the contents of the books:

$$\mathbf{o}_{\text{time}} = 0.665 \mathbf{v}_{\text{waste}} + 0.090 \mathbf{v}_{\text{of}} + 0.245 \mathbf{v}_{\text{time}} = (0.665, 0.910, 0.245, 0.090) \quad [1 \times d_k]$$

and for the whole sentence at once, $\mathbf{O} = \mathbf{A}\mathbf{V}$ ($T \times T$ times $T \times d_k$). The highlighted row of $\mathbf{A}$ holds **time**’s shares; the highlighted column of $\mathbf{V}$ holds the second number of each value arrow:

	A		V		O									
waste	1.000	0.000	0.000	$\times$	1	1	0	0	$=$	waste	1.000	1.000	0.000	0.000
of	0.500	0.500	0.000		0	0	0	1		of	0.500	0.500	0.000	0.500
time	0.665	0.090	0.245		0	1	1	0		time	0.665	0.910	0.245	0.090
	3×3				3×4					3×4				

highlighted row (time) · highlighted column (column 2) = (0.665)(1) + (0.090)(0) + (0.245)(1) = **0.910**

Every row of O is a blend of value arrows, mixed in the proportions the token asked for. The arrow of **time** that leaves this step is no longer “time in general”; it carries 66.5% of what **waste** had to say.

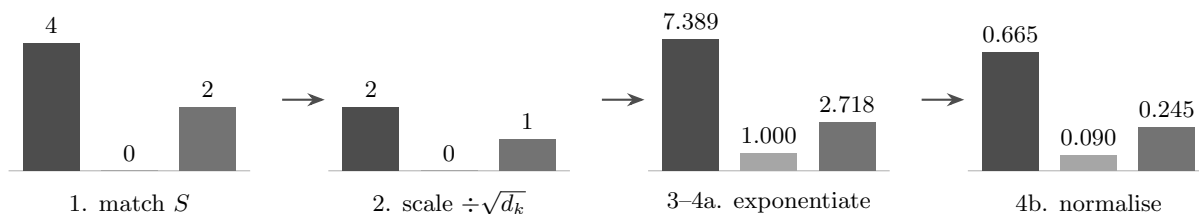


Figure 12: The frames of attention for the query **time**. Bars: **waste**, **of**, **time**. In the web edition this strip is an animation you can scrub and re-run with your own weights.

Check yourself 3.5. After masking, why can the three next-word exercises of **waste of time** be trained in a single pass?

- (a) The mask only stops cheating; it has nothing to do with doing them together.
- (b) Each row’s output depends only on that token and earlier ones, so every row is an honest answer to its own exercise.
- (c) The raw scores are already percentages, so no extra step is needed per row.
- (d) Because **time** gives no attention to itself.

*In one sentence: Match queries with keys, scale, mask the future, turn scores into shares, and blend the values: **time** ends up carrying 66.5% of **waste**.*

3.7 Why softmax, and not “divide by the total”?

PICTURE IT. A pizza is shared among friends according to votes. Softmax is the sharing rule.

WHERE THIS PICTURE BREAKS. Votes are never negative, but scores can be. And the pizza rule we are about to build rewards a clear winner far more than a fair-minded host would.

Three properties make e^{score} the right rule, and each one is a requirement you could have written down yourself:

- (i) *Shares must be positive*, but scores can be negative (arrows pointing apart). e^s is positive for every s , and $e^{-\infty} = 0$, which is what the mask needs.
- (ii) *Only differences should matter*. If every friend’s vote goes up by 10, nobody deserves more

pizza. Indeed $e^{s+10} = e^{10}e^s$, and the common factor e^{10} cancels in the division; our script confirms that $(2, 0, 1)$ and the same scores plus 10 give identical shares.

- (iii) *Leaders should lead.* Each extra point of score multiplies a share's weight by $e \approx 2.718$, so a clear winner gets a clear majority instead of a narrow one.

Predict first. Keep **time**'s query pointing the same way but make it twice as long. Will the share for **waste** go up, go down, or stay at 0.665?

Its scaled scores become $(4, 0, 2)$ and the shares become $(0.867, 0.016, 0.117)$: same ranking, much more decisive. *The direction of a query decides what it looks for; its length decides how sure it is.*

A tempting wrong idea (M3.11). *Softmax shares depend on the ratios between scores, as when dividing by the total.* That is how percentages are usually made, so it is the first rule anyone tries. Dividing by the total would give $(0.667, 0.000, 0.333)$ for $(2, 0, 1)$ and $(0.389, 0.278, 0.333)$ for $(7, 5, 6)$: the shares would change when every score moves up by the same amount. Softmax gives $(0.665, 0.090, 0.245)$ for both, because of property (ii).

Check yourself 3.6. After scaling, **time**'s scores are $(2, 0, 1)$. Add 5 to all three, giving $(7, 5, 6)$, and apply softmax. What happens to the shares? Then pick the reason.

- (a) They do not change.
- (b) They all grow, because every score grew.
- (c) They become more even, since 7, 5 and 6 are closer together relative to their size.
- (d) **waste**'s share grows most, because it had the largest score.

Reason: (i) softmax only sees differences between scores; (ii) shares must still add to 1; (iii) e^x grows faster for bigger x .

In one sentence: Softmax makes shares that are positive, ignore a common shift, and favour the leader, and a longer query makes it more decisive.

3.8 Why divide by $\sqrt{d_k}$?

In the toy the division looks cosmetic. It is not.

PICTURE IT. Toss a coin d_k times; step forward on heads, back on tails. After 4 tosses you are usually within about 2 steps of the start; after 1024 tosses, usually within about 32. The more tosses, the further you wander, but not d_k steps: about $\sqrt{d_k}$.

WHERE THIS PICTURE BREAKS. Coin tosses are exactly $+1$ or -1 . The products inside a dot product take any size; the $\sqrt{d_k}$ rule still holds when each product has mean 0 and variance 1, which is how weights start out.

Predict first. At $d_k = 1024$, with no scaling, what share do you expect the top key to get out of 8 random keys, before any training? Closer to $1/8$, to $1/2$, or to 1?

SAY IT EXACTLY. A dot product of two unrelated arrows is a sum of d_k products that are equally likely to be positive or negative, like those coin tosses. If each coordinate has mean 0 and variance 1, the sum has variance d_k , so its typical size is $\sqrt{d_k}$. We measured it (random queries against 8 random keys, 4000 trials, fixed seed):

d_k	spread of $\mathbf{q} \cdot \mathbf{k}$ (theory $\sqrt{d_k}$)	largest share, unscaled	largest share, scaled
4	2.03 (2.0)	0.531	0.347
64	7.89 (8.0)	0.874	0.360
1024	31.47 (32.0)	0.969	0.359

Read the third column with property (iii) of softmax in mind. At the widths real models use, random scores are tens of points apart, so softmax hands almost everything to one key (0.969 on average at $d_k = 1024$) *before the model has learned anything*. It has made up its mind at random, and, as Chapter 8 shows, a softmax that has made up its mind barely responds to learning. Dividing by $\sqrt{d_k}$ shrinks the scores back to the size of a one-step walk, and the starting shares stay spread (about 0.359) at every width.

WHY IT HAD TO BE THIS WAY. A design rule you will meet again in Chapter 11: *keep the size of every signal independent of the size of the model*, so that making the model bigger does not silently change how it behaves.

A tempting wrong idea (M3.3). *Dividing by $\sqrt{d_k}$ normalises the vectors to length 1.* Cosine similarity divides by the lengths, and the formula looks alike, so the two get merged. A session put it as “why do we divide by the total length? yeah, we want to get it back to normalized vector.” But $\sqrt{d_k}$ is one fixed number for the whole model; it does not depend on any vector’s length. It controls the *spread* of scores across random pairs, not the length of any one arrow. Lengths still matter after scaling, as §3.7 showed.

Check yourself 3.7. Why are attention scores divided by $\sqrt{d_k}$?

- (a) To give every query and key length 1.
- (b) To keep the typical spread of scores near 1 at any width, so softmax does not pick a winner at random before training.
- (c) To turn the scores into percentages.
- (d) Because in the toy $\sqrt{d_k} = 2$ makes the arithmetic tidy.

In one sentence: Random dot products grow like $\sqrt{d_k}$, so dividing by it keeps softmax undecided at the start, at any model width.

3.9 Not one token, not one sentence: the whole batch at once

We followed **time** by hand, but the machine never handles one token. Look again at the shapes: $S = QK^\top$ computed every row at once. Now put a second customer reply in the same batch, **not very useful**, with its own arrows (rows of the same embedding table). The batch is two matrices stacked one behind the other, and so is the table of scores:

PICTURE IT. Two rooms run the same handshake tournament in parallel, each with its own table. Nobody shakes hands with the other room.

WHERE THIS PICTURE BREAKS. Two real rooms would need two organisers. Here one multiplication fills both tables; the rooms are separate only because the numbers are arranged that way.

SAY IT EXACTLY. Stack the sentences into one block of numbers of shape $B \times T \times d = 2 \times 3 \times 4$. The same machines W_Q, W_K, W_V apply to every token of every sentence, so Q, K, V have shape

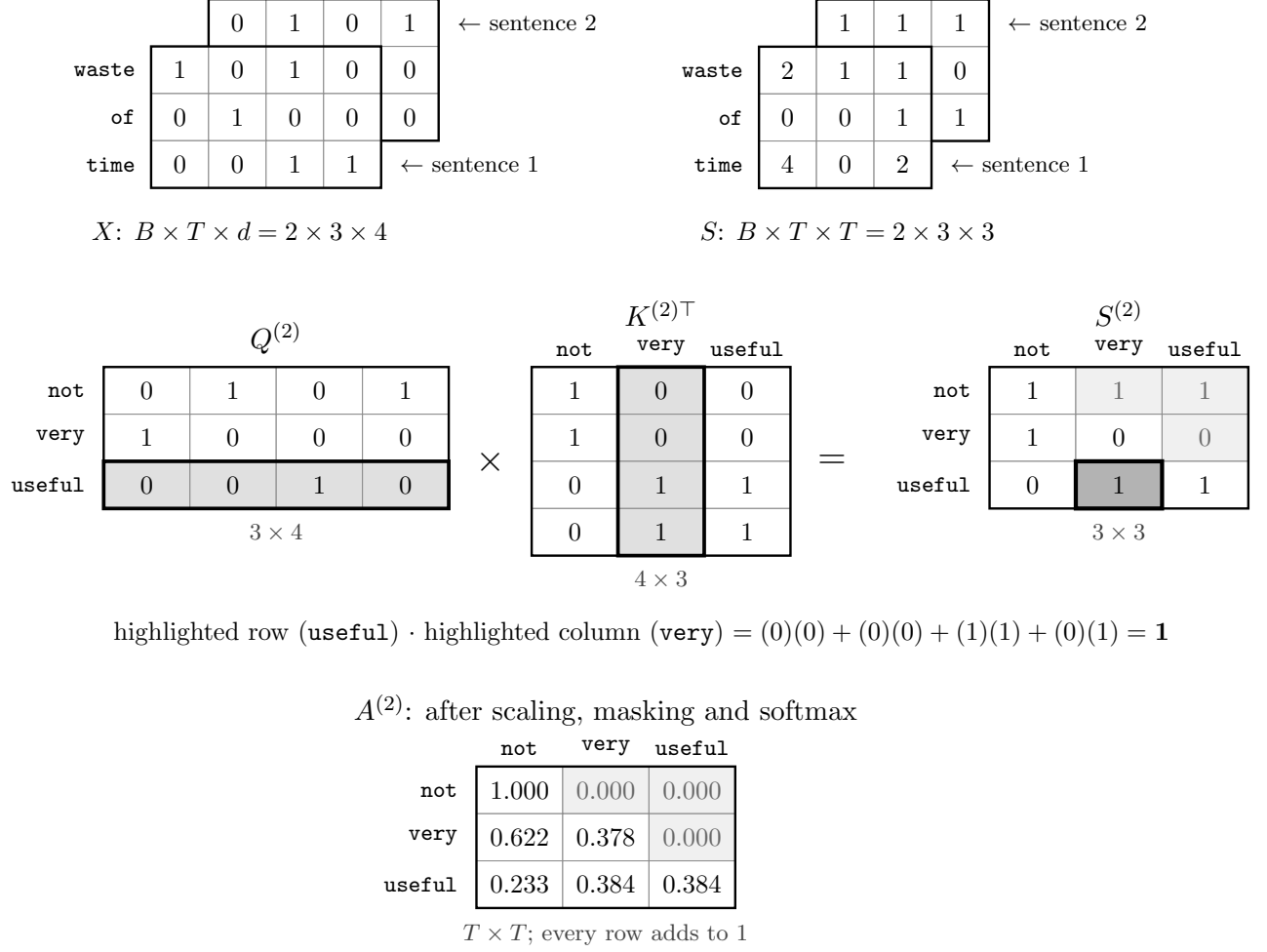


Figure 13: Top: a batch of $B = 2$ sentences. Each sentence keeps its own 3×4 matrix X and gets its own 3×3 table S . Bottom: the second sentence’s table, built by the same multiplication as in Figure 10’s machines, and its shares after scaling, masking and softmax.

$B \times T \times d_k$. The transpose now flips only the last two axes of K , and

$$S = QK^\top \quad [(B \times T \times d_k)(B \times d_k \times T) = B \times T \times T]$$

is B tournament tables computed side by side: 18 scores in one operation, with no score ever mixing the two sentences. After the mask and softmax, $O = AV$ has shape $B \times T \times d_k$, and every one of its $B \times T = 6$ rows will go on to make a next-word guess. Our script checks that sentence 1 gives exactly the same numbers in the batch as it gave alone.

WHY IT HAD TO BE THIS WAY. This is where Chapter 1’s history pays off. The RNN had to walk each sentence word by word. Here a batch of B sentences of T tokens is a handful of large matrix multiplications, and those multiplications are what GPUs are made of.

In one sentence: A batch is B separate attention tables computed in one operation, giving $B \times T$ guesses per pass.

3.10 How learning shapes the arrows

Our toy machines were chosen by hand. In a real model, W_Q , W_K and W_V start as random numbers and are learned from the next-word game. Chapter 8 does the bookkeeping; here is the whole idea as a picture.

Every score is $\|q\|\|k\|\cos\theta$, so there are only two ways to change how much one token attends to another: *turn* the arrows (change the angle) or *stretch* them (change the length). Training plays the game millions of times. Each time the model guesses the next word, it checks the answer, and every weight is nudged a tiny step in the direction that would have made the right word a little more likely:

- if looking more at **waste** would have helped **time** predict **Sorry**, the query of **time** is turned a little towards the key of **waste** (and that key a little towards the query);
- if looking at **of** only added noise, the arrows are turned a little apart;
- if the right choice was clear, the query is stretched, making the softmax more decisive (§3.7);
- and the value arrows are turned towards whatever turned out to be useful to hand over.

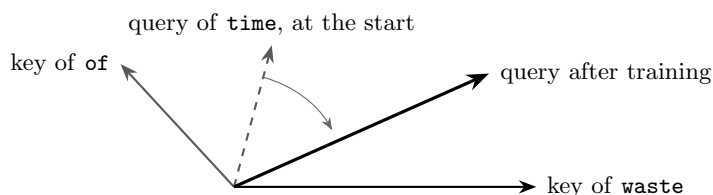


Figure 14: Learning as turning and stretching arrows (schematic, not to scale). The query is turned towards the key that helped, away from the one that did not, and stretched to be more confident. Training never moves the arrows directly; it changes the machines W_Q, W_K, W_V that produce them.

WHERE THIS PICTURE BREAKS. The figure turns one arrow. Training turns the *machines*, so every arrow those machines produce turns together, including arrows for sentences that are not in this batch.

That last point makes this powerful. A nudge learned from **waste of time** also moves the arrows produced for **waste of money** and **waste of effort**, because those words have nearby arrows (Requirement 1). That is how a model trained on finitely many sentences handles sentences it has never seen, the exact thing the counting notebook of §1.3 could not do.

In one sentence: Learning only turns and stretches arrows, by adjusting the machines that make them, so lessons spread to similar words.

3.11 What we built, and the next failure

Abstraction — attention is a soft lookup table. A Python dictionary takes a query, finds the one key that matches *exactly*, and returns its value. Attention does the same, softly: every key matches a little, measured by a dot product, and you get a blend of all the values. Because “a little” is a smooth number, the lookup can be learned. The same pattern appears whenever a system must pick from candidates by relevance: a search engine ranking documents, a recommender ranking products, or the retriever in an AI agent’s memory system, which scores stored memories against the current conversation and decides how much of each one reaches

the model.

Predict first. Swap the first two tokens and feed `of waste time`. Will the output for `time` change?

The output for `time` is

$$\mathbf{o}_{\text{time}} = (0.665, 0.910, 0.245, 0.090),$$

identical to before. The mask decides who may look at whom, but the scores themselves cannot tell `waste of` from `of waste`: to `time`, its past is a bag, not a sequence. We have quietly lost Requirement 2, the very thing the RNN gave us for free, and we are back to the bag of words of §1.4. Chapter 4 fixes it, and the fix (RoPE) turns out to be a *rotation* of exactly these query and key arrows, by an angle that depends on the position.

A tempting wrong idea (M3.10). *Attention already knows the order of the words.* Masking is about order, so it seems order must be handled. In a session the point was made that “any of this transformation does not take that”: none of W_Q , W_K , W_V sees a position. The mask only says which tokens are visible; among the visible ones, attention cannot tell who came first.

Check yourself 3.8. Feed `of waste time` instead of `waste of time`. What happens to the output for `time`?

- (a) It is unchanged: `time` sees the same set of earlier tokens, and nothing in the scores knows their order.
- (b) It changes, because the causal mask encodes word order.
- (c) It changes, because `waste`’s value is now in a different row.
- (d) It changes, because the scores are percentages of position.

In one sentence: Attention is a learnable soft lookup that meets Requirements 3 and 4, but on its own it cannot tell word order.

Looking back. (1) Chapter 1: why does one sentence of T tokens give T training exercises? (2) Chapter 1: which requirement did the RNN fail that attention now meets, and which one did attention lose? (3) Chapter 2: where do the rows of X come from? (4) Chapter 2: why does a tokenizer split a rare word into pieces instead of giving it its own token?

Exercises

Exercise 3.1. Compute by hand the attention output for the token `of`. Then explain, using only the geometry of $\mathbf{q}_{\text{of}}$ and the keys, why its two shares are equal.

Exercise 3.2. A model has $d_k = 256$. Without scaling, roughly how far apart would the scores of unrelated tokens typically be? What would you divide by, and why that number?

Exercise 3.3. Fill the gaps for the token `useful` in the second sentence of §3.9. Its raw scores are $(0, 1, 1)$. After dividing by $\sqrt{d_k}$ they are $(_, _, _)$. Exponentiate, divide by the total, and check your answer against the last row of $A^{(2)}$.

Exercise 3.4. A second customer has clues $(10, 0, 3)$ with the same importances $(0.5, 1, -2)$. Compute the churn score. Is the customer arrow pointing with or against the importance arrow?

Answers to Chapter 3

Check yourself 3.1. (b). (a) is M3.1: the arrow does not change with context. (c) is the uniform average that §3.1 showed cannot learn. (d) is a real issue (Chapter 4), but even with positions, one arrow cannot see its neighbours.

Check yourself 3.2. (b). (a) is M3.8: **b** wins on dot product only because it is longer. (c) mixes up “positive” with “equal”. (d): the angles answer it without drawing.

Check yourself 3.3. (a). (b) and (c) are M3.9: the transpose changes no arrow. (d) is wrong because QK is not defined.

Check yourself 3.4. (b). (a) is M3.4, (c) is M3.6, (d) is M3.5.

Check yourself 3.5. (b). (a) is M3.7: the mask is also what lets all rows train together. (c) is M3.2. (d) is false: **time** gives 24.5% to itself.

Check yourself 3.6. (a), for reason (i). Softmax gives (0.665, 0.090, 0.245) for both lists. (b) is M3.2. (c) is M3.11: dividing by the total would give (0.389, 0.278, 0.333), the shift option (c) expects. (d) is M3.8.

Check yourself 3.7. (b). (a) is M3.3. (c) is M3.2: softmax does that. (d) is true in the toy only.

Check yourself 3.8. (a). (b) is M3.10. (c): the rows moved, but the set of value arrows **time** blends is the same. (d) mixes two wrong ideas.

Looking back. (1) Each prefix asks for its own next word, and the mask lets all prefixes be answered in one pass. (2) Attention meets Requirement 3 (direct reach) and loses Requirement 2 (order). (3) Each token id picks one row of the embedding table. (4) So the vocabulary stays a fixed size and no word is ever unknown: a rare word becomes known pieces.

Exercise 3.1. $\mathbf{q}_{\text{of}} = (0, 1, 0, 0)$ points along the second direction only. Both $\mathbf{k}_{\text{waste}}$ and $\mathbf{k}_{\text{of}}$ have 0 there: both are at right angles to the query, so both scores are 0, the softmax of (0, 0) is (0.5, 0.5), and $\mathbf{o}_{\text{of}} = 0.5 \mathbf{v}_{\text{waste}} + 0.5 \mathbf{v}_{\text{of}} = (0.500, 0.500, 0.000, 0.500)$. A query at right angles to every key has nothing to prefer, and attention falls back to the plain average of §3.1. *Uniform attention is what “no information” looks like.*

Exercise 3.2. About $\sqrt{256} = 16$ points, enough for softmax to put nearly all its weight on one key at random. Divide by $\sqrt{256}$ to bring the typical spread back to about one point, whatever d_k is.

Exercise 3.3. (0, 0.5, 0.5); the two equal scores give equal shares, and the last row of $A^{(2)}$ shows the result. The strategy that works: do one row at a time, and check the shares add to 1.

Exercise 3.4. $(10, 0, 3) \cdot (0.5, 1, -2) = -1$. Negative: the customer arrow points partly against the importance arrow.

What you now know

- The dot product as weighted clues, as a shadow, and as lengths times agreement, and why the two formulas match.
- A matrix as recipe cards and as a machine that moves arrows; the transpose as standing keys up as columns.
- Queries, keys and values, and a full attention computation by hand, with masking and softmax.
- Why softmax uses e^x , why scores are divided by $\sqrt{d_k}$, and how a batch runs in one operation.
- Learning as turning and stretching arrows, and the next failure: attention alone ignores order.

4 Order: from sinusoids to rotations (RoPE)

Map of this chapter. Chapter 3 ended with a failure: attention cannot tell **waste of time** from **waste time**. This chapter gives the model a sense of order. We follow the fixes in the order researchers found them: learning a vector for each position (GPT-2), adding fixed waves (the 2017 transformer), and finally rotating the queries and keys by an angle that grows with position (RoPE, used by Qwen3 and most current models). On the way we prove, with numbers, the property that makes rotation the winner: the score between two tokens depends only on *how far apart* they are.

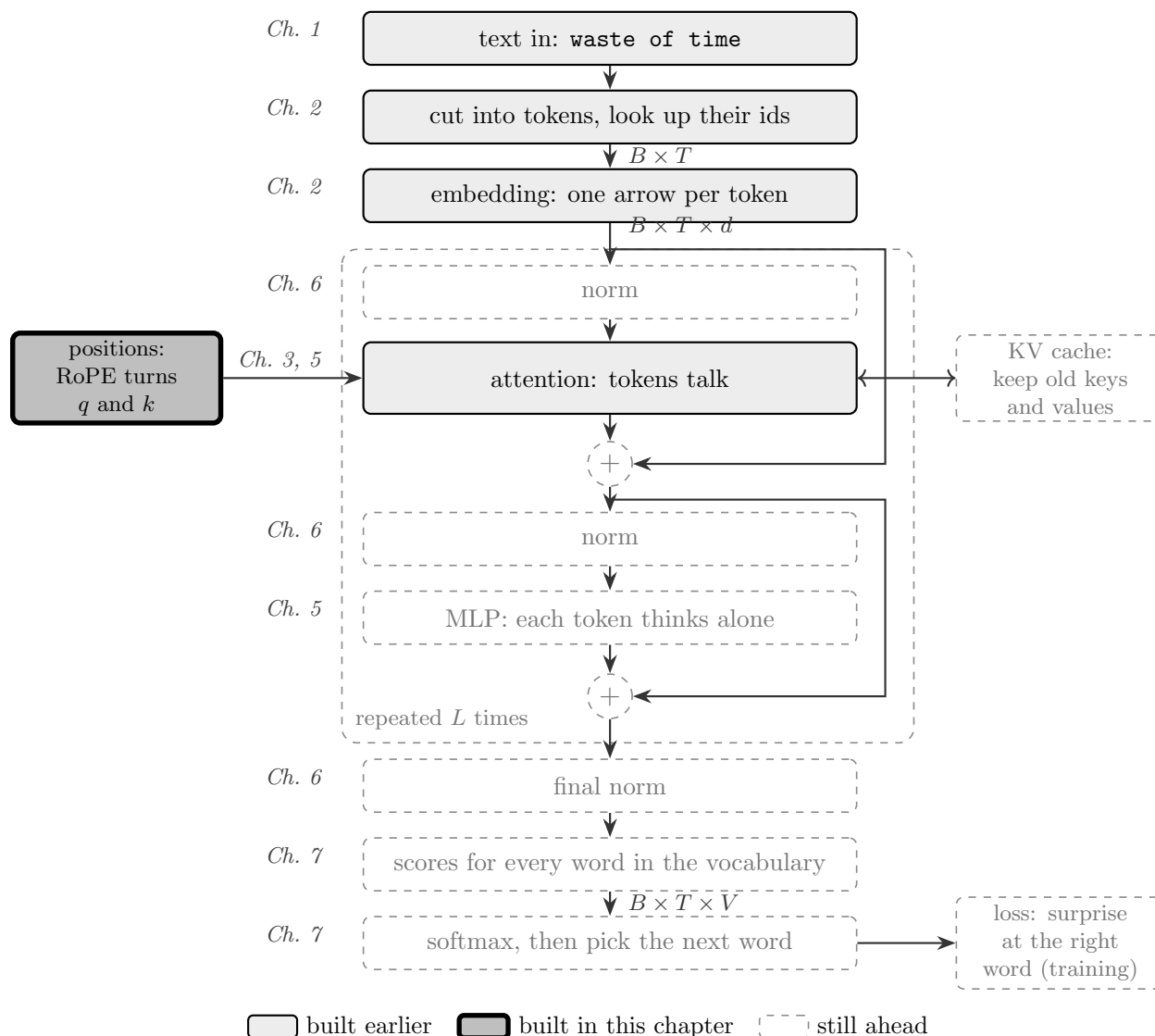


Figure 15: Where we are in the machine. You are here: Chapter 4 adds the *positions* box on the left. It does not add a new layer to the stack. It turns each query and key inside attention by an angle that depends on where the token stands, so attention can finally tell **waste of time** from **of waste time**.

Before you start. (1) A clock's minute hand turns 6° per minute. Where is it after 15 minutes? (2) What are $\cos 0^\circ$, $\cos 90^\circ$ and $\cos 180^\circ$? (3) If you turn an arrow without stretching it, does its length

change? (4) The second hand of a clock and the hour hand both point at 12. Can you tell the time from the second hand alone?

Refreshers. (1) $15 \times 6^\circ = 90^\circ$: pointing at 3. (2) 1, 0 and -1 : same direction, right angle, opposite. (3) No; turning keeps the length. (4) No: it repeats every minute, so you also need the slower hands. Keep this last answer in mind; it is the whole idea of §4.6.

4.1 The failure, once more

Predict first. In Chapter 3 we fed `of waste time` instead of `waste of time`. Before looking back, guess: did the output for `time` change a lot, a little, or not at all?

Not at all. Both orders give $\mathbf{o}_{\text{time}} = (0.665, 0.910, 0.245, 0.090)$. Look at where order could have entered the computation. The queries, keys and values are made from each token’s own arrow, and the token’s arrow is the same wherever it stands. The score $\mathbf{q} \cdot \mathbf{k}$ only sees the two arrows. The mask only decides which tokens are visible. So to `time`, its past is a bag, as in the bag-of-words attempt of §1.4. For a support reply this is serious: `not useful`, `very good` and `very useful`, `not good` would look alike once again.

A tempting wrong idea (M4.1). “The mask already tells the model the order.” The mask tells each token *which* tokens it may see. It says nothing about *where* they stand. In session 9 we put it as a question: “do you really think any of the transformation takes that positional thing”? The answer is no, and §4.1 shows it with numbers: same set of earlier tokens, same output.

In one sentence: attention compares arrows that carry no position, so we have to put position into the arrows ourselves.

4.2 Attempt: add a position vector

Predict first. The simplest idea: add the position number to the arrow, so the first token gets $+0$, the second $+1$, and so on. Can you see a problem with token number 1000?

PICTURE IT. Give every seat in a cinema a small badge with its seat number, and pin it onto whoever sits there. The person is the same; the badge says where they are.

Why plain numbers fail. If you add 1000 to every coordinate of the 1000th token, its arrow is dominated by the badge and the word is lost: $(0, 0, 1, 1)$ becomes $(1000, 1000, 1001, 1001)$, and every token far into a long text looks the same. The badge has to stay small, and it has to look different at every position.

Two ways to make good badges. GPT-2 (Radford and colleagues, 2019) *learned* a table of badges, one row per position, exactly like the embedding table of Chapter 2: 1024 positions, so it cannot handle a 1025th token at all. The original transformer (Vaswani and colleagues, 2017) used *fixed waves* instead. For position p and coordinate pair i (pairs $0, 1; 2, 3; \dots$),

$$\text{PE}(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right), \quad \text{PE}(p, 2i + 1) = \cos\left(\frac{p}{10000^{2i/d}}\right).$$

Each pair of coordinates is a wave with its own speed. For our toy, $d = 4$ and positions 0, 1, 2, so the badges form a $T \times d = 3 \times 4$ table, one row per position. Figure 16 adds it to the token arrows, cell by cell.

					PE									$X + PE$			
X																	
						sin	cos	sin	cos								
waste	1	0	1	0	pos 0	0.00	1.00	0.00	1.00	=	waste	1.00	1.00	1.00	1.00		
of	0	1	0	0	pos 1	0.84	0.54	0.01	1.00		of	0.84	1.54	0.01	1.00		
time	0	0	1	1	pos 2	0.91	-0.42	0.02	1.00		time	0.91	-0.42	1.02	2.00		
token arrows					position badges				what attention now sees								

Figure 16: Sinusoidal badges as a matrix. Row p of PE is the badge for position p ; it is added to the token arrow in the same row. Notice the second pair of columns barely changes from row to row (it is the slow wave), while the first pair changes a lot (the fast wave).

Every entry of PE stays between -1 and 1 , however long the text. Add these badges to the toy arrows and run the attention of Chapter 3 again. **time**’s output becomes $(0.956, 0.866, 1.238, 0.738)$ for **waste of time** and $(1.694, 0.988, 1.113, 0.465)$ for **of waste time**. The two orders now differ (by up to 0.738 in one coordinate). Requirement 2 is back.

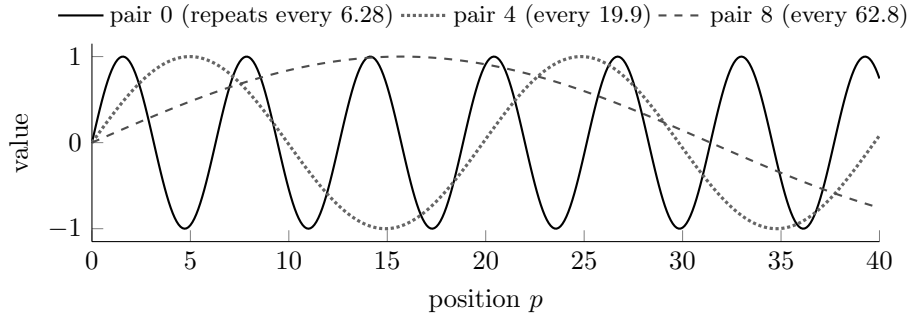


Figure 17: Three of the waves used for $d = 64$: solid is fast, dotted is medium, dashed is slow. Fast waves tell neighbours apart; slow waves tell far positions apart. For $d = 64$ the repeat lengths run from 6.28 to 47,117 positions.

A tempting wrong idea (M4.2). “Add the position number itself.” The number grows without limit and swamps the word. A useful badge has two properties at once: it stays small, and it is different at every position. Waves have both.

WHERE THIS PICTURE BREAKS. A badge is pinned on and stays separate from the person. Here the badge is *added* into the same numbers as the word, so the model receives a single blended arrow and must learn to pull the two apart. That blending is the weakness the next fix removes.

In one sentence: adding a small, position-specific vector to each arrow restores order, as long as the vector stays bounded and differs from position to position.

4.3 What should position mean? Absolute or relative

Tell me: in **waste** of **time**, what matters to **time** about **waste**? That **waste** is word number 0 of the message, or that it is two words back? Put the same three words in the middle of a long complaint and **waste** might be word 340. Nothing about the meaning has changed. What stayed the same is the *distance*: two words back.

Added badges mix this up. The score between **time** and **waste** becomes a dot product of (word + badge) with (word + badge), which contains word-badge cross terms that depend on the absolute positions. The model can learn to undo this, but we are asking it to do extra work. Shaw, Uszkoreit and Vaswani (2018) proposed putting the *distance* directly into attention. Su and colleagues (2021) found an elegant way to do it with no extra table at all: rotation.

A tempting wrong idea (M4.3). “Positions have to be learned, like every other weight.” In session 9 we said: “we want you to also learn positions.” Learning them (GPT-2) is one option. The sinusoids of §4.2 and the rotations of §4.4 are fixed formulas with nothing to learn, and they work for positions never seen in training.

In one sentence: what usually matters is how far apart two tokens are, so a good position method should make the score depend on distance.

4.4 The fix: rotate the questions and the labels

Predict first. If you turn two arrows by the *same* angle, does the angle between them change? If you turn one by 30° and the other by 50° , what is the angle between them now, compared with before?

PICTURE IT. A clock. Split each query and key into pairs of numbers, and draw each pair as a clock hand. A token at position p has each of its hands turned forward by p ticks. Turning both hands of a pair by the same amount leaves their angle unchanged; only the *difference* in ticks shows up between them. That is exactly “how far apart”.

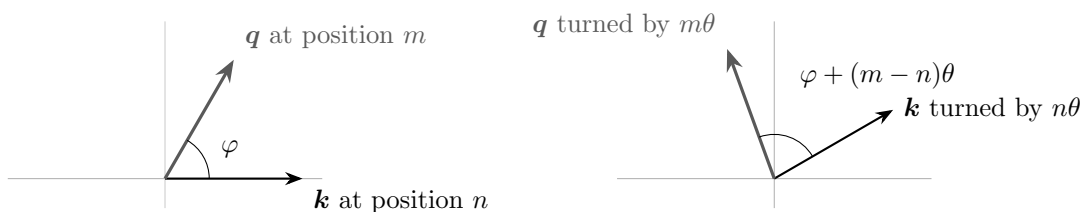


Figure 18: Rotary position embedding on one pair of coordinates. Each arrow is turned by its own position times θ . The angle between them changes by $(m - n)\theta$, which depends only on the distance.

The numbers. Our toy has $d_k = 4$, so two pairs: coordinates $(1, 2)$ and $(3, 4)$. RoPE turns pair i by $p\theta_i$ at position p , with $\theta_i = b^{-2i/d}$ for a chosen base b . Real models use $b = 10,000$ or more. With $d = 4$ that makes the second pair turn by only 0.01 radians per position, too slow to see in a three-word example (the shares barely move: $(0.665, 0.090, 0.245)$). So for the toy we use $b = 4$, which gives $\theta = (1, 0.50)$ radians per position: the first hand turns 57.3° per word, the second 28.65° .

How does a turn act on numbers? It is a 2×2 matrix multiplying one pair. Take **time**'s query, $(0, 0, 1, 1)$, at position 2. Its second pair is $(1, 1)$ and must turn by $2\theta_2 = 2 \times 0.50 = 1$ radian. Figure 19 does that multiplication.

$$\begin{array}{ccc}
 R(2\theta_2) & & \text{pair 2 of } \mathbf{q}_{\text{time}} \quad \text{turned pair} \\
 \begin{array}{|c|c|} \hline 0.540 & -0.841 \\ \hline 0.841 & 0.540 \\ \hline \end{array} & \times & \begin{array}{c} x_3 \begin{array}{|c|} \hline 1 \\ \hline \end{array} \\ x_4 \begin{array}{|c|} \hline 1 \\ \hline \end{array} \end{array} = \begin{array}{|c|} \hline -0.301 \\ \hline 1.382 \\ \hline \end{array} \\
 2 \times 2 & & 2 \times 1 \quad \quad 2 \times 1 \\
 \\
 \text{row 1 of } R \cdot \text{column } (x_3, x_4) = (0.540)(1) + (-0.841)(1) = -0.301
 \end{array}$$

Figure 19: One turn, as a matrix product. Row 1 of the rotation matrix meets the pair $(x_3, x_4) = (1, 1)$ and gives the first number of the turned pair. The second row gives the second number the same way. The pair's length is $\sqrt{2}$ before and after.

Only queries and keys are turned; values are left alone. Do the same for every pair of every row, each row by its own position, and you get Figure 20. **waste** sits at position 0, so its row is not turned at all.

$$\begin{array}{ccc}
 \begin{array}{l} \text{waste (p=0)} \\ \text{of (p=1)} \\ \text{time (p=2)} \end{array} & \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ \hline \end{array} & \longrightarrow & \begin{array}{l} Q \text{ after: row } p \text{ turned by } p\theta \\ \begin{array}{|c|c|c|c|} \hline 1.00 & 0.00 & 1.00 & 0.00 \\ \hline -0.84 & 0.54 & 0.00 & 0.00 \\ \hline 0.00 & 0.00 & -0.30 & 1.38 \\ \hline \end{array} \end{array} \\
 \\
 \begin{array}{l} \text{waste (p=0)} \\ \text{of (p=1)} \\ \text{time (p=2)} \end{array} & \begin{array}{|c|c|c|c|} \hline 0 & 0 & 2 & 2 \\ \hline 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 \\ \hline \end{array} & \longrightarrow & \begin{array}{l} K \text{ after: row } p \text{ turned by } p\theta \\ \begin{array}{|c|c|c|c|} \hline 0.00 & 0.00 & 2.00 & 2.00 \\ \hline 0.54 & 0.84 & 0.00 & 0.00 \\ \hline -0.91 & -0.42 & -0.30 & 1.38 \\ \hline \end{array} \end{array}
 \end{array}$$

Figure 20: Queries and keys of **waste** of **time** before and after RoPE. Row p is turned by p times θ (both of its pairs, each with its own speed). Row **waste** ($p = 0$) is unchanged.

Read the last row of Q after turning: that is **time**'s turned query. Now swap the first two words: **waste** moves to position 1, one step from **time** instead of two. Figure 21 shows the whole score table $QK^\top$ (before scaling) for both orders, without and with RoPE.

waste of time: no positions				waste of time: with RoPE			
	waste of time				waste of time		
waste	2	1	1	waste	2.00	0.54	-1.21
of	0	0	1	of	0.00	0.00	0.54
time	4	0	2	time	2.16	0.00	2.00

of waste time: no positions				of waste time: with RoPE			
	of waste time				of waste time		
of	0	0	1	of	0.00	0.00	-0.42
waste	1	2	1	waste	0.54	2.00	-0.44
time	0	4	2	time	0.00	3.51	2.00

Figure 21: Score tables for both word orders. Grey cells are covered by the causal mask. Compare the outlined **time** rows. Without positions they hold the same three numbers, only reordered: **waste** scores 4 wherever it stands. With RoPE **waste**’s score depends on how far it is from **time**.

For the query **time**, after scaling and softmax:

order	dot products (before scaling)	shares after softmax	output $\mathbf{o}_{\text{time}}$
waste of time	(2.161, 0.000, 2.000)	(0.442, 0.150, 0.408)	(0.442, 0.850, 0.408, 0.150)
of waste time	(0.000, 3.510, 2.000)	(0.105, 0.609, 0.286)	(0.609, 0.895, 0.286, 0.105)

Each row lists the tokens in sentence order. **waste**’s share goes from 0.442 to 0.609 when it moves closer. Order now changes the answer, and nothing was added to any arrow: the script checks that every turned arrow keeps its length.

SAY IT EXACTLY. For a pair (x_1, x_2) at position p , RoPE applies the rotation

$$\begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \cos p\theta & -\sin p\theta \\ \sin p\theta & \cos p\theta \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

to every pair of every query and key, each pair with its own θ_i . Write R_p for this rotation. Because turning by $m\theta$ and then measuring against something turned by $n\theta$ is the same as turning by $(m - n)\theta$ alone,

$$(R_m \mathbf{q}) \cdot (R_n \mathbf{k}) = \mathbf{q} \cdot (R_{n-m} \mathbf{k}),$$

which depends on m and n only through $m - n$. We checked this numerically on 1000 random queries and keys of width 8 at random positions, shifting both positions by the same amount: the largest difference in score was 4.1×10^{-13} , which is rounding error. In the toy, **time** at position 2 looking at **waste** at 0 scores 2.161, and so does **time** at 7 looking at **waste** at 5.

A tempting wrong idea (M4.4). “RoPE changes the token’s embedding.” It does not touch the embedding table or the values. It turns the query and key after W_Q and W_K have made them, inside every attention layer, and it never changes their length. What a token hands over (its value) is the same wherever it stands; only *who finds it* depends on distance.

Check yourself 4.1. After RoPE, `time` at position 2 and `waste` at position 0 give a score of 2.161. What score do they give at positions 12 and 10?

- (a) 2.161, because only the distance (2) matters.
- (b) A larger score, because later words are turned more.
- (c) A score near 0, because the arrows have been turned so far that they no longer line up.
- (d) It cannot be known without recomputing the embeddings at the new positions.

WHY IT HAD TO BE THIS WAY. Rotation is the one operation that keeps lengths (so the scale of every score stays as Chapter 3 needs it, §3.8) and turns “two absolute positions” into “one distance” for free. No table to learn, no limit on length from a table, and no blending of word and badge.

In one sentence: RoPE turns each pair of query and key coordinates by an angle proportional to position, so every attention score depends only on how far apart the two tokens are.

4.5 Many hands at different speeds

Predict first. A hand that turns 57.3° per word comes back to almost the same place after 44 words (44 radians is 7.00 full turns). Could the model then confuse a word 44 tokens away with a word in the same spot?

With one hand, yes. In the toy, the fast hand gives $\cos(44\theta_1) = 0.9998$, almost exactly what distance 0 gives ($\cos 0 = 1$). But the slow hand turns $\theta_2 = 0.50$ per word, and $\cos(44\theta_2) = -1.0000$: completely different. Like the second, minute and hour hands of a clock, each hand alone repeats, but together they pin down the distance. This is why θ_i is spread over many speeds, from one turn every few tokens to one turn every many thousands.

SAY IT EXACTLY. With head width 128 and base $b = 10,000$, the slowest pair completes a turn only every 54,410 positions. Long-context models raise the base: the published Qwen3 configuration (Qwen3 technical report, 2025; to be checked against its `config.json` in Chapter 10) uses $b = 1,000,000$, which with head width 128 makes the slowest hand turn once every 5,063,256 positions.

WHERE THIS PICTURE BREAKS. A clock tells the time of day exactly. Attention does not read the hands directly; it only sees the dot product, which mixes all hands together, weighted by how much of the query and key sit in each pair. If a query puts all its weight on the fast pairs, it effectively looks at short distances only. The model learns where to put its weight.

A tempting wrong idea (M4.6). “Rotations repeat, so a far word gets confused with a near one.” Each hand repeats, the set of hands does not, over the lengths the model is built for. Past the slowest hand’s turn the guarantee weakens, which is one reason models have a maximum context length.

Check yourself 4.2. Why does RoPE use many different speeds θ_i instead of one?

- (a) One speed would make every word at a multiple of the repeat distance look as if it were at distance 0.
- (b) Different speeds make the arrows longer.
- (c) Each speed stores the absolute position of one word.

(d) More speeds mean the values can be rotated too.

In one sentence: fast hands separate neighbours, slow hands separate distant words, and together they make distances distinguishable over long texts.

4.6 What we built

Abstraction — encode relationships, not coordinates. RoPE works because it stores what the task needs (a distance) rather than what is easy to write down (an absolute index). The same move pays off in system design. A memory system that stores “three messages ago the user said X” survives the conversation getting longer; one that stores “message 17 said X” has to be fixed every time messages are inserted. When you design a feature, ask: is my model asked to compare things, and am I giving it the comparison or two raw positions to subtract?

Looking back. (a) In §3.2, the dot product has two knobs. Which one does RoPE leave alone, and which one does it change? (b) Why does §3.8’s division by $\sqrt{d_k}$ still work after RoPE? (c) Which requirement from Chapter 1 did Chapter 3 lose and this chapter restore?

Exercises

Exercise 4.1. Turn the arrow $(1, 0)$ by 90° and the arrow $(0, 1)$ by 90° . What is the dot product before and after? Explain in one sentence why this is the heart of RoPE.

Exercise 4.2. In the toy with $b = 4$, the first hand turns 1 radian per position. How many positions does it take to come back close to where it started? Why could that matter for a 30-word support message?

Exercise 4.3. GPT-2 learned one position vector per position for 1024 positions, each of width 768. How many numbers is that table? What happens when a text has 1025 tokens?

Exercise 4.4. A colleague says: “With sinusoidal badges, the model knows absolute position; with RoPE, it knows only relative position. So RoPE loses information.” What would you answer? Use the support reply as your example.

Answers to Chapter 4

Check yourself 4.1. (a). Both positions moved by 10, the distance is still 2, and the score depends only on the distance. (b) and (c) treat RoPE as tied to absolute position (M4.5); (d) reflects M4.4, since the embeddings are never changed.

Check yourself 4.2. (a). With a single speed, distances that differ by a whole number of turns look the same (M4.6 is the over-reaction to this). (b) is wrong because rotation keeps lengths; (c) is M4.5; (d) is M4.4, values are never rotated.

Looking back. (a) It leaves the lengths alone and changes the angle. (b) Rotation keeps each arrow’s length, so dot products keep the same typical size and the same $\sqrt{d_k}$ applies. (c) Requirement 2: knowing which word came where.

Exercise 4.1. Before: $(1, 0) \cdot (0, 1) = 0$. After: $(0, 1) \cdot (-1, 0) = 0$. Turning both arrows by the same angle leaves their angle, and so their dot product, unchanged: only the *difference* in turning can change a score.

Exercise 4.2. One full turn is $2\pi \approx 6.28$ radians, so a little more than every 6 positions, and it is close to a full turn after 44 (7.00 turns). In a 30-word message, the fast hand alone would mix up some distances; the slower hand (0.50 radians per word, a turn every 12.6 words) separates them. With only two hands the toy is still limited; real models have one hand per pair of coordinates, 64 per head for head width 128.

Exercise 4.3. $1024 \times 768 = 786,432$ numbers. A 1025th token has no row, so the model cannot process it without extending and re-training the table.

Exercise 4.4. For the reply, what matters is that **Sorry** comes right after the complaint, not that the complaint started at word 340 of the chat. Relative position is what the task needs. Absolute position can still be recovered where it matters (for example, through the distance to the first token of the text). In practice, most current open models, including Llama (Touvron and colleagues, 2023) and Qwen3, use RoPE, which suggests little that matters is lost.

What you now know.

- Why attention needs help with order, shown with the swap test.
- The three historical fixes: learned position vectors, sinusoidal waves, and rotation (RoPE).
- Why rotation makes every score depend only on distance, checked numerically.
- Why many rotation speeds are needed, and what raising the base does for long texts.

5 Many heads, the MLP, the residual stream

Map of this chapter. *The problem:* one attention head lets **time** look back, but it can only ask one question at a time, and nothing yet lets a token *think* about what it heard. *The steps:* see why one head blurs two answers into one (§5.1); split the width into two heads that ask different questions (§5.2); stitch their answers back together (§5.3); give each token a private checklist of patterns, the MLP (§5.4); and keep every layer’s work on a shared notebook, the residual stream (§5.5). *Where it ends:* with one complete transformer block that, on our three tokens, predicts **of** after **waste**, **time** after **of** and **Sorry** after **time**, all in one pass.

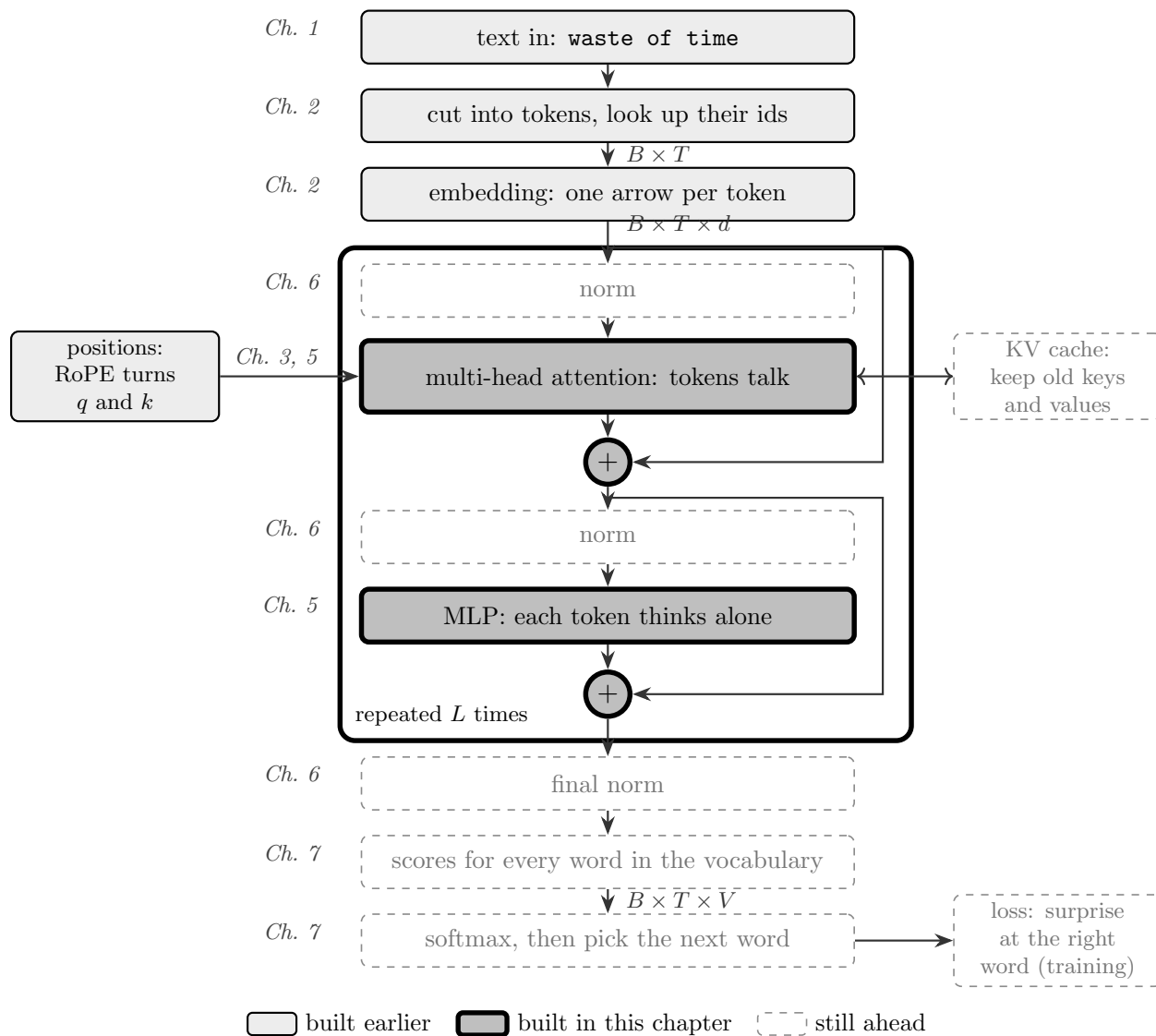


Figure 22: Where we are in the machine. You are here: Chapter 5 turns the single attention head into several heads, adds the MLP (each token thinks alone), and adds the two + circles of the residual stream. With these, the dashed frame becomes one complete layer, which a real model repeats L times. The norms are still ahead (Chapter 6).

Before you start. Five quick questions. Answer each, then read the refresher under it.

1. Add the arrows $(1, 2)$ and $(3, -1)$.
Refresher: add matching entries: $(1 + 3, 2 - 1) = (4, 1)$. On paper, put the second arrow's tail on the first arrow's head.
2. What is the average of the arrows $(1, 1, 0, 0)$ and $(0, 0, 0, 1)$?
Refresher: add them and halve every entry: $(0.5, 0.5, 0, 0.5)$.
3. Write the list $(2, 1)$ followed by the list $(3, 4)$ as one list.
Refresher: placing lists end to end is called *concatenating*: $(2, 1, 3, 4)$.
4. What is the larger of 0 and -1.5 ? Of 0 and 1.569 ?
Refresher: the function $\max(0, x)$ keeps positive numbers and turns negative ones into 0. Here: 0 and 1.569 .
5. In Chapter 3, how much of its attention did **time** give to **waste**?
Refresher: 66.5%, and 9.0% to **of**. If that number is hazy, reread §3.6 before going on.

5.1 One head, one blurred answer

Predict first. In Chapter 3, one head computed *one* set of shares for **time**, and handed back *one* blend of value arrows. Suppose **time** needs two different things: the judgement in **waste**, and the fact that **of** sits right before it. Can one blend carry both, and keep them apart? Make a guess before reading on.

PICTURE IT. Two colleagues report to you at the same moment, into the same microphone. You hear one mixed sound. You can tell that two people spoke, but you cannot tell who said what.

SAY IT EXACTLY. Take the value arrows of Chapter 3 for **waste** and **of**. A head that splits its attention evenly hands back half of one plus half of the other, entry by entry (top row of Figure 23). Now take two *different* value arrows (bottom row). Their even blend is the same arrow. Our script checks this.

$$\begin{array}{rcccl}
 & \mathbf{v}_{\text{waste}} & & \mathbf{v}_{\text{of}} & \\
 0.5 \times & \begin{array}{|c|c|c|c|} \hline 1 & 1 & 0 & 0 \\ \hline \end{array} & + & 0.5 \times & \begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 1 \\ \hline \end{array} & = & \begin{array}{|c|c|c|c|} \hline 0.5 & 0.5 & 0.0 & 0.5 \\ \hline \end{array} & \text{what time receives} \\
 & \text{another pair} & & & & & \\
 0.5 \times & \begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 1 \\ \hline \end{array} & + & 0.5 \times & \begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 0 \\ \hline \end{array} & = & \begin{array}{|c|c|c|c|} \hline 0.5 & 0.5 & 0.0 & 0.5 \\ \hline \end{array} & \text{the same blend}
 \end{array}$$

Figure 23: One head, one blend. Top: half of $\mathbf{v}_{\text{waste}}$ plus half of $\mathbf{v}_{\text{of}}$, entry by entry. Bottom: a different pair of arrows gives exactly the same shaded result, so the result alone cannot say who said what.

Two different pairs of reports, one identical blend: after the averaging, the token cannot know which pair it heard. A weighted average is a single arrow, and a single arrow can be built from many different mixtures.

WHERE THIS PICTURE BREAKS. The microphone picture suggests the information is gone for good. In a trained model the value arrows are learned, and they can be made to point in separate directions so that a blend is still readable. The real limit is that one head has one set of shares: it must decide

how much **waste** matters and how much **of** matters with a single softmax, so emphasising one always takes attention from the other.

WHY IT HAD TO BE THIS WAY. A softmax hands out shares that add to 1. So within one head, attention is a fixed budget, and every question competes for it. To ask two questions without competition, you need two budgets.

A tempting wrong idea (M5.2). *One head can attend to two tokens equally well, by giving each 50%. It can attend to both, but it cannot keep their messages apart, and it cannot make one share bigger without shrinking the other. That is the difference between “listening to two people” and “hearing two people talk at once”.*

Check yourself 5.1. A single head gives **time** the shares $(0.5, 0.5, 0)$ over **waste**, **of**, **time**. What can **time** tell from the output alone?

- (a) Exactly what **waste** said and exactly what **of** said.
- (b) The average of what they said, but not which part came from whom.
- (c) Nothing at all, because averages destroy everything.
- (d) What **time** itself said, since it always attends to itself.

In one sentence: One head returns one weighted average, so asking two questions needs two heads.

5.2 Two heads, each half as wide

PICTURE IT. Instead of one reporter who listens to everyone, send two reporters with different briefs. One is told “find any judgement nearby”. The other is told “find the small linking words”. Each writes a short note.

The budget does not grow. Our arrows still have $d = 4$ numbers, so the two heads get $d_k = 2$ each. In the sessions this came up as “how that 64 became 32”: a model with width 64 and two heads gives each head 32 recipes. Each head is narrower, and in exchange there are more of them.

Worked example: head 1, fully. Head 1 has its own three machines, each 4×2 (four numbers in, two out):

$$W_Q^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W_K^{(1)} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}, \quad W_V^{(1)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad [d \times d_k = 4 \times 2]$$

Figure 24 does the three products. In each one, the outlined row meets the outlined column, and the dark cell is the number they make.

$$\begin{array}{c}
\begin{array}{c} \mathbf{x}_{\text{time}} \\ \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 1 \\ \hline \end{array} \\
1 \times 4
\end{array}
\times
\begin{array}{c}
W_Q^{(1)} \\
\begin{array}{|c|c|} \hline 0 & 0 \\ \hline 0 & 0 \\ \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} \\
4 \times 2
\end{array}
=
\begin{array}{c}
\mathbf{q}_{\text{time}}^{(1)} \\
\text{time} \end{array} \begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array} \\
1 \times 2
\end{array}$$

row **time** · column 1 of $W_Q^{(1)} = (0)(0) + (0)(0) + (1)(1) + (1)(0) = \mathbf{1}$

$$\begin{array}{c}
\begin{array}{c} X \\ \text{waste} \\ \text{of} \\ \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ \hline \end{array} \\
3 \times 4
\end{array}
\times
\begin{array}{c}
W_K^{(1)} \\
\begin{array}{|c|c|} \hline 1 & 1 \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline 0 & 0 \\ \hline \end{array} \\
4 \times 2
\end{array}
=
\begin{array}{c}
K^{(1)} \\
\text{waste} \\ \text{of} \\ \text{time} \end{array} \begin{array}{|c|c|} \hline 2 & 2 \\ \hline 0 & 0 \\ \hline 1 & 1 \\ \hline \end{array} \\
3 \times 2
\end{array}$$

row **waste** · column 1 of $W_K^{(1)} = (1)(1) + (0)(0) + (1)(1) + (0)(0) = \mathbf{2}$

$$\begin{array}{c}
\mathbf{q}_{\text{time}}^{(1)} \\
\text{time} \end{array} \begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array}
\times
\begin{array}{c}
K^{(1)\top} \\
\begin{array}{|c|c|c|} \hline \text{waste} & \text{of} & \text{time} \\ \hline 2 & 0 & 1 \\ \hline 2 & 0 & 1 \\ \hline \end{array} \\
2 \times 3
\end{array}
=
\begin{array}{c}
\text{raw scores} \\
\text{time} \end{array} \begin{array}{|c|c|c|} \hline \text{waste} & \text{of} & \text{time} \\ \hline 4 & 0 & 2 \\ \hline \end{array} \\
1 \times 3
\end{array}$$

row **time** · column **waste** = $(1)(2) + (1)(2) = \mathbf{4}$

Figure 24: Head 1, step by step. Top: **time**'s arrow through $W_Q^{(1)}$ gives its 1×2 query. Middle: every token's arrow through $W_K^{(1)}$ gives one key per row. Bottom: the query against the transposed keys gives one raw score per token; **waste** wins with 4.

So the raw scores are $(4, 0, 2)$. Dividing by $\sqrt{d_k} = \sqrt{2}$ gives $(2.828, 0.000, 1.414)$, and the softmax gives the shares

$$\mathbf{a}_{\text{time}}^{(1)} = (0.768, 0.045, 0.187).$$

Head 1 gives 76.8% of its attention to **waste**. It is the judgement-finder.

Head 2: your turn. Head 2 uses

$$W_Q^{(2)} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad W_K^{(2)} = \begin{bmatrix} 0 & 0 \\ 2 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Predict first. Look at the second row of $W_K^{(2)}$: only **of**, whose arrow is $(0, 1, 0, 0)$, can produce a non-zero key. Which token will head 2 favour? Fill in the question marks of Figure 25: $\mathbf{q}_{\text{time}}^{(2)}$, the three keys, then the three row scores. Check with Answer 5.2.

$$\begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \mathbf{x}_{\text{time}} \\ \hline 0 & 0 & 1 & 1 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline W_Q^{(2)} \\ \hline 0 & 0 \\ \hline 0 & 0 \\ \hline 1 & 0 \\ \hline 0 & 0 \\ \hline \end{array} = \begin{array}{|c|c|} \hline \mathbf{q}_{\text{time}}^{(2)} \\ \hline ? & ? \\ \hline \end{array}$$

$$\begin{array}{c} \text{waste} \\ \text{of} \\ \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline X \\ \hline 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ \hline \end{array} \times \begin{array}{|c|c|} \hline W_K^{(2)} \\ \hline 0 & 0 \\ \hline 2 & 0 \\ \hline 0 & 0 \\ \hline 0 & 0 \\ \hline \end{array} = \begin{array}{c} \text{waste} \\ \text{of} \\ \text{time} \end{array} \begin{array}{|c|c|} \hline K^{(2)} \\ \hline ? & ? \\ \hline ? & ? \\ \hline ? & ? \\ \hline \end{array}$$

Figure 25: Head 2, your turn. Multiply as in Figure 24 and fill in the cells marked ?.

Sanity check. The scaled scores are

$$(0.000, 1.414, 0.000),$$

and the shares come out as $(0.164, 0.673, 0.164)$: 67.3% on **of**. The shares add to 1, and the favourite is the token you predicted.

time attends to	waste	of	time
Chapter 3 (one head, $d_k = 4$)	66.5%	9.0%	24.5%
Head 1 ($d_k = 2$)	76.8%	4.5%	18.7%
Head 2 ($d_k = 2$)	16.4%	67.3%	16.4%

Each head now spends its whole budget on its own question. One head mostly hears **waste**; the other mostly hears **of**.

WHERE THIS PICTURE BREAKS. Our two reporters have tidy, nameable briefs because we wrote the weights by hand. Heads in trained models are learned, often overlap, and many have no neat job. Michel, Levy and Neubig (2019) removed most of the heads of trained models one at a time and found that many could go with little loss.

WHY IT HAD TO BE THIS WAY. Splitting the width keeps the cost the same (the four machines of the block still hold $4 \times d^2$ numbers in total) while giving the token several independent budgets of attention. The price is that each head asks with fewer numbers, so each question is cruder.

A tempting wrong idea (M5.1). *More heads is always better.* With the width fixed, more heads means narrower heads. Taken to the extreme, a head with $d_k = 1$ compares single numbers, a very blunt question. In the sessions it came up in exactly these words: “it is not always it helps”. The number of heads is a trade-off to choose, and Chapter 11 treats it as one.

A tempting wrong idea (M5.5). *Each head has a fixed job, like “the grammar head”.* Our toy heads do, because we designed them. In a trained model a head’s job is whatever helped predict the next token, and it is often a mixture.

Check yourself 5.2. A model has width $d = 64$. You change it from 2 heads to 8 heads and keep d fixed. What happens?

- (a) Each head becomes 4 times wider, so each question is sharper.
- (b) Each head drops from 32 to 8 numbers, and there are 4 times as many independent sets of shares.
- (c) The number of attention weights grows 4 times.
- (d) The model necessarily gets better, because more heads see more.

In one sentence: Splitting the width into heads gives a token several separate attention budgets at no extra cost, each one asking a cruder question.

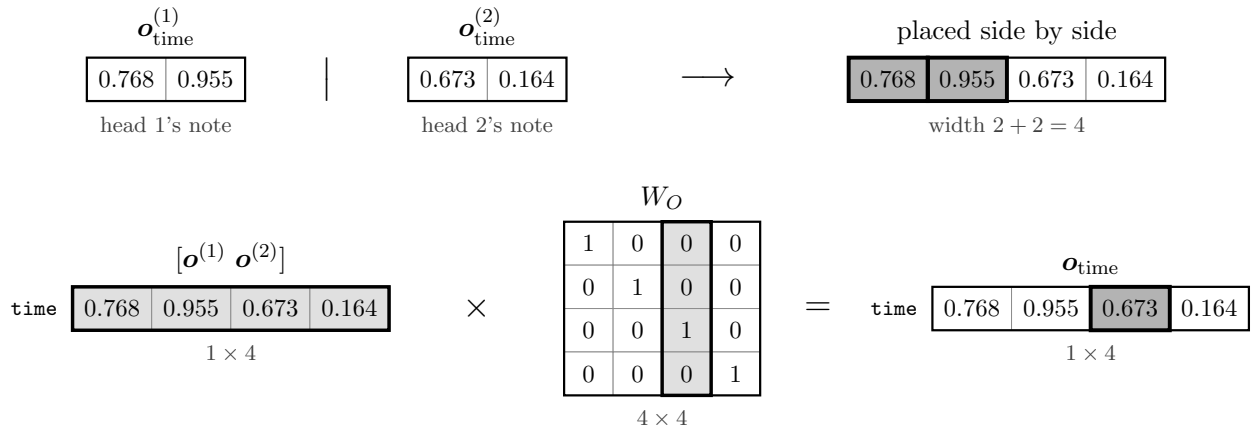
5.3 Stitching the notes together

PICTURE IT. The two reporters staple their notes side by side, and an editor rewrites the stapled page in the house style.

SAY IT EXACTLY. Each head hands back a short arrow for **time**, of width $d_k = 2$. Concatenate them into one arrow of width $d = 4$, then multiply by an output matrix W_O ($d \times d$), the editor:

$$\mathbf{o}_{\text{time}} = [\mathbf{o}_{\text{time}}^{(1)} \mathbf{o}_{\text{time}}^{(2)}] W_O \quad [(1 \times d)(d \times d) = 1 \times d]$$

Figure 26 shows both steps with the numbers.



$$\text{row time} \cdot \text{column 3 of } W_O = (0.768)(0) + (0.955)(0) + (0.673)(1) + (0.164)(0) = \mathbf{0.673}$$

Figure 26: Stitching the heads. Top: head 1’s note and head 2’s note are placed side by side (head 1’s part shaded). Bottom: the stitched arrow times W_O . Our toy W_O is the identity, so each cell passes straight through; a trained W_O would mix them.

In the toy we set W_O to the identity matrix, so you can see head 1’s note in the first two numbers and head 2’s in the last two. A trained W_O mixes them, so that what one head found can be combined with what the other found.

WHY IT HAD TO BE THIS WAY. Concatenation keeps the heads’ answers apart, which was the point of §5.1. W_O then brings them back into the shared d -dimensional space every later step uses.

In one sentence: Head outputs are placed side by side and mixed back to width d by one learned matrix.

5.4 The MLP: each token thinks alone

After attention, **time** has heard from **waste** and **of**. What does it do with what it heard?

Predict first. Before reading on, tell me: should this next step look at the other tokens again, or work on each token alone? Why?

PICTURE IT. After a meeting, each person goes back to their own desk with their notes and checks them against a personal list of patterns: “if I see this, then write that”. Nobody talks during this part.

SAY IT EXACTLY. The MLP (multi-layer perceptron, also called the feed-forward network) applies to every token’s arrow separately:

$$\text{MLP}(\mathbf{h}) = \max(0, \mathbf{h}W_1 + \mathbf{b}_1) W_2 \quad [(1 \times d)(d \times d_{\text{ff}})(d_{\text{ff}} \times d) = 1 \times d]$$

Read it as a list of d_{ff} patterns. Column j of W_1 is a *key*: a direction to check for. The dot product of $\mathbf{h}$ with it, plus a threshold b_j , says how strongly the pattern is present. The function $\max(0, \cdot)$, called ReLU, lets a pattern *fire* only above its threshold. Row j of W_2 is the matching *value*: what to write when the pattern fires. Geva and colleagues (2021) showed that trained MLPs behave very much like this key–value memory.

Our toy has $d_{\text{ff}} = 16$, four times d , as in the original transformer and GPT-2. To keep the arithmetic on paper, only three neurons are switched on; the other thirteen are all zeros.

neuron	key (column of W_1)	threshold	value (row of W_2)
1	(1, 0, −1, 0)	0.0	(−4, −1, −2, −1)
2	(0, 1, 0, 0)	−1.0	(−1, −4, 0, 2)
3	(−1, 0, 1, 1)	−0.5	(0, 0, −4, −1)

The MLP reads $\mathbf{h}_1$, the token’s arrow after attention has been added in (§5.5 explains the adding). For our three tokens it is

$$H_1 = \begin{matrix} & \text{waste} \\ & \text{of} \\ \text{time} & \end{matrix} \begin{bmatrix} 2.000 & 1.000 & 1.000 & 0.000 \\ 0.500 & 1.500 & 0.500 & 0.000 \\ 0.768 & 0.955 & 1.673 & 1.164 \end{bmatrix}.$$

Now follow the three steps of the formula in pictures. First, every row of H_1 is checked against every key (Figure 27).

$$\begin{array}{c}
 H_1 \\
 \begin{array}{c|c|c|c}
 \text{waste} & 2.000 & 1.000 & 1.000 & 0.000 \\
 \text{of} & 0.500 & 1.500 & 0.500 & 0.000 \\
 \text{time} & 0.768 & 0.955 & 1.673 & 1.164
 \end{array} \\
 3 \times 4
 \end{array}
 \times
 \begin{array}{c}
 W_1 \text{ (neurons 1-3)} \\
 \begin{array}{c|c|c}
 \text{n1} & \text{n2} & \text{n3} \\
 1 & 0 & -1 \\
 0 & 1 & 0 \\
 -1 & 0 & 1 \\
 0 & 0 & 1
 \end{array} \\
 4 \times 3
 \end{array}
 =
 \begin{array}{c}
 H_1 W_1 \\
 \begin{array}{c|c|c}
 \text{n1} & \text{n2} & \text{n3} \\
 \text{waste} & 1.000 & 1.000 & -1.000 \\
 \text{of} & 0.000 & 1.500 & 0.000 \\
 \text{time} & -0.905 & 0.955 & 2.069
 \end{array} \\
 3 \times 3
 \end{array}$$

$$\text{row time} \cdot \text{column n3} = (0.768)(-1) + (0.955)(0) + (1.673)(1) + (1.164)(1) = \mathbf{2.069}$$

Figure 27: Step 1, the checks. Each column of W_1 is one neuron's key. The outlined row of H_1 (**time**) meets neuron 3's key and gives the dark cell. Only the three neurons that are switched on are drawn; the other thirteen columns are zeros.

Second, add each neuron's threshold and let ReLU switch off everything negative (Figure 28).

$$\begin{array}{c}
 H_1 W_1 \\
 \begin{array}{c|c|c}
 \text{n1} & \text{n2} & \text{n3} \\
 \text{waste} & 1.000 & 1.000 & -1.000 \\
 \text{of} & 0.000 & 1.500 & 0.000 \\
 \text{time} & -0.905 & 0.955 & 2.069
 \end{array} \\
 + \\
 \begin{array}{c|c|c}
 \text{thresholds } b_1 \\
 \text{n1} & \text{n2} & \text{n3} \\
 0.0 & -1.0 & -0.5
 \end{array} \\
 = \\
 \begin{array}{c}
 \text{before ReLU} \\
 \begin{array}{c|c|c}
 \text{n1} & \text{n2} & \text{n3} \\
 \text{waste} & 1.000 & 0.000 & -1.500 \\
 \text{of} & 0.000 & 0.500 & -0.500 \\
 \text{time} & -0.905 & -0.045 & 1.569
 \end{array} \\
 \xrightarrow{\max(0, \cdot)} \\
 \begin{array}{c}
 \text{after ReLU: negatives become 0} \\
 \begin{array}{c|c|c}
 \text{n1} & \text{n2} & \text{n3} \\
 \text{waste} & 1.000 & 0.000 & 0.000 \\
 \text{of} & 0.000 & 0.500 & 0.000 \\
 \text{time} & 0.000 & 0.000 & 1.569
 \end{array}
 \end{array}
 \end{array}$$

Figure 28: Step 2, thresholds and ReLU. Top: the thresholds are added to every row. Bottom: every negative number becomes 0. What survives sits on the diagonal (dark cells): neuron 1 fires on **waste**, neuron 2 on **of**, neuron 3 on **time**.

Third, each firing neuron writes its value, scaled by how strongly it fired (Figure 29).

$$\begin{array}{c}
 \text{after ReLU} \\
 \begin{array}{c|c|c}
 \text{n1} & \text{n2} & \text{n3} \\
 \text{waste} & 1.000 & 0.000 & 0.000 \\
 \text{of} & 0.000 & 0.500 & 0.000 \\
 \text{time} & 0.000 & 0.000 & 1.569
 \end{array} \\
 3 \times 3
 \end{array}
 \times
 \begin{array}{c}
 W_2 \text{ (rows 1-3)} \\
 \begin{array}{c|c|c|c}
 \text{n1} & -4 & -1 & -2 & -1 \\
 \text{n2} & -1 & -4 & 0 & 2 \\
 \text{n3} & 0 & 0 & -4 & -1
 \end{array} \\
 3 \times 4
 \end{array}
 =
 \begin{array}{c}
 \text{MLP output} \\
 \begin{array}{c|c|c|c}
 \text{waste} & -4.000 & -1.000 & -2.000 & -1.000 \\
 \text{of} & -0.500 & -2.000 & 0.000 & 1.000 \\
 \text{time} & 0.000 & 0.000 & -6.274 & -1.569
 \end{array} \\
 3 \times 4
 \end{array}$$

$$\text{row time} \cdot \text{column 3} = (0.000)(-2) + (0.000)(0) + (1.569)(-4) = \mathbf{-6.274}$$

Figure 29: Step 3, the values. The firing strengths times the rows of W_2 . For **time** only neuron 3 is non-zero, so its row of the output is 1.569 times neuron 3's value.

For **time**, the MLP writes $\text{MLP}(\mathbf{h}_{1,\text{time}}) = (0.000, 0.000, -6.274, -1.569)$. Look at that value arrow: it mostly *subtracts*. The arrow of **time** is dominated by **time**’s own coordinates 3 and 4; neuron 3 removes them, and what is left is mostly the part attention brought in from **waste**. Writing by erasing is common: a pattern can say “less of this” as readily as “more of that”.

WHERE THIS PICTURE BREAKS. A checklist suggests each neuron has one clean meaning. In trained models, one neuron often responds to several unrelated patterns, and one pattern is often spread across many neurons. The key–value reading is a good first picture, not a complete one.

WHY IT HAD TO BE THIS WAY. Why work on each token alone? Attention has already done the talking: the arrow of **time** now carries what **waste** and **of** said. Thinking per token is cheap and fully parallel, because every row is handled the same way at the same time. Why make the middle layer wider, $4d$? A wider middle means more patterns to check. In the sessions the toy model went from 64 numbers to 192 and back, and the reason is the same: room for more keys. Why ReLU? It is the simplest function that can switch a pattern off. GPT-2 uses a smooth cousin, GELU (Hendrycks and Gimpel, 2016), and Qwen uses a gated version, SwiGLU (Shazeer, 2020), which multiplies two keys’ answers together; Chapter 10 shows it on the real model.

A tempting wrong idea (M5.3). *The MLP is where tokens exchange information.* It comes right after attention, and it holds most of the block’s weights, so it is tempting to think it does the mixing. It does not: every row is processed alone. In the sessions the question came up as “why doesn’t your ffn looks back to other tokens?” The answer is that it does not need to, because attention already carried the other tokens’ information into this token’s arrow.

Check yourself 5.3. In the toy, change the arrow of **waste** to anything you like, but leave $\mathbf{h}_1$ of **time** unchanged. What happens to the MLP’s output for **time**?

- (a) It changes, because the MLP reads all three tokens.
- (b) It stays the same, because the MLP reads only the row it is working on.
- (c) It becomes zero, because **waste**’s neuron stops firing.
- (d) It depends on the attention shares of **waste**.

In one sentence: The MLP checks each token’s arrow, on its own, against a list of learned patterns and writes what the firing patterns say.

5.5 The residual stream: add, never overwrite

There is one more piece, and it is the simplest. After attention, we did not *replace* the arrow of **time** with $\mathbf{o}_{\text{time}}$. We *added*:

$$\mathbf{h}_1 = \mathbf{x} + \text{Attn}(\mathbf{x}), \quad \mathbf{h}_2 = \mathbf{h}_1 + \text{MLP}(\mathbf{h}_1).$$

PICTURE IT. A notebook is passed down a row of desks. Each desk reads the page and adds a note in the margin. No desk tears the page out. Whoever reads the notebook at the end sees the original text and every note.

SAY IT EXACTLY. For **time**, the two additions are entry by entry (Figure 30).

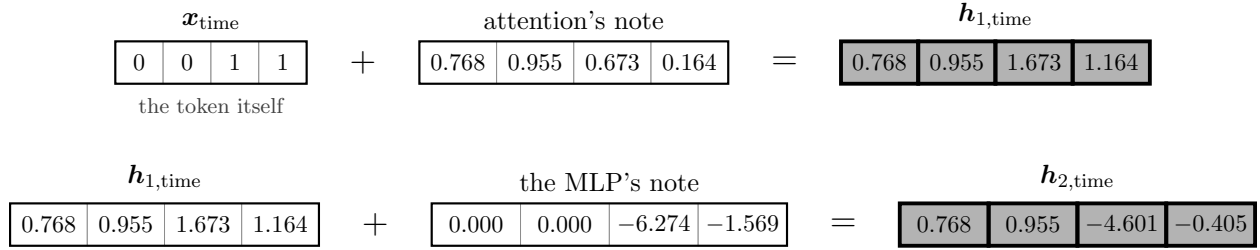


Figure 30: The residual stream for **time**. Top: the token's own arrow plus attention's note gives $\mathbf{h}_1$. Bottom: $\mathbf{h}_1$ plus the MLP's note gives $\mathbf{h}_2$. Nothing is replaced; each layer adds.

For the whole sentence, $H_2 = \begin{bmatrix} -2.000 & 0.000 & -1.000 & -1.000 \\ 0.000 & -0.500 & 0.500 & 1.000 \\ 0.768 & 0.955 & -4.601 & -0.405 \end{bmatrix}$.

The same block runs on every sentence of a batch at once, with the same weights. Figure 31 stacks H_1 for our two customer replies, the $B \times T \times d$ block of Chapter 3, §3.9.

		0.0	1.0	1.0	2.0	← not very useful
waste	2.0	1.0	1.0	0.0	0.5	
of	0.5	1.5	0.5	0.0	0.7	
time	0.8	1.0	1.7	1.2		← waste of time

$H_1: B \times T \times d = 2 \times 3 \times 4$

Figure 31: H_1 for a batch of two sentences (rounded to one decimal). The front sheet is **waste of time**; the sheet behind is **not very useful**. Every number in both sheets was made by the same heads and the same W_O .

Now read each row's guess for the next token, using the method Chapter 7 develops: compare the row with every token's arrow by dot product and pick the largest. Row 1 picks **of**, row 2 picks **time**, row 3 picks **Sorry** (score 1.72, ahead of **of** at 0.95). One pass, three positions, three correct next tokens: the $B \times T$ predictions of §1.9 on a real, if tiny, example.

Predict first. What happens if we remove the residual additions, so that $\mathbf{h}_1 = \text{Attn}(\mathbf{x})$ and $\mathbf{h}_2 = \text{MLP}(\mathbf{h}_1)$? Will the guesses survive?

Sanity check. They do not. Without the additions the three rows guess **of**, **waste**, **of**. Without the MLP but with the additions, **time** guesses .. Every piece is needed.

Deeper stacks. A real model repeats the block many times. To see what the additions do over depth, we stacked untrained blocks of a realistic shape ($d = 64$, 4 heads, $d_{\text{ff}} = 256$), fed 32 random sentences of 16 tokens from a vocabulary of 100, and asked after each layer: can we still tell which token each position started as, by finding the nearest token arrow?

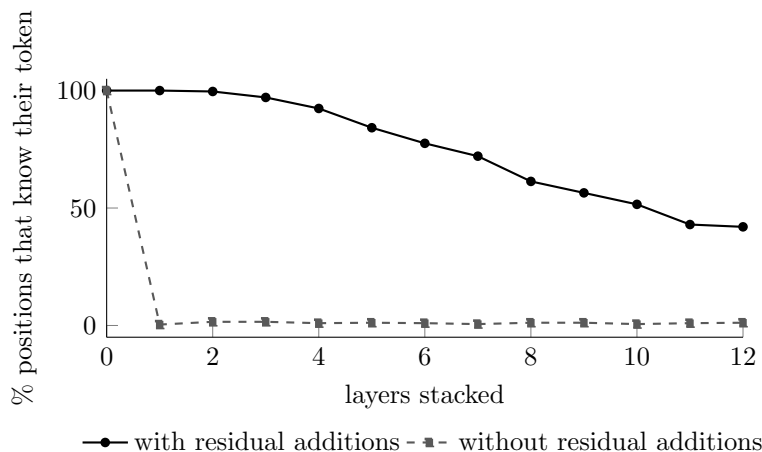


Figure 32: Untrained random blocks, fixed seed. With the additions, 92.4% of positions still know their token after 4 layers and 42.0% after 12. Without them, 0.4% after one layer: chance level is 1.0%.

Without the additions, one layer is enough to lose almost every token’s identity. In the sessions this was described as the model that “forgot the original in representation without combining the effects of it”.

WHERE THIS PICTURE BREAKS. A notebook is only written on, but the residual stream can also be *corrected*: neuron 3 wrote negative numbers to cancel part of what was there. Think of margin notes that can say “ignore the line above”.

WHY IT HAD TO BE THIS WAY. Adding keeps a straight path from the input to the output, so the original token and every layer’s contribution stay readable at the end. Elhage and colleagues (2021) call this shared space the *residual stream*: every layer reads from it and writes into it. The idea of adding a layer’s input to its output comes from image networks, where He and colleagues (2016) used it to train networks with more than a hundred layers.

A tempting wrong idea (M5.4). *Each layer replaces a token’s arrow with a new one.* Diagrams draw layers as boxes in a row, which suggests a relay where each runner drops what came before. The figure above shows what that would cost. Each layer *adds* to the arrow instead.

Check yourself 5.4. In the toy, $\mathbf{x}_{\text{time}} = (0, 0, 1, 1)$ and attention’s output is $(0.768, 0.955, 0.673, 0.164)$. What is $\mathbf{h}_1$ for time?

- (a) $(0.768, 0.955, 0.673, 0.164)$, because attention replaces the arrow.
- (b) $(0.768, 0.955, 1.673, 1.164)$, because the two are added.
- (c) $(0, 0, 1, 1)$, because attention only changes other tokens.
- (d) The average of the two arrows.

Abstraction — add, do not overwrite. Systems that must stay correct over many steps keep the original and append changes: a bank keeps every transaction rather than only the balance; a version-control history keeps every commit. An agent’s memory layer that rewrites a stored memory in place loses the evidence of what was first said; one that keeps the raw

episode and adds annotations can always be checked and corrected later. The residual stream makes the same choice inside the model.

In one sentence: Every layer adds its contribution to the token's arrow, so the original and every layer's notes stay readable all the way to the top.

5.6 The whole block, and where its weights live

Putting it together, one block (still without the normalisation of Chapter 6) is

$$\mathbf{h}_1 = \mathbf{x} + [\text{head}_1(\mathbf{x}) \quad \text{head}_2(\mathbf{x})] W_O, \quad \mathbf{h}_2 = \mathbf{h}_1 + \max(0, \mathbf{h}_1 W_1 + \mathbf{b}_1) W_2.$$

Count the weights. Attention holds four $d \times d$ matrices' worth (W_Q, W_K, W_V split across the heads, plus W_O): $4d^2$. The MLP holds $d \times 4d$ plus $4d \times d$: $8d^2$. So about two thirds of a block's weights sit in the MLP. In the toy: 64 in attention and 144 in the MLP (including its 16 thresholds).

In one sentence: A block is talk (heads), think (MLP) and add (residual), and most of its weights are in the thinking.

Looking back. (1) In Chapter 3, why was it $QK^\top$ and not QK ? (2) In Chapter 4, which of $\mathbf{q}, \mathbf{k}, \mathbf{v}$ does RoPE rotate, and which does it leave alone? (3) In Chapter 1, how many next-token predictions does one pass make for a batch of B sentences of T tokens?

Exercise 5.1. Compute head 2's shares and output for the token `of` (row 2). Explain the shares using the query alone.

Exercise 5.2. GPT-2 small has $d = 768$ and $d_{\text{ff}} = 3072$. Ignoring biases, how many weights are in one block's attention, in its MLP, and in total?

Exercise 5.3. Using H_1 from §5.4, design a fourth neuron (a key with entries from $\{-1, 0, 1\}$ and a threshold) that fires on `of` and on no other row. Is it unique?

Exercise 5.4. In your own words, explain to someone who has never seen a transformer why stacking twelve blocks without the residual additions would fail, using the figure in §5.5.

Answers to Chapter 5

Check yourself 5.1. (b). The output is one weighted average; different pairs of messages can give the same average (§5.1). (a) is M5.2: one head cannot keep two messages apart. (c) goes too far: the average is still information. (d) is wrong: `time`'s own share here is 0.

Check yourself 5.2. (b). $64/8 = 8$ numbers per head, and 8 separate softmaxes instead of 2. (a) is backwards. (c) is wrong: the total size of W_Q, W_K, W_V, W_O stays $4d^2$. (d) is M5.1.

Check yourself 5.3. (b). The MLP reads one row at a time. (a) and (d) are M5.3: the other tokens reach `time` only through attention, which happened before. (c) mixes up two rows: neuron 1 firing on `waste` does not affect `time`'s row.

Check yourself 5.4. (b). The residual addition: $(0, 0, 1, 1) + (0.768, 0.955, 0.673, 0.164)$. (a) is M5.4. (c) is wrong: attention writes into every token's arrow, including `time`'s. (d) halves both, which no step does.

Answer 5.2 (head 2, your turn). $\mathbf{q}_{\text{time}}^{(2)} = (1, 0)$. The keys are the columns of $K^{(2)\top}$ below, one per token; only `of`'s is non-zero. Query against keys:

$$\begin{array}{c}
 \begin{array}{c} q_{\text{time}}^{(2)} \\ \text{time} \end{array} \begin{array}{|c|c|} \hline 1 & 0 \\ \hline \end{array} \\
 1 \times 2
 \end{array}
 \times
 \begin{array}{c}
 K^{(2)\top} \\
 \begin{array}{c} \text{waste} \quad \text{of} \quad \text{time} \end{array} \\
 \begin{array}{|c|c|c|} \hline 0 & 2 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \\
 2 \times 3
 \end{array}
 =
 \begin{array}{c}
 \begin{array}{c} \text{raw scores} \\ \text{waste} \quad \text{of} \quad \text{time} \end{array} \\
 \begin{array}{|c|c|c|} \hline 0 & 2 & 0 \\ \hline \end{array} \\
 1 \times 3
 \end{array}$$

row **time** · column **of** = (1)(2) + (0)(0) = **2**

Raw scores (0, 2, 0); scaled (0.000, 1.414, 0.000); shares (0.164, 0.673, 0.164).

Looking back. (1) Q and K are both $T \times d_k$; only $QK^\top$ pairs each query row with each key row. (2) RoPE rotates queries and keys; values are left alone. (3) $B \times T$.

Exercise 5.1. $q_{\text{of}}^{(2)} = (0, 0)$: a query of all zeros scores every key 0, so the shares are (0.500, 0.500) over **waste** and **of**, and the output is (0.500, 0.000). An empty question gives the plain average, as in Chapter 3.

Exercise 5.2. Attention $4d^2 = 2,359,296$; MLP $2 \times d \times d_{\text{ff}} = 4,718,592$; total 7,077,888. Chapter 10 checks this against the real model, biases and norms included.

Exercise 5.3. Neuron 2, key (0, 1, 0, 0) with threshold -1 , already works: it gives 0.5 on **of** and nothing elsewhere. Others work too, for example $(-1, 1, 0, 0)$ with threshold 0. It is not unique: many keys separate the rows, which is one reason trained neurons are hard to name.

Exercise 5.4. Without the additions, each layer's output is all that reaches the next layer. After one untrained layer only 0.4% of positions can still be matched to their starting token, roughly chance; with the additions, the original arrow travels up unchanged and later layers add to it.

What you now know

- Why one head returns one blend, and why several narrower heads can ask several questions at the same cost.
- How head outputs are concatenated and mixed by W_O .
- How the MLP works on each token alone, as keys (patterns), thresholds and values.
- Why every layer adds to the residual stream instead of replacing it, with measured numbers.
- How one toy block predicts **of**, **time** and **Sorry** at the three positions in a single pass.

6 Normalisation, and why depth is hard

Map of this chapter. *The problem:* real models stack dozens of blocks, and every block adds to the residual stream. What happens to the size of the arrows after 48 additions? *The steps:* measure it on untrained blocks (§6.1); fix it by resizing every arrow before it is used, with RMSNorm or LayerNorm, and see what each does to an arrow geometrically (§6.2); put the norm into our toy block and check its predictions (§6.3); decide whether the norm goes before or after each layer, with measurements (§6.4). *Where it ends:* with the block modern models use, *pre-norm*, and a measured reason for choosing it.

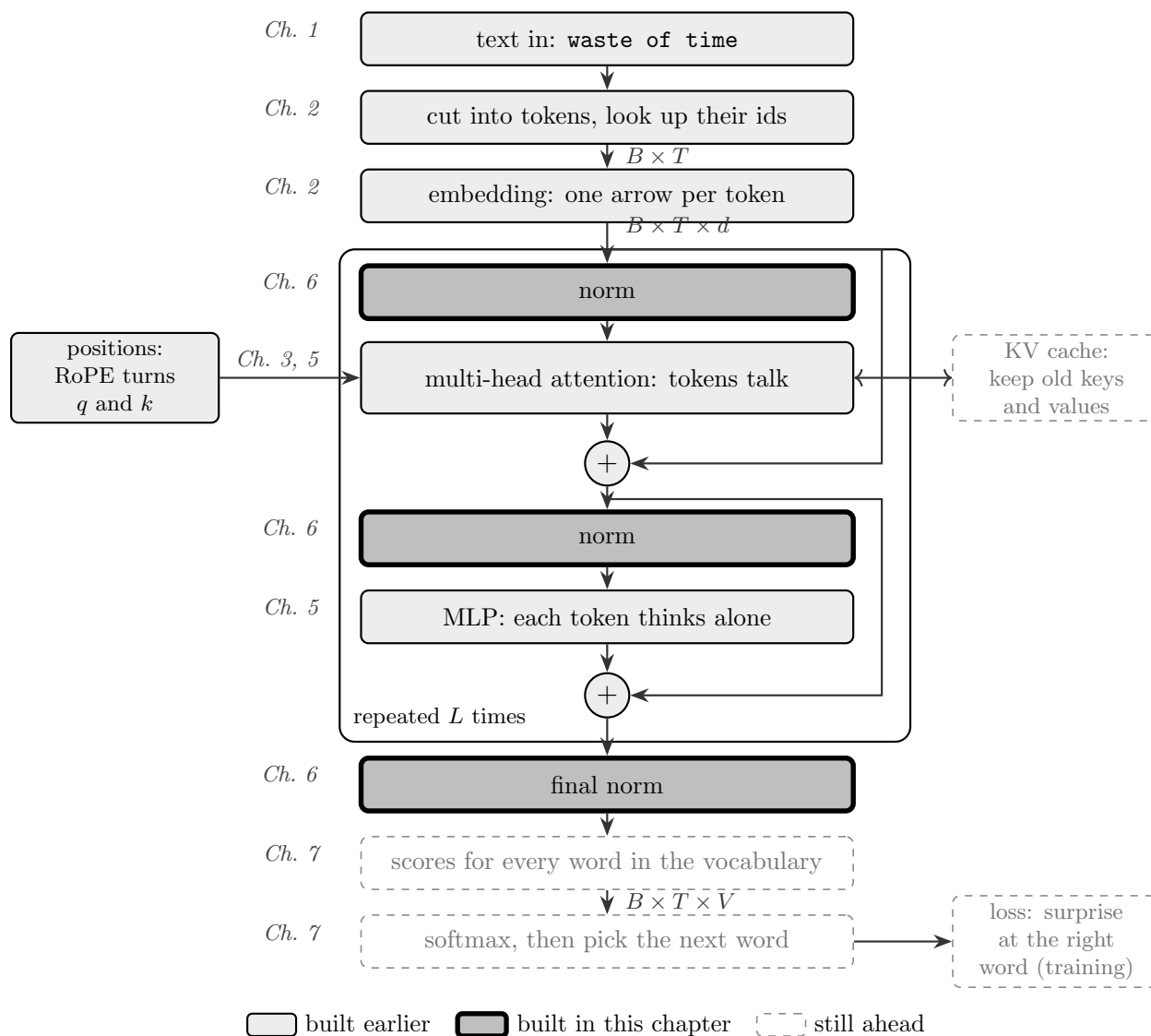


Figure 33: Where we are in the machine. You are here: Chapter 6 adds the *norm* boxes. One sits in front of attention, one in front of the MLP, and one after the last layer. Each resizes a token’s arrow before it is read; none of them touches the residual stream itself.

Before you start. Five quick questions. Answer each, then read the refresher under it.

1. What is the average of the numbers 1, 0, 1, 0?

Refresher: add and divide by how many: $2/4 = 0.50$.

2. Square each of 1, 0, 1, 0, take the average, then the square root. What do you get?

Refresher: $\sqrt{(1+0+1+0)/4} = \sqrt{0.5} \approx 0.707$. This is the *root mean square* (RMS): a typical size of the numbers, ignoring their signs.

3. How long is the arrow (1, 0, 1, 0)?

Refresher: square the entries, add, take the square root: $\sqrt{2} \approx 1.414$ (Chapter 3, §3.2).

4. If you multiply the arrow (1, 0, 1, 0) by 10, does its direction change?

Refresher: no. Every entry grows by the same factor, so the arrow points the same way and is 10 times longer.

5. In Chapter 1, what was left of a signal multiplied by 1.1 at each of 50 steps?

Refresher: $1.1^{50} \approx 117$. Small growth, repeated, becomes very large.

6.1 What goes wrong with depth

Predict first. Stack 48 untrained blocks of the kind built in Chapter 5, with the residual additions but nothing else. The arrows start with a typical size (RMS) of about 1. Tell me your guess for their size after 6 layers, after 12, after 48.

PICTURE IT. A microphone too close to its own speaker. Each pass around the loop adds a little, and the little is proportional to what is already there, so a whisper becomes a howl.

SAY IT EXACTLY. We ran exactly this: blocks with $d = 64$, 4 heads, weights drawn at the usual starting scale, a fixed seed, and 32 random sentences. The typical size of the residual stream after L layers:

layers L	1	6	12	24	48
residual additions, no norm	1.4	25.7	789	2.63×10^5	1.14×10^{11}
no additions, no norm	0.402	0.0798	0.0137	8.81×10^{-5}	2.17×10^{-8}

With the additions, each block's update is roughly proportional to what it reads, so the stream is multiplied by a little more than 1 every layer, and after 48 layers its typical size is 1.14×10^{11} . Without the additions, each layer shrinks the signal and it fades to almost nothing. This is the same repeated multiplication that made the RNN of Chapter 1 fade or explode, now happening across layers instead of across words.

WHERE THIS PICTURE BREAKS. The microphone keeps getting louder for ever. In a model, the growth also changes *behaviour*: softmax and ReLU respond differently to large and small inputs, so a block that worked at size 1 does something else at size 1000.

WHY IT HAD TO BE THIS WAY. Chapter 3 gave a design rule: keep the size of every signal independent of the size of the model. Depth is one more kind of size. Each block was built expecting arrows of a certain size; if the size drifts with depth, every block sees inputs it was not built for.

A tempting wrong idea (M6.4). *More layers always means a better model.* Without something to control size, more layers mean arrows that explode or fade, and a model that cannot be trained. Depth is useful only once this is fixed, and even then it is a trade-off against width and cost (Chapter 11).

In one sentence: Stacking blocks multiplies the size of the arrows layer after layer, so without control they explode or fade.

6.2 Normalise: resize every arrow, keep its direction

Predict first. A block wants its input at a standard size. What is the simplest change to an arrow that fixes its size but keeps what it points at?

PICTURE IT. Photos of different sizes are all resized to fit the same frame before they go in the album. The picture in each photo stays the same; only its size changes.

SAY IT EXACTLY. **RMSNorm** (Zhang and Sennrich, 2019) divides an arrow by its root mean square:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d} \sum_i x_i^2}, \quad \text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \quad [1 \times d \rightarrow 1 \times d]$$

For **waste**, $\mathbf{x} = (1, 0, 1, 0)$ has $\text{RMS} = 0.707$, so RMSNorm gives $(1.414, 0.000, 1.414, 0.000)$. Same direction; new length $\sqrt{d} = 2$. Every arrow that leaves RMSNorm has length exactly $\sqrt{d}$: it lands on a sphere of that radius. Multiplying the input by 10 changes nothing in the output, which our script checks. Here is every step, on the vector itself:

input $\mathbf{x}$	1	0	1	0	waste
square each entry	1	0	1	0	average of these = 0.500
$\mathbf{x} / \text{RMS}(\mathbf{x})$	1.41	0.00	1.41	0.00	$\text{RMS} = \sqrt{0.500} = 0.707$; divide every entry
times gain $\mathbf{g}$	1	1	1	1	the toy's gain: all 1 (learned in real models)
output	1.41	0.00	1.41	0.00	length = 2.000 = $\sqrt{d}$

Figure 34: RMSNorm on **waste**, one step per line. Read down: square, average, square root, divide, multiply by the gain. The direction never changes; only the length does.

LayerNorm (Ba, Kiros and Hinton, 2016) does one more thing first. It subtracts the average of the entries, then divides by their spread:

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma}, \quad \mu = \frac{1}{d} \sum_i x_i, \quad \sigma = \sqrt{\frac{1}{d} \sum_i (x_i - \mu)^2}.$$

For **waste**: $\mu = 0.50$, and the result is $(1.000, -1.000, 1.000, -1.000)$, also of length 2:

input $\mathbf{x}$	1	0	1	0	waste
minus the mean μ	0.50	-0.50	0.50	-0.50	$\mu = 0.500$, subtracted from every entry
divide by σ	1	-1	1	-1	$\sigma = 0.500$, the spread of the line above
times gain $\mathbf{g}$	1	1	1	1	all 1 in the toy
output	1	-1	1	-1	entries add to 0; length = 2.000

Figure 35: LayerNorm on **waste**. The extra step is the second line: the average is taken away from every entry, so the output's entries always add to zero.

Geometrically, subtracting the average moves the arrow onto the flat slice of space where the entries add to zero; dividing then puts it on the sphere. So LayerNorm ignores any amount added equally to every entry: $(4, 3, 4, 3)$ gives the same $(1.000, -1.000, 1.000, -1.000)$, while RMSNorm turns it into $(1.131, 0.849, 1.131, 0.849)$.

A whole sentence goes through the norm at once, as a $T \times d$ matrix, and every row is resized with its own RMS. Look at the middle column: **waste** and **time** share an RMS, **of** has a smaller one, and each row is divided only by its own number.

	X					RMS		$\text{RMSNorm}(X)$
waste	1	0	1	0		0.71		1.41
of	0	1	0	0	$\div$	0.50	$=$	0.00
time	0	0	1	1		0.71		0.00
	$T \times d = 3 \times 4$					3×1		3×4

Figure 36: RMSNorm on the sentence matrix. The highlighted row, **time**, is divided by its own RMS, 0.707; no row borrows another row's number. The LayerNorm of each row follows the same rule with mean and spread.

Real models then multiply each coordinate by a learned *gain* (and LayerNorm adds a learned shift), so the model can choose the size it wants per coordinate. In our toy every gain is 1.

WHERE THIS PICTURE BREAKS. Resizing photos treats each photo on its own, and so does the norm: it rescales one token's arrow using only that token's numbers. It does not make tokens equal to each other in content, and it does not compare them.

The same holds for a batch. Two customer replies stacked as a $B \times T \times d$ block go in; the same block shape comes out, and every one of the six rows has been resized on its own:

		0	1	0	1	← sentence 2
waste	1	0	1	0	0	
of	0	1	0	0	0	
time	0	0	1	1		← sentence 1

$X: B \times T \times d = 2 \times 3 \times 4$

⇓ each of the six rows divided by its own RMS

		0.00	1.41	0.00	1.41	← sentence 2
waste	1.41	0.00	1.41	0.00	0.00	
of	0.00	2.00	0.00	0.00	0.00	
time	0.00	0.00	1.41	1.41		← sentence 1

$\text{RMSNorm}(X): B \times T \times d = 2 \times 3 \times 4$

Figure 37: RMSNorm on a batch of $B = 2$ sentences. Shape in, shape out: $2 \times 3 \times 4$. No number moves between rows, and none moves between sentences.

WHY IT HAD TO BE THIS WAY. The norm removes the one thing depth messes up, the size, and keeps the thing the model needs, the direction. Chapter 3 showed that a dot product has two knobs, direction and length; normalisation fixes the length knob of the input so that what a block computes depends on direction alone. RMSNorm skips the centring step, which costs a little computation and, in Zhang and Sennrich’s experiments, lost nothing; Llama and Qwen use it.

A tempting wrong idea (M6.1). *Normalisation is there so that no token can beat another.* In the sessions this came up as “to remove comparison bias so that nobody beats each other”. It is tempting because the word sounds like making things fair. But the norm acts on each token alone and keeps its direction, so tokens still differ as much as before. What it controls is the size of each arrow as it goes through many layers.

A tempting wrong idea (M6.2). *Normalising throws away the information in an arrow.* Only the length is removed (and, for LayerNorm, the average entry). The direction, which is where the meaning lives, is untouched.

A tempting wrong idea (M6.3). *LayerNorm makes every arrow length 1.* It makes the entries’ spread 1, so the length is $\sqrt{d}$: 2 in our toy, about 32 for $d = 1024$. “Unit length” is a different, related idea.

Check yourself 6.1. What does RMSNorm do to the arrow $(2, 0, 2, 0)$?

- (a) It returns $(1.414, 0.000, 1.414, 0.000)$, the same as for $(1, 0, 1, 0)$.
- (b) It returns $(1, 0, 1, 0)$, an arrow of length 1.
- (c) It returns $(1, -1, 1, -1)$, after removing the average.
- (d) It makes it equal to the RMSNorm of **time**, so the two tokens can no longer be told apart.

In one sentence: A norm rescales each token’s arrow to a fixed length, $\sqrt{d}$, and keeps its direction.

6.3 The toy block with a norm

Where does the norm go? In modern models, in front of each layer’s input, and once more before the output:

$$\mathbf{h}_1 = \mathbf{x} + \text{Attn}(\text{Norm}(\mathbf{x})), \quad \mathbf{h}_2 = \mathbf{h}_1 + \text{MLP}(\text{Norm}(\mathbf{h}_1)), \quad \text{output reads } \text{Norm}(\mathbf{h}_2).$$

Notice that the norm sits on the side road into each layer, not on the residual stream itself. The stream is still only ever added to.

Predict first. With RMSNorm in front, the attention inputs have length 2 instead of about 1.4. Will the heads become more decisive or less? Recall the length knob of Chapter 3, §3.7.

Sanity check. More decisive. **waste** enters attention as (1.414, 0.000, 1.414, 0.000). Head 1’s shares for **time** move from (0.768, 0.045, 0.187) without the norm to (0.941, 0.003, 0.056) with it; head 2’s from (0.164, 0.673, 0.164) to (0.018, 0.965, 0.018). Longer arrows, sharper softmax.

The three MLP neurons still fire on their own rows only, now with strengths (0.952, 0.625, 0.934) instead of (1.000, 0.500, 1.569), and the toy still predicts **of**, **time**, **Sorry** at the three positions. Chapter 7 starts from exactly this block.

What if we swap in LayerNorm, keeping every weight the same? The predictions become **not**, **time**, **Sorry**: the first position now prefers **not** (score 1.64) over **of** (1.46). Our hand-chosen weights were checked with RMSNorm, and LayerNorm sends the arrows to different places. A trained model learns its weights together with its norm; swapping the norm afterwards breaks it in the same way.

In one sentence: With pre-norm the toy keeps its three correct guesses and its heads become sharper, because every input now has the same length.

6.4 Before or after? A measured choice

There are two places to put the norm.

- **Post-norm** (the original transformer, Vaswani and colleagues, 2017): add, then normalise the stream itself: $\mathbf{x} \leftarrow \text{Norm}(\mathbf{x} + F(\mathbf{x}))$.
- **Pre-norm** (GPT-2 and most models since): normalise the copy that goes into the layer, add the result to the untouched stream: $\mathbf{x} \leftarrow \mathbf{x} + F(\text{Norm}(\mathbf{x}))$.

Predict first. Both keep sizes under control. Which one lets the top of a 48-layer stack still see the token that came in at the bottom?

PICTURE IT. Post-norm is a notebook that gets photocopied at a fixed size after every note is added. Each copy shrinks the older writing to make room, so after many copies the first page is a speck. Pre-norm gives each desk a resized copy to read, but every desk writes on the original notebook, which keeps all the old writing at full size.

SAY IT EXACTLY. We measured both on the same 48 untrained blocks. “Agreement with the input” is the cosine of the angle between each position’s arrow at layer L and its arrow at the start (1 means same direction, 0 means unrelated). “Knows its token” is the test from Chapter 5.

layers L	1	6	12	24	48
size (RMS), post-norm	1	1	1	1	1
size (RMS), pre-norm	1.35	2.59	3.85	5.4	8.1
agreement with input, post-norm	0.730	0.086	0.013	-0.013	0.009
agreement with input, pre-norm	0.751	0.388	0.268	0.196	0.128
knows its token, post-norm (%)	100.0	3.3	0.8	0.8	0.4
knows its token, pre-norm (%)	100.0	72.9	34.8	14.3	3.5

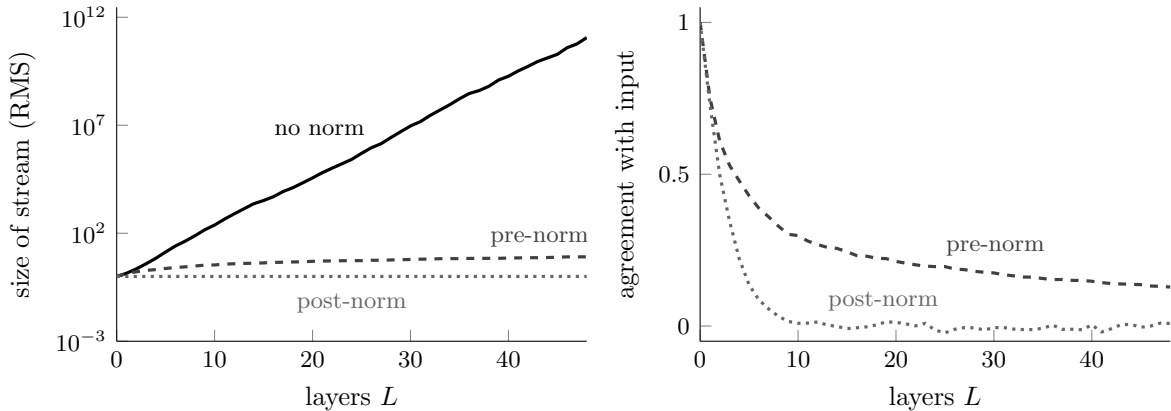


Figure 38: Forty-eight untrained blocks, fixed seed. Left: without a norm the size explodes; post-norm pins it at 1; pre-norm lets it grow slowly. Right: post-norm loses the input within a few layers; pre-norm keeps it visible much longer.

Two readings. First, post-norm holds the size at exactly 1 (with RMSNorm, the stream’s RMS is 1 by construction), but the input’s direction is gone by about 6 layers: each add-and-normalise shrinks the old content. If every update were as large as the stream and unrelated to it, one step would keep $1/\sqrt{2} \approx 0.707$ of the old content, and six steps would keep 0.125. Second, pre-norm lets the size grow, but slowly: from 12 to 48 layers it grows by a factor of 2.10, close to $\sqrt{48/12} = 2$, the coin-toss growth of Chapter 3. The input stays readable much longer because the stream itself is never rescaled.

WHERE THIS PICTURE BREAKS. Our measurement looks only at the forward pass of untrained blocks. The deeper reason for pre-norm is about learning: Xiong and colleagues (2020) showed that with post-norm the learning signals reaching the bottom layers are badly sized at the start of training, which is why the original transformer needed a slow “warm-up” of the learning rate. Appendix A gives that argument with derivatives.

WHY IT HAD TO BE THIS WAY. Pre-norm keeps a clean, untouched path from bottom to top (the residual stream) and puts the size control only where each layer reads. Information from the bottom reaches the top without being shrunk at every step, and, as Xiong and colleagues showed, so does the learning signal on its way down. That is why GPT-2, Llama and Qwen all use it, plus one final norm before the output, since the stream itself is never normalised.

A tempting wrong idea (M6.5). *Whether the norm sits before or after a layer is a detail.* Both versions compute the same kinds of things, so the placement looks cosmetic. The table shows a difference you can measure in the first few layers, and it decides whether a deep model can be trained without special care.

Check yourself 6.2. In a post-norm stack, why does the bottom token’s direction fade even though the size stays at exactly 1?

- (a) Because the norm throws away the direction.
- (b) Because each add-and-normalise step shrinks the old content to make room for the new.
- (c) Because the attention heads ignore the token itself.
- (d) Because size and direction are the same thing.

Abstraction — control what each stage reads, keep the record untouched. Pre-norm resizes the input of every layer but never edits the stream. A reliable pipeline of agents does the same: each step validates and normalises what it reads (a schema, a typed decision such as the yes/no and score questions from the memory-system sessions), while the raw record of what happened is kept as it was, so a later step can always go back to it. Normalising the record itself, step after step, slowly erases where it came from.

In one sentence: Pre-norm resizes each layer’s input and leaves the stream alone, which keeps sizes under control and keeps the bottom of a deep stack visible at the top.

Looking back. (1) In Chapter 3, why did we divide attention scores by $\sqrt{d_k}$, and which design rule did that follow? (2) In Chapter 5, what did the residual additions keep, according to the figure in §5.5? (3) In Chapter 1, what happened to an RNN’s memory when each step multiplied it by 0.9?

Exercise 6.1. Compute the RMSNorm and the LayerNorm of $(0, 0, 1, 1)$ by hand, and check both lengths.

Exercise 6.2. LayerNorm gives the same output for $(1, 0, 1, 0)$ and $(4, 3, 4, 3)$. Give one situation where ignoring a shift of every entry is helpful, and one where it could lose information.

Exercise 6.3. Assume each add-and-normalise step of post-norm keeps $1/\sqrt{2}$ of the old content. How much is left after 6 steps? Why is the measured agreement after 6 layers larger than this?

Exercise 6.4. In our pre-norm measurement the stream’s size grows roughly like the square root of the depth. Predict its size at 192 layers from the value at 48 layers.

Answers to Chapter 6

Check yourself 6.1. (a). RMSNorm ignores the overall scale: $(2, 0, 2, 0)$ is $(1, 0, 1, 0)$ doubled. (b) is M6.3: the length is $\sqrt{d} = 2$, not 1. (c) is LayerNorm, not RMSNorm. (d) is M6.1 and M6.2: **time** points elsewhere, and its direction survives the norm.

Check yourself 6.2. (b). Every step rescales the sum of old and new, so the old part’s share shrinks step by step. (a) is M6.2. (c) is unrelated to the norm. (d) confuses length with direction: the norm fixes one and keeps the other.

Looking back. (1) Random dot products grow like $\sqrt{d_k}$; dividing keeps the size independent of the model’s width. (2) The identity of each token: most positions still knew their token after 12 layers. (3) It faded: $0.9^{100} \approx 2.66 \times 10^{-5}$.

Exercise 6.1. $\text{RMS}(0, 0, 1, 1) = 0.707$, so RMSNorm gives $(0.000, 0.000, 1.414, 1.414)$. LayerNorm: $\mu = 0.50$, giving $(-1.000, -1.000, 1.000, 1.000)$. Both have length 2.

Exercise 6.2. Helpful: if some input adds the same offset to every entry for a reason unrelated to meaning, ignoring it removes noise. Losing: if the overall level of the entries carries meaning, LayerNorm erases it, while RMSNorm keeps it.

Exercise 6.3. $(1/\sqrt{2})^6 = 0.125$. Real updates are smaller than the stream and not fully unrelated to it, so each step shrinks the old content less; the measured value after 6 layers is 0.086.

Exercise 6.4. Four times the depth gives $\sqrt{4} = 2$ times the size: about 16.2, from 8.1 at 48 layers.

What you now know

- Why stacking blocks makes arrow sizes explode or fade, with measured numbers.
- What RMSNorm and LayerNorm do to an arrow: fix its length at $\sqrt{d}$, keep its direction; LayerNorm also removes the average.
- Where the norm sits in a modern block, and what it does to our toy's attention.
- Why pre-norm is preferred to post-norm, measured on 48 layers, and where the full argument lives (Appendix A).

7 From scores to words: temperature and sampling

Map of this chapter. *The problem:* after the last block, **time** holds one arrow of d numbers. The customer is waiting for a word. How does an arrow become a word? *The steps:* score every word in the vocabulary against the arrow (§7.1); turn the scores into probabilities (§7.2); decide how adventurous to be, with temperature (§7.3) and with greedy, top- k and top- p choices (§7.4); see what randomness and seeds do (§7.5); ask what the probabilities actually mean (§7.6); and see why one wrong word does not stay small (§7.7). *Where it ends:* with the loop of Chapter 1, Figure 1, running on our toy: the arrow of **time** becomes Sorry.

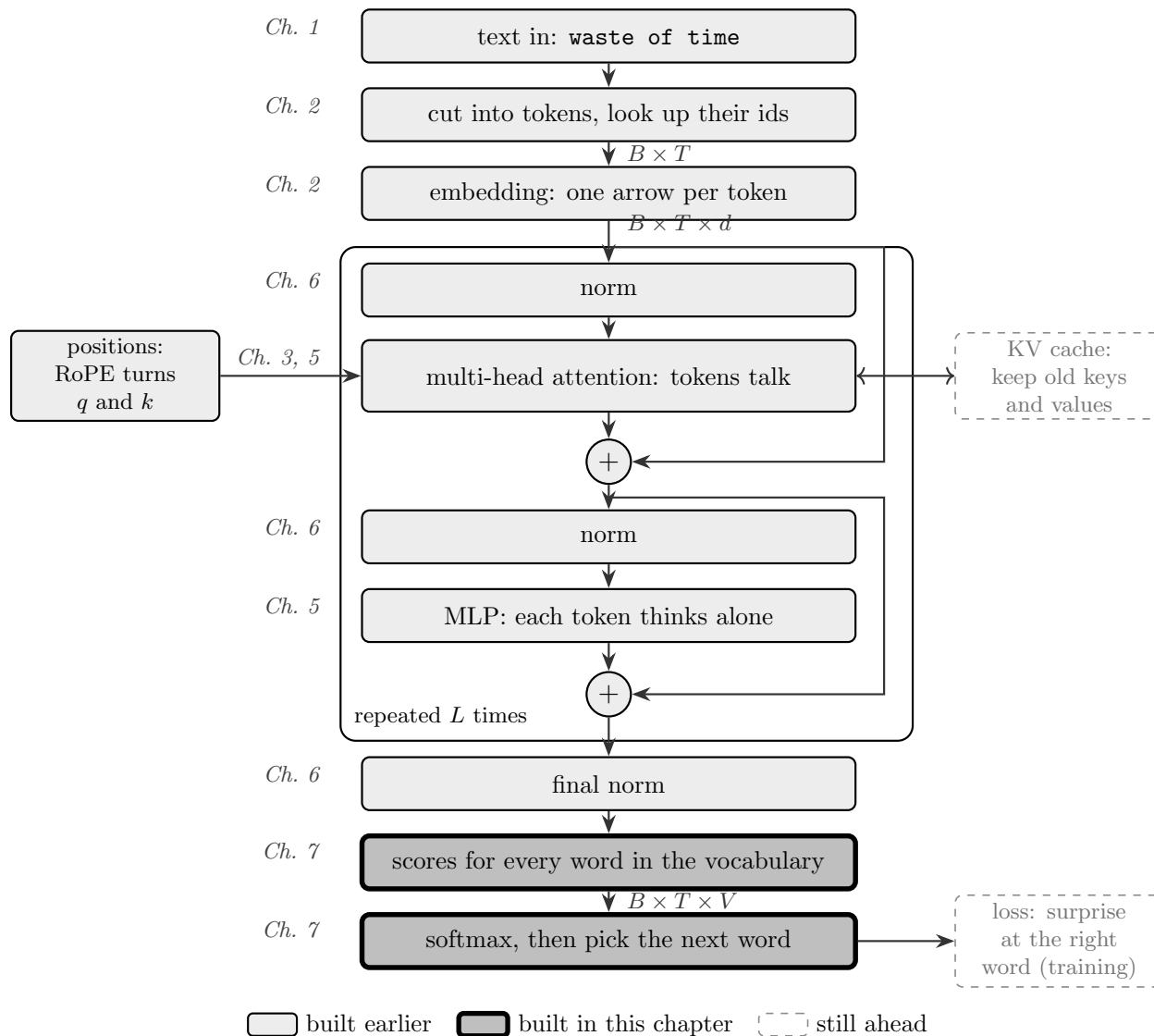


Figure 39: Where we are in the machine. You are here: Chapter 7 adds the last two boxes at the bottom. The arrow that leaves the final norm is scored against every word in the vocabulary, and softmax turns the scores into probabilities from which one word is picked.

Before you start. Five quick questions. Answer each, then read the refresher under it.

1. Compute the dot product of $(1, 1, 0, 0)$ and $(2, 3, 5, 7)$.
Refresher: multiply matching entries and add: $2 + 3 + 0 + 0 = 5$ (Chapter 3, §3.2).
2. Two arrows meet at 90° . What is their dot product?
Refresher: zero. The dot product is length times length times $\cos \theta$, and $\cos 90^\circ = 0$.
3. Scores are $(2, 0, 1)$. Write the softmax shares as fractions of $e^2 + e^0 + e^1$.
Refresher: share of each = $e^{\text{score}}/\text{total}$. Chapter 3, §3.6 computed these: $(0.665, 0.090, 0.245)$.
4. What is 0.95 multiplied by itself 7 times?
Refresher: $0.95^7 \approx 0.698$. Repeated multiplication by a number below 1 shrinks it.
5. If you halve every number in $(4, 0, 2)$, what happens to the gaps between them?
Refresher: they halve too: $(2, 0, 1)$ has gaps 2 and 1 instead of 4 and 2.

7.1 Score every word against the arrow

Chapters 5 and 6 left us with the arrow of **time** after one pre-norm block, and the final norm resizes it:

$$\mathbf{h} = (1.331, 1.410, -0.806, 0.091), \quad \mathbf{u} = \text{RMSNorm}(\mathbf{h}) = (1.267, 1.341, -0.767, 0.087),$$

an arrow of length $\|\mathbf{u}\| = 2.000$.

Predict first. Every word in our toy vocabulary has an arrow in the embedding table of Chapter 2. Which word's arrow do you expect $\mathbf{u}$ to agree with most? Look at its signs before you read on.

PICTURE IT. At a crossroads stands a signpost for every place you could go next, each pointing its own way. You hold a compass needle, $\mathbf{u}$. For each signpost, you ask how closely it points the way your needle does. The closest one is your best guess for where to go.

SAY IT EXACTLY. Score each word w by the dot product of $\mathbf{u}$ with its embedding $\mathbf{e}_w$. For all words at once, multiply by the transposed embedding table (Chapter 3, §3.4 explains the transpose):

$$\mathbf{z} = \mathbf{u} \mathbf{E}^\top \quad [(1 \times d)(d \times V) = 1 \times V]$$

These raw scores are called *logits*. Figure 40 shows the multiplication with every number. It is drawn standing up: $\mathbf{z} = \mathbf{u} \mathbf{E}^\top$ and $\mathbf{z}^\top = \mathbf{E} \mathbf{u}^\top$ hold the same twelve numbers (transposing both sides of a product swaps the order), and standing up, all twelve words fit on the page. Each row of $\mathbf{E}$ is one word's arrow; each row of the answer is that word's score.

	E				$\mathbf{u}^\top$		$\mathbf{z}^\top$
waste	1	0	1	0	1.27	waste	0.50
of	0	1	0	0	1.34	of	1.34
time	0	0	1	1	-0.77	time	-0.68
not	0	1	0	1	0.09	not	1.43
very	1	0	0	0	4×1	very	1.27
useful	0	0	1	0		useful	-0.77
Sorry	1	1	0	0		Sorry	2.61
,	0	0	0	1		,	0.09
this	1	0	0	1	$\times$	this	1.35
has	0	1	1	0		has	0.57
been	1	1	1	0		been	1.84
.	0	1	1	1		.	0.66
	12×4						12×1

$$\text{row Sorry} \cdot \text{column } \mathbf{u}^\top = (1)(1.27) + (1)(1.34) + (0)(-0.77) + (0)(0.09) = \mathbf{2.61}$$

Figure 40: Scoring every word. The highlighted row is **Sorry**’s arrow; it meets the column $\mathbf{u}^\top$, and the dot product, written out underneath, is **Sorry**’s logit. Every other row does the same with its own arrow.

For a whole batch, the shape is $B \times T \times V$: one score per word, per position, per sentence, the end of the road in Chapter 1, §1.9. Here are both customer replies of Chapter 3 at once:

	waste	of	time	not	very	useful	Sorry	,	this	has	been	.	
	-2.0	0.3	-1.9	0.4	0.0	-2.0	0.3	0.1	0.1	-1.7	-1.7	-1.6	← sentence 2
pos 1	-2.3	0.5	-1.9	-0.5	-1.4	-0.9	-0.9	-1.0	-2.4	-0.4	-1.9	-1.4	-1.9
pos 2	1.2	-0.9	2.5	0.5	0.1	1.1	-0.8	1.4	1.5	0.2	0.3	1.6	-1.2
pos 3	0.5	1.3	-0.7	1.4	1.3	-0.8	2.6	0.1	1.4	0.6	1.8	0.7	← sentence 1
$Z: B \times T \times V = 2 \times 3 \times 12$													

Figure 41: Logits for a batch of $B = 2$ sentences of $T = 3$ tokens over $V = 12$ words. Each row is one position’s twelve scores; the word labels sit on the back sheet, and the front sheet’s columns are the same twelve words. The front sheet is **waste of time**; its third row is the column in the figure above.

Our toy reuses the input table E at the output. This *tying* saves a $d \times V$ matrix, and GPT-2 does the same (Press and Wolf, 2017). Other models learn a separate output matrix; in the sessions it was called “w vocab”.

Each score is length times length times agreement (Chapter 3, §3.2). Here is every word, sorted by score:

word	arrow e_w	$\ e_w\ $	angle to u	logit $u \cdot e_w$	e^{logit}	probability
Sorry	(1, 1, 0, 0)	1.414	23°	2.61	13.57	0.318
been	(1, 1, 1, 0)	1.732	58°	1.84	6.30	0.148
not	(0, 1, 0, 1)	1.414	60°	1.43	4.17	0.098
this	(1, 0, 0, 1)	1.414	61°	1.35	3.87	0.091
of	(0, 1, 0, 0)	1.000	48°	1.34	3.82	0.090
very	(1, 0, 0, 0)	1.000	51°	1.27	3.55	0.083
.	(0, 1, 1, 1)	1.732	79°	0.66	1.94	0.045
has	(0, 1, 1, 0)	1.414	78°	0.57	1.78	0.042
waste	(1, 0, 1, 0)	1.414	80°	0.50	1.65	0.039
,	(0, 0, 0, 1)	1.000	88°	0.09	1.09	0.026
time	(0, 0, 1, 1)	1.414	104°	-0.68	0.51	0.012
useful	(0, 0, 1, 0)	1.000	113°	-0.77	0.46	0.011

Sorry wins: its arrow is only 23° from u . Now look at **of** and **been**. **of** points closer to u (48°) than **been** does (58°), yet **been** scores higher, 1.84 against 1.34. The reason is the other knob: **been**'s arrow has length 1.732, **of**'s only 1.000. Direction and length both count.

WHERE THIS PICTURE BREAKS. Signposts are fixed in the ground. Here the “signposts” are the same arrows the model reads at its input, and they are learned: training moves them so that the right word's signpost ends up pointing where the model's needle points.

WHY IT HAD TO BE THIS WAY. Why compare with the embedding table and not with something new? The output must be one of the V tokens the model knows, and the embedding table already holds one arrow per token. Scoring by dot product reuses the geometry of every earlier chapter: agreement times length.

A tempting wrong idea (M7.1). *The model can write any word, even one it has never seen.* The last step scores exactly V tokens, no more. In the sessions: “do you really think my output token can be different from what is up in the embedding table of quen? nope.” New words still appear in its writing, but as sequences of known pieces (Chapter 2).

Check yourself 7.1. A model has a vocabulary of V tokens and width d . One pass over a batch of B sentences of T tokens produces logits of what shape?

- (a) $B \times T \times d$, one arrow per position.
- (b) $B \times T \times V$, one score per word per position.
- (c) $B \times V$, one score per word per sentence, for the last position only.
- (d) $B \times T \times (V + 1)$, with an extra slot for new words.

In one sentence: The final arrow is scored against every token's arrow by dot product, giving one logit per word.

7.2 From scores to probabilities

The logits can be negative and do not add to 1. We turn them into shares with the same softmax as Chapter 3, now over all $V = 12$ words instead of three tokens:

$$P(w) = \frac{e^{z_w}}{\sum_{w'} e^{z_{w'}}} \quad [1 \times V \rightarrow 1 \times V]$$

As vectors, one line per step:

	waste	of	time	not	very	useful	Sorry	,	this	has	been	.
logit z	0.50	1.34	-0.68	1.43	1.27	-0.77	2.61	0.09	1.35	0.57	1.84	0.66
e^z	1.65	3.82	0.51	4.17	3.55	0.46	13.57	1.09	3.87	1.78	6.30	1.94
$P(w)$	0.039	0.090	0.012	0.098	0.083	0.011	0.318	0.026	0.091	0.042	0.148	0.045

Figure 42: Softmax on the twelve logits of **time**. Top: logits. Middle: e to each logit, all positive. Bottom: each middle number divided by their total; the bottom line adds to 1. **Sorry**’s column is highlighted.

For our arrow, the total of the e^{logit} line is 42.71, and **Sorry** gets $P = 0.318$, against 0.083 if all twelve words were equally likely. **been** comes second with 0.148.

Sanity check. The probabilities in the table add to 1, and **Sorry** has the largest logit and the largest probability. One more check you can do in your head: the ratio of two probabilities depends only on the gap between their logits. The gap between **Sorry** and **been** is 0.77, and $e^{0.77} \approx 2.15$, which matches $0.318/0.148 \approx 2.15$.

WHY IT HAD TO BE THIS WAY. Softmax keeps the three properties that Chapter 3 asked for: positive shares, only differences matter, and leaders lead. A logit is “how much more than the others”, and softmax turns “how much more” into “how often”.

In one sentence: Softmax turns the logits into one probability per word, and only the gaps between logits matter.

7.3 Temperature: how bold to be

Predict first. Divide every logit by 2 before the softmax. Will **Sorry**’s probability go up or down? What if you divide by 0.5?

PICTURE IT. A dimmer switch on how decisive the model is. Turned down, the favourite takes nearly everything. Turned up, the long shots get a real chance. In Chapter 3 you met the same knob as the *length* of a query: a longer query gave a more decisive softmax. Dividing logits by a temperature τ shrinks or stretches all the gaps at once.

SAY IT EXACTLY. $P_\tau(w) = \frac{e^{z_w/\tau}}{\sum_{w'} e^{z_{w'}/\tau}}$. Here is the whole probability vector at three temperatures:

	waste	of	time	not	very	useful	Sorry	,	this	has	been	.
$\tau = 0.5$	0.009	0.050	0.001	0.059	0.043	0.001	0.625	0.004	0.051	0.011	0.135	0.013
$\tau = 1$	0.039	0.090	0.012	0.098	0.083	0.011	0.318	0.026	0.091	0.042	0.148	0.045
$\tau = 2$	0.063	0.095	0.035	0.100	0.092	0.033	0.180	0.051	0.096	0.065	0.123	0.068

Figure 43: The same twelve logits, divided by τ before the softmax. Read down **Sorry**’s column: the lower the temperature, the bigger its share. The order of the words never changes.

And a few numbers in more detail, including an extreme $\tau = 0.1$:

temperature τ	0.1	0.5	1	2
$P(\text{Sorry})$	1.000	0.625	0.318	0.180
$P(\text{been})$	0.000	0.135	0.148	0.123
$P(\text{not})$	0.000	0.059	0.098	0.100
least likely word	0.000	0.001	0.011	0.033
spread, in bits	0.01	1.94	3.04	3.43

The last row measures how spread out the probabilities are: the average number of yes/no questions you would need to guess the word drawn. With all twelve words equally likely it would be $\log_2 12 = 3.58$ bits.

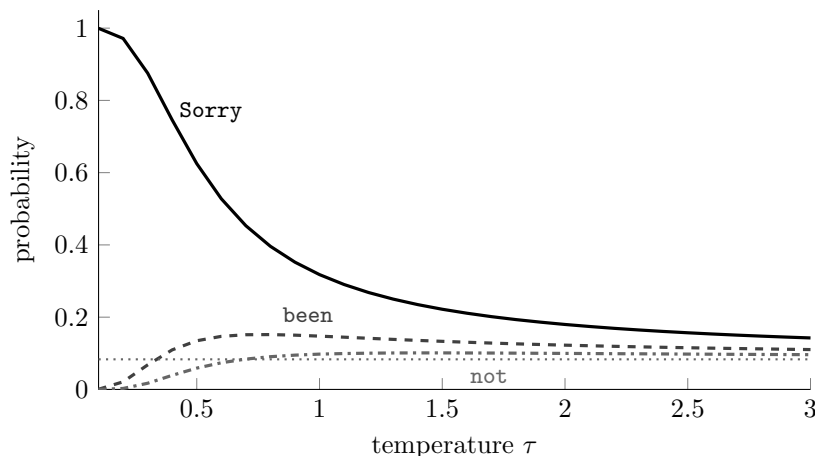


Figure 44: Temperature, measured on the toy. Low temperature hands **Sorry** nearly everything; high temperature pushes every word towards one twelfth (the dotted line, $1/12$).

WHERE THIS PICTURE BREAKS. A dimmer has no opinion about what the room looks like; it only changes how much you see. Temperature is the same: it never changes the *order* of the words. If the model’s favourite word is wrong, a low temperature makes it reliably wrong.

WHY IT HAD TO BE THIS WAY. We need the knob because different jobs want different behaviour from the same model: a reply that will be sent without review should be steady; a list of ideas for a subject line should vary.

A tempting wrong idea (M7.3). *Turning the temperature down makes the model more correct.* It makes the model more *repeatable*. In the sessions it came up as the usual fix for unreliable models: “they will control it with the help of um temperature or other small small things”. That helps when the favourite is right and the trouble is the occasional long shot. It cannot help when the favourite is wrong; that needs a better model, better context, or a check outside the model.

Check yourself 7.2. The logits for three words are $(4, 2, 0)$. You set the temperature to 2. Which statement is true?

- (a) The probabilities become the same as for logits $(2, 1, 0)$ at temperature 1.
- (b) The word with logit 0 becomes the favourite.
- (c) The probabilities do not change, because temperature only affects sampling.
- (d) The favourite is now more likely to be correct.

In one sentence: Temperature divides all logits by one number, making the model bolder or steadier without changing the order of the words.

7.4 Choosing one word: greedy, top- k , top- p

Probabilities are not yet a word. Something has to choose.

Greedy. Take the highest probability every time. Our toy, continuing `waste of time` greedily for four steps, writes `Sorry , not not`. The first two are right; then it repeats itself, because the toy only ever learned three patterns. Larger models fall into loops under greedy decoding too; Holtzman and colleagues (2020) documented this repetition.

Sampling. Draw a word at random, each with its probability. Every word can appear, including unlikely ones, which is where much of the variety, and many of the odd mistakes, come from.

Top- k (Fan, Lewis and Dauphin, 2018) keeps only the k most likely words, rescales them to add to 1, and samples. With $k = 3$ our toy keeps `Sorry, been` and `not`, which held 0.563 of the probability; after rescaling, `Sorry` has 0.564.

Top- p , also called nucleus sampling (Holtzman and colleagues, 2020), keeps the smallest set of top words whose probabilities add to at least p . With $p = 0.5$ that is 3 words (`Sorry, been, not`, together 0.563); with $p = 0.9$ it is 8 words. Top- p adapts: when the model is sure, the set is small; when it is unsure, the set grows.

A tempting wrong idea (M7.4). *Picking the top word every time gives the best reply.* It gives the best *next word*, which is not the same thing. A word that is a little less likely now can lead to a much more likely reply overall; greedy never looks ahead to find it. Beam search, which keeps several partial replies alive at once, is one answer to this.

Check yourself 7.3. A model is very sure about the next word: the top word has probability 0.97. Which rule keeps the fewest words?

- (a) Top- k with $k = 5$.
- (b) Top- p with $p = 0.9$.
- (c) Sampling from all words.
- (d) They all keep the same number of words.

In one sentence: Greedy picks the favourite, sampling draws by probability, and top- k and top- p sample only among the likely words.

7.5 Randomness you can repeat

Predict first. Sample 1,000 times from our toy’s probabilities at temperature 1. About how many times will **Sorry** appear?

Sanity check. With a fixed seed of 42, it appeared 303 times, close to the expected $1,000 \times 0.318$; been appeared 142 times. The first ten draws were **has**, **Sorry**, **been**, **this**, **of**, **.**, **this**, **has**, **of**, **Sorry**. Run again with the same seed and you get exactly the same ten; our script checks it.

PICTURE IT. A shuffled deck of cards. The order looks random, but if you write down how it was shuffled (the seed), you can deal the same hand again.

WHY IT HAD TO BE THIS WAY. Computer randomness comes from a recipe started by a seed. Fix the seed and the “random” choices repeat, which is how you debug a model’s output or test it fairly. In a product, people usually do not fix it, because the variety is wanted.

A tempting wrong idea (M7.5). *A model that gives different answers to the same prompt is broken.* If it samples, different answers are expected. The sessions put the real problem precisely: “it should give a different answer but there is no patterns to it”. Variation is fine; variation you cannot measure or explain is the problem.

Abstraction — set the randomness by the cost of a mistake. In the sessions the rule was stated plainly: “anything which deals with the real consequences should never be given a creative freedom.” A system can use the same model in two modes. Where a wrong output is cheap and variety is valuable (drafting replies for a human to edit), sample. Where a wrong output is costly (a reply sent with no review, a release that may be blocked), use steady decoding and put a separate, measurable decision layer in front of the action, like the score and yes/no questions of the memory-system sessions.

In one sentence: Sampling is random but repeatable with a seed, and how much randomness to allow is a decision about the cost of mistakes.

7.6 What the probability means

Our toy gives the true next token these probabilities at its three positions: 0.320 for **of** after **waste**, 0.320 for **time** after **waste of**, and 0.318 for **Sorry** after **waste of time**. Each is a statement about the *next token*: “of all the ways this text could continue, this one gets about a third”.

PICTURE IT. A weather forecaster who says “30% chance of rain” is *calibrated* if, over all the days they said 30%, it rained on about 30% of them. You cannot tell from one forecast; you check the record.

SAY IT EXACTLY. A model is calibrated if, among all the cases where it gives probability p , the outcome happens a fraction p of the time. That can only be measured on data the model did not train on. Guo and colleagues (2017) measured it for modern image classifiers and found them often overconfident.

WHERE THIS PICTURE BREAKS. The forecaster forecasts the weather. The model forecasts *text*. “Sorry” getting 0.318 says how often this kind of text continues with “Sorry”, not how likely it is that apologising is the right thing to do.

WHY IT HAD TO BE THIS WAY. Decisions need numbers that mean what they say. A probability you can act on has to be checked against outcomes, outside the model.

A tempting wrong idea (M7.2). *The probability the model gives is how sure it is that the answer is true.* Its probabilities are about next tokens. And if a model *writes* “I am 90% sure”, those words are more next tokens. As the sessions put it: “that is not calibrated. that is next token prediction after all”. Calibrated confidence is measured, not read off the text.

Check yourself 7.4. A model replies “I am 95% confident that the refund was processed.” What does the 95% tell you?

- (a) The refund was processed with probability 0.95.
- (b) The text “95%” was a likely continuation; it is not a measured probability of the refund.
- (c) The model’s top logit was 0.95.
- (d) Lower the temperature and the 95% becomes a true probability.

In one sentence: Model probabilities describe how text continues; whether they match reality has to be measured on held-out cases.

7.7 One wrong word does not stay small

Chapter 1 said that every word is written in the light of all earlier words, including the model’s own. Here is what that costs. Let us assume, to keep it simple, that each word of a reply is right with the same probability, whatever came before. The chance that the whole 7-word reply **Sorry, this has not been useful** is right:

each word right with probability	0.99	0.95	0.90
whole reply of 7 words right	0.932	0.698	0.478
a 100-word answer right	0.366	0.006	2.7×10^{-5}

PICTURE IT. Walking directions where each step is a little off. One small error is harmless; but each later step starts from where the last one left you, so the errors add up.

WHERE THIS PICTURE BREAKS. Real word choices are not independent: a model that has started an apology is more likely to finish it well, and one that has made a wrong claim may keep building on it. The table is the simplest version of the effect, not an exact forecast.

WHY IT HAD TO BE THIS WAY. This is the price of the loop in Chapter 1, Figure 1. The model reads its own words as if they were true, so a wrong word becomes part of the context for every word after it. Much of what is called hallucination is this compounding, and it is why long answers need checking more than short ones.

A tempting wrong idea (M7.6). *One wrong word is a small, local mistake.* It is one word, but every later word is conditioned on it. In the loop of Chapter 1, the error becomes part of the input.

Check yourself 7.5. Each word of a reply is right with probability 0.95, independently. Which reply is most likely to be entirely right?

- (a) A 7-word reply.
- (b) A 100-word reply, because more words give the model more chances to correct itself.
- (c) They are equally likely, because each word is 0.95.
- (d) It depends only on the temperature.

In one sentence: Small per-word error rates multiply over a reply, so long outputs fail far more often than short ones.

Looking back. (1) In Chapter 3, what were the two knobs of a dot product, and which one did temperature act like? (2) In Chapter 1, why did we say hallucination is a consequence of the process and not a bug? (3) In Chapter 6, why does the output read $\text{Norm}(\mathbf{h}_2)$ and not $\mathbf{h}_2$ itself?

Exercise 7.1. Using $\mathbf{u} = (1.267, 1.341, -0.767, 0.087)$, compute the logit of **been** (arrow $(1, 1, 1, 0)$) and of **time** (arrow $(0, 0, 1, 1)$) by hand. Check against the table.

Exercise 7.2. For logits $(2, 0, 1)$, compute the softmax at temperature 1 and at temperature 0.5. Which word gains, and why?

Exercise 7.3. Using the probability column of the table in §7.1, list the words kept by top- p with $p = 0.5$, and the probability of **Sorry** after rescaling.

Exercise 7.4. You run two products on the same model: (a) automatic replies sent to customers with no human review; (b) brainstorming subject lines for a marketing email. Choose a decoding rule and temperature for each, and justify each choice by the cost of a mistake.

Answers to Chapter 7

Check yourself 7.1. (b). One logit per vocabulary word, for every position of every sentence. (a) is the shape before the output step. (c) forgets that every position predicts its next token. (d) is M7.1: there is no slot for unknown words.

Check yourself 7.2. (a). Dividing by 2 gives logits $(2, 1, 0)$. (b) is wrong: temperature never changes the order. (c) is wrong: temperature changes the probabilities themselves. (d) is M7.3: it changes boldness, not correctness.

Check yourself 7.3. (b). Top- p with $p = 0.9$ keeps one word, since $0.97 \geq 0.9$; top- k always keeps 5. (c) keeps all words. (d) ignores that top- p adapts to how sure the model is.

Check yourself 7.4. (b). The words are text; their 95% is not a measured probability. (a) is M7.2. (c) confuses a logit with a probability. (d) is M7.3: temperature does not calibrate anything.

Check yourself 7.5. (a). $0.95^7 \approx 0.698$, but $0.95^{100} \approx 0.006$. (b) is M7.6 in reverse: more words give more chances to go wrong, and the model reads its own errors as context. (c) forgets that the chances multiply. (d) is M7.3 again.

Looking back. (1) Direction and length; temperature acts like the length of the query, scaling every score. (2) Because each guess is built on earlier guesses, including wrong ones; the same process makes the model capable. (3) The pre-norm stream is never normalised inside the blocks, so its size drifts with depth; the final norm fixes it before scoring.

Exercise 7.1. $\mathbf{u} \cdot (1, 1, 1, 0)$ adds the first three entries of $\mathbf{u}$: 1.84. $\mathbf{u} \cdot (0, 0, 1, 1)$ adds the last two: -0.68.

Exercise 7.2. At temperature 1: (0.665, 0.090, 0.245), as in Chapter 3. At 0.5 the logits double to (4, 0, 2) and the shares become (0.867, 0.016, 0.117), the “twice as long query” of Chapter 3, §3.7. The favourite gains, because doubling the logits doubles its lead.

Exercise 7.3. Sorry, been, not, together 0.563. After rescaling, Sorry has $0.318/0.563 \approx 0.564$.

Exercise 7.4. (a) Mistakes are costly and seen by customers: greedy or a low temperature, plus a separate check (a calibrated yes/no “safe to send?” decision, or a human handoff) before sending. (b) Mistakes are cheap and variety is the point: sample with top- p around 0.9 and a temperature near 1, and let a person pick.

What you now know

- How the last arrow becomes one score per word, by dot products with the embedding table.
- How softmax turns scores into probabilities, and why only the gaps matter.
- What temperature, greedy, sampling, top- k and top- p each do, with numbers from the toy.
- Why seeds make randomness repeatable, and how to set randomness by the cost of a mistake.
- Why model probabilities are about text, why calibration must be measured, and why errors compound over a reply.

8 Learning: how the arrows get nudged

Map of this chapter. Chapter 3 ended with a promise: training turns and stretches the query, key and value arrows until the model attends to the right words. This chapter keeps that promise. We first need a number that says how wrong a guess is (§8.1), then a way to get many such numbers from one sentence (§8.2). With that number we can draw a landscape and walk downhill on it (§8.3), watch the arrows of our toy turn as it learns (§8.4), see why real training uses small random batches (§8.5), and finally train a real, tiny transformer on my own lecture transcripts and read what it writes (§8.6). There is no calculus in this chapter; Appendix A has it for readers who want it.

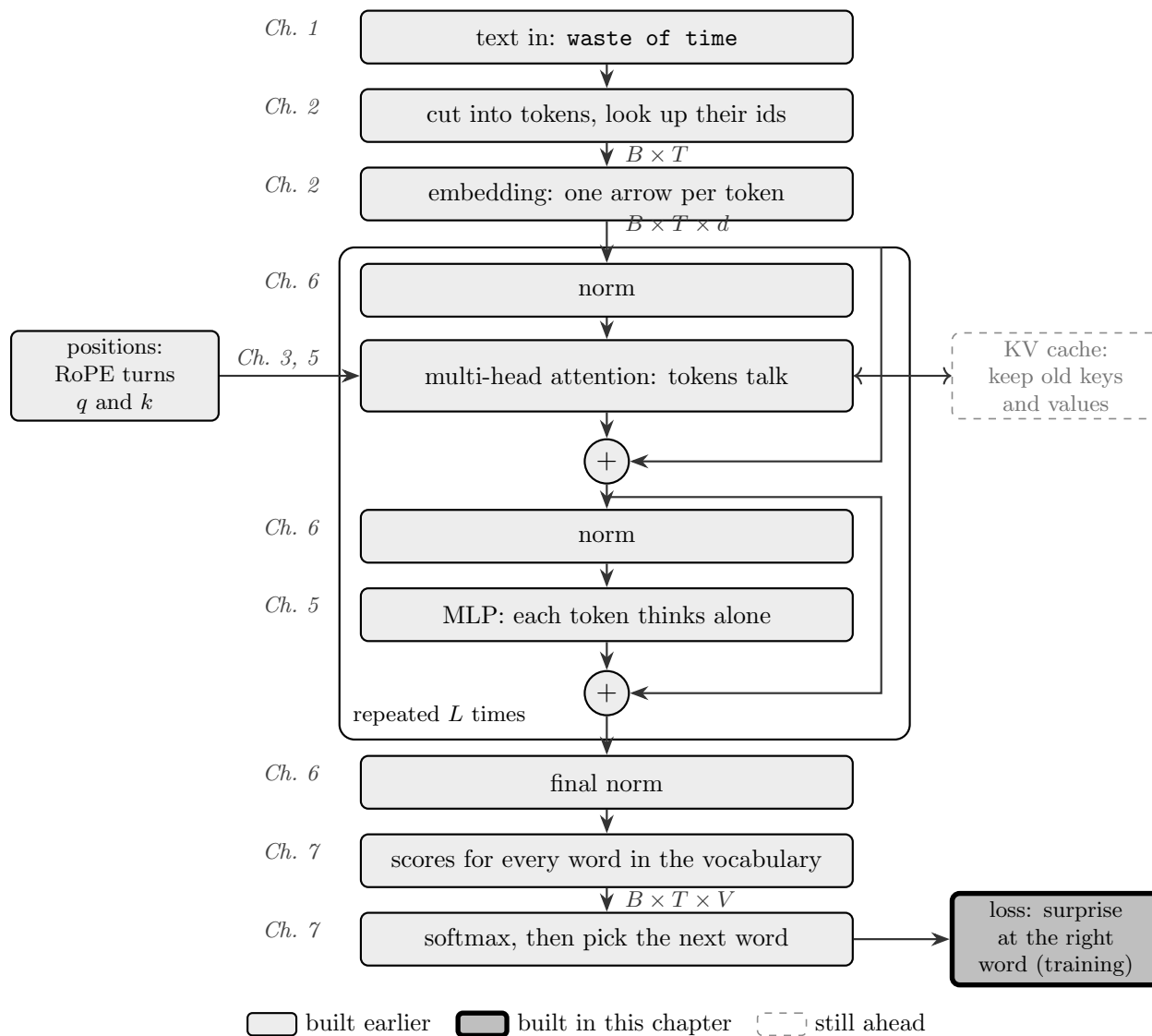


Figure 45: Where we are in the machine. You are here: the *loss* box at the bottom right. Nothing new is added to the model itself. This chapter adds the one number that compares the model’s guess with the right next word, and the walk downhill that changes every weight in the solid blocks above it. The dashed blocks come later.

Before you start.

- (1) What is $0.5 \times 0.5 \times 0.5 \times 0.5$? *Refresher: multiply step by step: 0.25, 0.125, 0.0625. Multiplying numbers below 1 makes them smaller every time.*
- (2) The natural logarithm $\ln x$ answers “ e to what power gives x ?”. What is $\ln 1$? Is $\ln 0.5$ positive or negative? *Refresher: $e^0 = 1$, so $\ln 1 = 0$. Since e^{negative} is below 1, the log of any number between 0 and 1 is negative: $\ln 0.5 \approx -0.693$.*
- (3) Logs turn multiplying into adding: $\ln(a \times b) = \ln a + \ln b$. Check it with $a = b = e$. *Refresher: $\ln(e \cdot e) = \ln e^2 = 2 = 1 + 1$.*
- (4) You stand on a slope in thick fog. You step one foot left and feel the ground go down. Which way do you walk? *Refresher: left. You do not need to see the valley; you need the slope under your feet.*

8.1 How wrong is a guess? Surprise

Predict first. Our toy reads **waste of time** and must guess the next word. The right answer is **Sorry**. Suppose the model gives **Sorry** a probability of 15%. Before reading on, write down how you would score that guess. Is it “wrong” because another word got more? Is it “85% wrong”?

Let us first see what our toy actually says. Take the attention output for **time** from Chapter 3, $\mathbf{o}_{\text{time}} = (0.665, 0.910, 0.245, 0.090)$. To turn it into a score for every word in the 12-word toy vocabulary, we take its dot product with every word’s own embedding arrow (the same table E that gave the words their arrows; reusing it is called *tying*, and Chapter 7 explains why it is reasonable). Then softmax turns the 12 scores into shares, exactly as in §3.7. Figure 46 shows the whole calculation as matrices.

Sanity check. The probabilities in the column $\mathbf{p}$ are all positive and add to 1 (our script checks this). **Sorry** gets 14.6%, second place behind **been**. Not great, not terrible. We need one number for “how bad”.

PICTURE IT. A weather forecaster says “90% chance of sun” and it rains. You are very surprised, and you trust the forecaster less. If they had said “50–50”, the rain would surprise you a little. If they had said “90% rain”, hardly at all. Surprise depends only on how much probability the forecaster put on *what actually happened*.

SAY IT EXACTLY. The surprise of an outcome that was given probability p is $-\ln p$:

probability given to the right word	1	0.5	0.1	0.01
surprise $-\ln p$	0.000	0.693	2.303	4.605

A certain and correct guess costs nothing. Each time the probability on the right word is divided by 10, the surprise goes up by the same 2.303. For our toy, read the highlighted row of Figure 46: $\mathbf{y}$ points at **Sorry**, where $\mathbf{p}$ holds 0.146, so the surprise is $-\ln 0.146 = 1.927$. The other eleven cells of $\mathbf{p}$ do not enter the loss at all. This number is the *loss* for this one exercise. Averaged over many exercises it is called the *cross-entropy loss*, and it is the only number a language model is trained to make small:

$$\text{loss} = -\frac{1}{N} \sum_{\text{exercises}} \ln P(\text{right next word} \mid \text{words before it}).$$

A model that knows nothing and spreads its bets evenly over the 12 words gives each $1/12$, a loss of $\ln 12 = 2.485$. Our toy, with hand-picked weights, already does a little better.

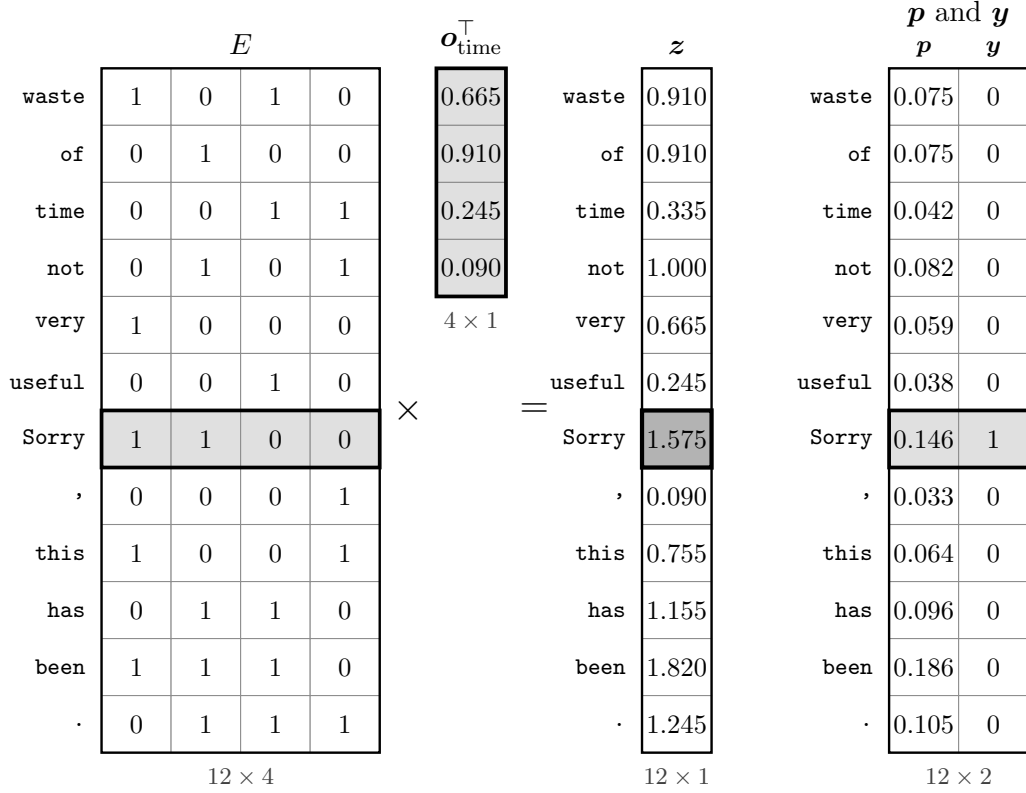


Figure 46: Scoring every word at once. Left: the embedding table E (one row per word) times **time**'s attention output, stood up as a column. Each row meets the column in a dot product; the highlighted row is **Sorry**, worked out underneath. Right: softmax turns the 12 scores z into probabilities p ; next to them is the answer y , a 1 on the right word and 0 everywhere else.

WHERE THIS PICTURE BREAKS. A forecaster is judged on rain or sun. Our loss looks only at the probability on the one right word. It does not care whether the wrong probability went to **been** (close in meaning) or to **.** (not close at all). So the loss cannot tell “nearly right” from “far off”; it can only tell “put more on the right word”.

WHY IT HAD TO BE THIS WAY. Why the logarithm and not, say, $1-p$? Chapter 1 wrote a sentence’s probability as a product of next-word probabilities. Products of many numbers below 1 collapse towards zero: a computer that multiplies 2000 probabilities of 0.5 gets exactly 0.0, because the true value is too small to store. The log turns that product into a sum, $2000 \times 0.693 = 1386.3$, which is easy to store, and a sum can be shared out exercise by exercise. The log has a second effect you should notice: as p goes towards 0, $-\ln p$ grows without limit. A model that is confident *and* wrong is punished far more than one that admits it does not know. That is the behaviour we want from anything that will later say things to customers.

A tempting wrong idea (M8.1). *“The loss is the share of wrong answers, like a test score.”* A test score only checks whether the top answer was right. Our toy’s top answer is **been**, which is wrong, yet its loss of 1.927 is better than the 2.485 of a know-nothing model, because it did put 14.6% on **Sorry**. The loss rewards moving probability towards the right word long before that word wins.

A tempting wrong idea (M8.2). *“For text, the error is the prediction minus the answer.”* In my house-price example in the sessions, you were “subtracting yhat minus y”. That works when both are numbers. Here the answer is the word **Sorry** and the prediction is a list of 12 probabilities: **Sorry** minus **been** means nothing. What we can measure is how much probability went to the right word, and that is what surprise measures. (Appendix A shows a nice twist: the *direction* of the correction still looks like “prediction minus answer”, one probability at a time.)

Check yourself 8.1. A model gives the right next word a probability of 0.5. What is its loss on this exercise?

- (a) 0.5, because half of the probability went to wrong words.
- (b) 0.693.
- (c) 0, because 0.5 is more than any other word got, so the guess was right.
- (d) -0.5 , the prediction minus the answer.

In one sentence: the loss is the surprise $-\ln p$ at the right next word, averaged over exercises, and it rewards every bit of probability moved towards the right word.

8.2 One pass, many exercises

Predict first. The full training text is **waste of time Sorry , this has not been useful:** 10 tokens. How many next-word exercises does one pass through the model give us? One? Ten?

Chapter 3 built attention so that row t only sees tokens $1, \dots, t$ (the mask). So the output at every position is an honest guess about the token after it, and all of them come out of the same pass. Running our skeleton model once over the whole reply gives 9 exercises at once:

	waste	of	time	not	very	useful	Sorry	,	this	has	been	.	$-\ln p$
after waste	0.07	0.07	0.03	0.07	0.07	0.03	0.20	0.03	0.07	0.07	0.20	0.07	2.606
after of	0.07	0.07	0.07	0.11	0.07	0.04	0.11	0.07	0.11	0.07	0.11	0.11	2.698
after time	0.07	0.07	0.04	0.08	0.06	0.04	0.15	0.03	0.06	0.10	0.19	0.10	1.927
after Sorry	0.08	0.06	0.08	0.10	0.06	0.05	0.09	0.06	0.10	0.08	0.12	0.13	2.752
after ,	0.09	0.06	0.07	0.09	0.06	0.05	0.11	0.05	0.09	0.09	0.15	0.12	2.465
after this	0.10	0.05	0.08	0.08	0.07	0.05	0.10	0.05	0.10	0.08	0.14	0.11	2.571
after has	0.09	0.06	0.08	0.08	0.06	0.06	0.09	0.05	0.08	0.09	0.15	0.13	2.576
after not	0.09	0.05	0.09	0.08	0.05	0.06	0.08	0.05	0.08	0.09	0.14	0.13	1.935
after been	0.08	0.05	0.08	0.09	0.05	0.05	0.09	0.06	0.09	0.08	0.13	0.14	3.069

9 exercises $\times$ 12 words, plus the surprise at the dark cell (mean 2.511)

Figure 47: One pass, nine exercises. Each row is the toy’s probability for every next word after reading up to the word on the left; the dark cell is the probability it gave the word that really came next. The column on the right is the surprise $-\ln p$ for that dark cell. Row 3 (after **time**) is the exercise of §8.1.

The average surprise over the nine is 2.511; the sum is 22.598. Look at row 3, after **time**: it is the same 1.927 as in §8.1, to every decimal, even though the model now also saw the six reply words. That is the mask doing its job (our script asserts the two are equal).

PICTURE IT. A tutor marks a student’s essay line by line. Before the student writes line 4, the tutor shows the correct line 3, so one early slip does not ruin every later line. In training, the model is always given the true earlier words, never its own guesses. This is called *teacher forcing* (Williams and Zipser, 1989).

WHERE THIS PICTURE BREAKS. The tutor helps during practice only. When the model writes a real reply (Chapter 7), nobody hands it the correct earlier words; it builds on its own. This gap between practice and use is one source of the compounding errors we called hallucination in Chapter 1.

WHY IT HAD TO BE THIS WAY. Without the mask, each pass would give one honest exercise and the rest would be spoiled by peeking. With it, every position of every sentence in a batch is an exercise. Here is the smallest batch that shows it: two replies of three tokens each, stacked as in §3.9, and the grid of words they must guess.

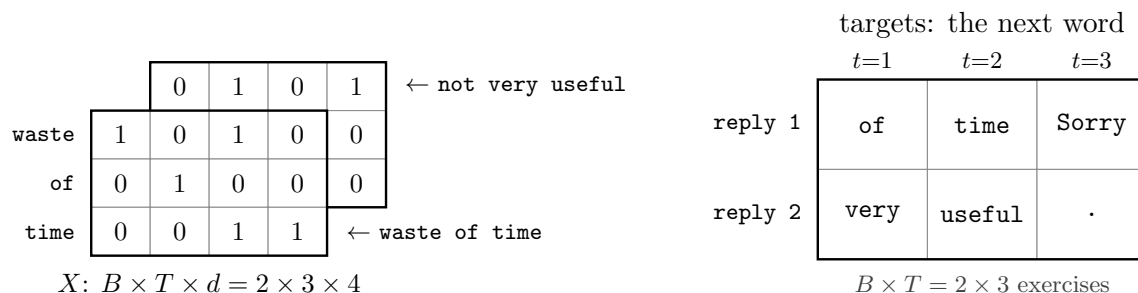


Figure 48: A batch of $B = 2$ replies, $T = 3$ tokens each. Left: the input arrows, one 3×4 sheet per reply, stacked into a $2 \times 3 \times 4$ block. Right: the targets, each input shifted one place to the left, so every cell is one exercise. (The full stop that ends reply 2 is our assumption, for the example.)

A real batch of 32 sequences of 64 tokens is the same picture, bigger: 32×64 exercises per pass. That factor, more than anything clever, is why transformers can be trained on so much text.

A tempting wrong idea (M8.5). “A sentence gives the model one exercise.” In the sessions this came up as “because we are predicting one token at a time, right?” That is true when the model *writes*. When it *learns*, every position is an exercise, all solved together in one pass. The figure in §8.2 has nine.

Check yourself 8.2. In the figure of §8.2, why is the probability of **Sorry** in row 3 exactly the same as when the model saw only **waste of time**?

- (a) Because **Sorry** is the most likely word in both cases.
- (b) Because the mask stops row 3 from seeing any later token, so the later words cannot change it.
- (c) Because the model has not been trained yet.
- (d) Because each sentence gives only one exercise, so row 3 is computed alone.

In one sentence: thanks to the mask, one pass over T tokens gives $T - 1$ honest exercises, and the loss is their average surprise.

8.3 The landscape, and walking downhill

Predict first. Keep everything in the toy fixed except one knob: the direction of **time**’s query. At the moment it points straight at **waste**’s key (angle 0° , as in §3.6). Suppose we turned it to point at **of**’s key instead (90°). Will the loss go up or down? By roughly how much?

PICTURE IT. You are on a hillside in thick fog, and you want to reach the lowest point of the valley. You cannot see it. What you can do is feel the ground: put one foot a little to the left, then a little to the right, and notice which way is lower. Step that way. Repeat. In one session I put it as an algorithm: “look at the weights ask the gradient step the downhill repeat”. Here the hillside is the loss, and your position is the setting of the knob.

Let us draw the hillside for our one knob. We turn **time**’s query through every angle, keeping its length at 1.414, and compute the loss each time:

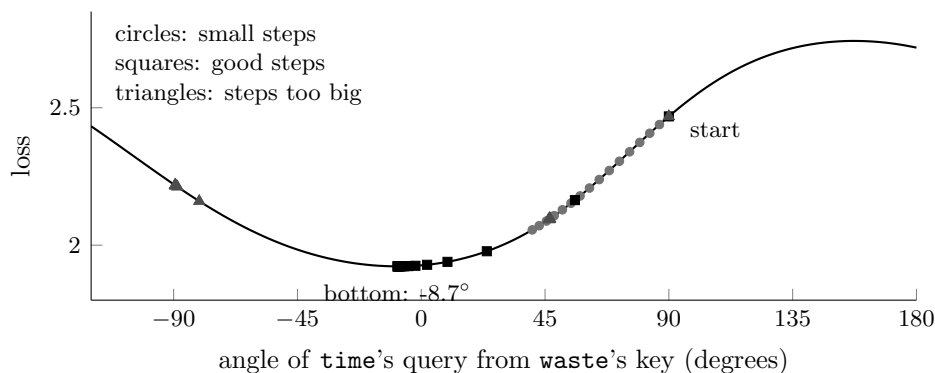


Figure 49: A real slice of the toy’s loss landscape: one knob, the direction of `time`’s query. Marks show 15 steps of three walkers who all start at 90° and differ only in step size.

Sanity check. At 90° (pointing at `of`) the loss is 2.468; at 0° (pointing at `waste`, our hand-picked setting) it is 1.927; the very bottom is at -8.7° , with loss 1.923. So the hand-picked direction was nearly the best one, and pointing at `of` is worse. Does that match your prediction?

Now be the walker in the fog at 90° . Probe: at 89° the loss is 2.460, at 91° it is 2.477. Lower is towards smaller angles, so turn towards `waste`. The steepness is about 0.0085 loss per degree. How far to step? The rule every training method uses is

$$\text{new setting} = \text{old setting} - (\text{step size}) \times (\text{slope}).$$

The step size is called the *learning rate*. Three walkers, three learning rates, 15 steps each:

- **Small** (400): after 15 steps the angle has crawled from 90° to only 40.4° . Safe and painfully slow.
- **Good** (4000, ten times larger): the first step alone moves to 55.9° ; after 15 steps it is at -8.6° , close to the bottom. Steps shrink by themselves near the bottom, because the ground flattens there and the slope gets small.
- **Too big** (20000, fifty times the small one): the first step overshoots the valley to -80.7° , the next jumps back to 46.3° , then -88.6° , 46.8° , and so on. It bounces between the two sides of the valley forever and ends at loss 2.217, worse than where the good walker is.

SAY IT EXACTLY. A real model has not one knob but millions (every number in every W). The slope of the loss for every knob at once is called the *gradient*, an arrow in knob-space that points uphill. Gradient descent steps against it: $\theta \leftarrow \theta - \eta g$, where θ lists every knob, η is the learning rate and g is the gradient. The idea has been around since Cauchy (1847).

WHERE THIS PICTURE BREAKS. Our probe-left-probe-right trick needs two extra passes per knob. With millions of knobs that is hopeless. Real training gets all the slopes at once, for the price of about one extra pass, with an exact bookkeeping method called *backpropagation* (Rumelhart, Hinton and Williams, 1986). Appendix A works it out and checks every result against probing. The picture of a hill also hides how strange a landscape with millions of directions is: there are many valleys, and wide flat regions.

WHY IT HAD TO BE THIS WAY. Why not jump straight to the bottom? Because nobody has a formula for where it is. All we can ever know is the ground under our feet: the loss and its slope at the current setting. Small, repeated, local corrections is the only plan that works without a map, which is why every model in this book is trained this way.

A tempting wrong idea (M8.3). “A bigger step size always learns faster.” The triangles show the opposite. In one of my sessions we tried exactly this with a larger step and saw that “my loss started to increase over the bit of a time.” Too large a step overshoots the valley; past a point it bounces or flies off. Choosing the learning rate is one of the first decisions in any training run (Chapter 11).

A tempting wrong idea (M8.7). “Training finds the best possible weights.” Our one-knob slice has a single valley, so here it does. A real landscape has many valleys, and walking downhill finds a low one near where you started, not the lowest one on the map. In practice that turns out to be good enough, and nobody can check that it is the best.

Check yourself 8.3. You probe a knob and find the loss is 3.10 a little to the left and 3.02 a little to the right. With a sensible learning rate, what do you do?

- (a) Step left, because the loss there is bigger and you want a bigger change.
- (b) Step right by a small amount in proportion to the slope.
- (c) Step right as far as possible, since bigger steps learn faster.
- (d) Stop: you have found the best setting, since one side is lower.

In one sentence: the loss draws a landscape over the knobs; training probes the slope where it stands and takes a step downhill whose size, the learning rate, must be neither too small nor too large.

8.4 The payoff: watching the arrows turn

Predict first. In §3.10 we promised that training turns **time**’s query towards **waste**’s key and stretches it. Let us test that honestly. Throw away our hand-picked W_Q, W_K, W_V and start from random numbers. At the start, **time** gives **waste** a share of 0.302, and its query points 95.9° away from **waste**’s key. After 60 downhill steps on the single exercise “**waste of time** $\rightarrow$ Sorry”, what do you expect the share and the angle to be?

We train all three machines at once, starting from the same random numbers, with three learning rates. Here the slopes come from backpropagation, the method of Appendix A.

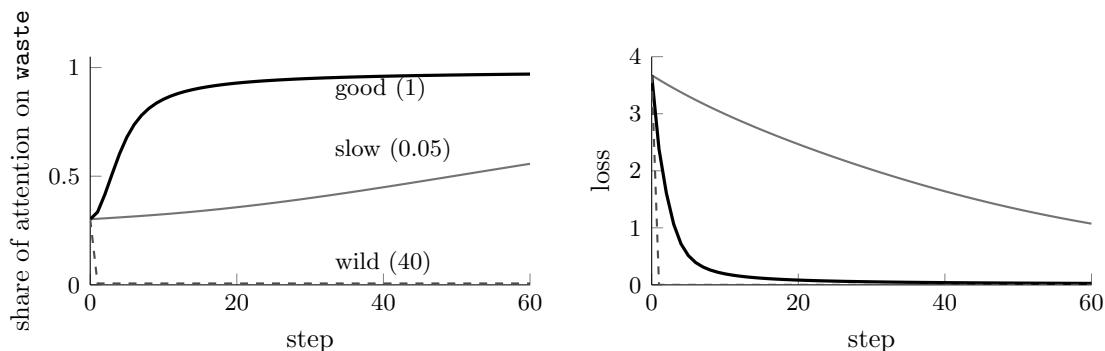


Figure 50: Training the toy’s three machines from random numbers. Left: how much **time** attends to **waste**. Right: the loss. The wild run drives the loss to zero while ignoring **waste**.

With the good learning rate, after 60 steps:

	start	after 60 steps
loss	3.675	0.026
share of time 's attention on waste	0.302	0.970
angle between time 's query and waste 's key	95.9°	9.4°
length of time 's query	1.959	3.372

Here are the two machines that did it, as numbers, and the arrows they now make:

W_Q , before				W_K , before			
0.06	-0.07	0.32	0.05	-0.27	-0.16	0.21	0.52
-0.27	0.18	0.65	0.47	-0.06	0.68	-0.33	0.18
-0.35	-0.63	-0.31	0.02	0.45	0.05	-0.37	-0.46
-1.16	-0.11	-0.62	-0.37	-0.23	0.11	-0.50	-0.10
4×4				4×4			
W_Q , after 60 steps				W_K , after 60 steps			
0.06	-0.07	0.32	0.05	-1.40	-0.96	-0.32	0.37
-0.27	0.18	0.65	0.47	0.55	1.11	-0.06	0.25
-0.81	-1.33	-0.30	0.09	-0.16	-0.38	-0.65	-0.53
-1.62	-0.81	-0.61	-0.30	0.29	0.48	-0.25	-0.02
4×4				4×4			

Figure 51: The query machine W_Q and the key machine W_K before and after 60 good steps. Look at which numbers moved. In W_Q only rows 3 and 4 changed: the only query that matters here is **time**'s, and its arrow $(0, 0, 1, 1)$ uses only those two rows (§3.3), so rows 1 and 2 got no nudge at all. In W_K every row moved, because the three keys together use all four rows. None of the numbers was told about **waste**.

$$\begin{array}{c}
\mathbf{x}_{\text{time}} \\
\text{time} \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 1 \\ \hline \end{array} \\
1 \times 4
\end{array}
\times
\begin{array}{c}
W_Q \\
\begin{array}{|c|c|c|c|} \hline 0.06 & -0.07 & 0.32 & 0.05 \\ \hline -0.27 & 0.18 & 0.65 & 0.47 \\ \hline -0.81 & -1.33 & -0.30 & 0.09 \\ \hline -1.62 & -0.81 & -0.61 & -0.30 \\ \hline \end{array} \\
4 \times 4
\end{array}
=
\begin{array}{c}
\mathbf{q}_{\text{time}} \\
\text{time} \begin{array}{|c|c|c|c|} \hline -2.43 & -2.14 & -0.92 & -0.21 \\ \hline \end{array} \\
1 \times 4
\end{array}$$

highlighted row (**time**) · highlighted column (column 1):
 $(0)(0.06) + (0)(-0.27) + (1)(-0.81) + (1)(-1.62) = -2.43$

$$\begin{array}{c}
\mathbf{x}_{\text{waste}} \\
\text{waste} \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \end{array} \\
1 \times 4
\end{array}
\times
\begin{array}{c}
W_K \\
\begin{array}{|c|c|c|c|} \hline -1.40 & -0.96 & -0.32 & 0.37 \\ \hline 0.55 & 1.11 & -0.06 & 0.25 \\ \hline -0.16 & -0.38 & -0.65 & -0.53 \\ \hline 0.29 & 0.48 & -0.25 & -0.02 \\ \hline \end{array} \\
4 \times 4
\end{array}
=
\begin{array}{c}
\mathbf{k}_{\text{waste}} \\
\text{waste} \begin{array}{|c|c|c|c|} \hline -1.56 & -1.34 & -0.97 & -0.16 \\ \hline \end{array} \\
1 \times 4
\end{array}$$

highlighted row (**waste**) · highlighted column (column 1):
 $(1)(-1.40) + (0)(0.55) + (1)(-0.16) + (0)(0.29) = -1.56$

Figure 52: After training: **time**’s arrow times the trained W_Q gives its query, and **waste**’s arrow times the trained W_K gives its key. One row times one column per number, as always; the highlighted cell is worked out underneath each product.

The query turned by about 87 degrees towards **waste**’s key and grew longer. That is §3.10 exactly: turned towards the key that helps, and stretched to be more decisive. Nobody told the model to look at **waste**. The only instruction was “make the surprise at **Sorry** smaller”, and attending to **waste** turned out to be the way downhill. Why that way? Look at **waste**’s value arrow in our original toy: (1, 1, 0, 0), the very same arrow as **Sorry**’s embedding. Handing it over pushes the output towards **Sorry**.

The slow run (learning rate 0.05) is going the same way but has only reached a share of 0.557 and a loss of 1.073.

Now look at the wild run (learning rate 40). Its loss is 0.000: perfect! Yet **time** gives **waste** a share of only 0.006, and the query points 96.1° away from **waste**’s key. Giant steps found a different valley: they changed what the other tokens hand over until **Sorry** came out anyway. On this one exercise that works. On any new sentence, a model that ignores **waste** is useless. This is our first look at a problem §8.6 measures properly: a low loss on the exercises you trained on does not mean the model learned what you hoped.

A tempting wrong idea (M8.4). “*Training adjusts each word’s arrow inside the sentence.*” Nothing in the runs above touched an arrow directly. Only the numbers in W_Q , W_K , W_V changed; the arrows moved because the machines that make them changed. That is why learning transfers: the same machine also makes the arrows for **waste of money**.

A tempting wrong idea (M8.8). “A low training loss means the model learned what we wanted.” The wild run has the lowest loss of all three and the worst attention. Loss measures only the exercises it was given.

Check yourself 8.4. In the good run, which change made the model attend to **waste**?

- (a) We instructed it to look at **waste**.
- (b) Steps that lowered the surprise at **Sorry** happened to turn the query towards **waste**’s key.
- (c) The arrow of **waste** was moved closer to the arrow of **time**.
- (d) The loss reached zero, which forces attention onto the most useful word.

*In one sentence: trained only to reduce surprise, the toy’s query for **time** turned from 95.9° to 9.4° off **waste**’s key and grew longer, while a run with steps that were too big found a zero loss that ignores **waste**.*

8.5 Batches, noise, and a smarter walker

Predict first. A real dataset has billions of exercises. Computing the exact downhill direction would mean running all of them before every single step. What if you used only a handful of exercises per step instead? Would the direction still be any good?

PICTURE IT. You are lost in a town and ask for directions. One passer-by points roughly the right way, but might be off. Ask eight people and average their arrows, and the average points much more reliably. Each person is one exercise; their arrow is the downhill direction for that exercise alone.

Let us measure it on our nine exercises (§8.2), from the random start of §8.4. For each batch size we draw 300 random batches, compute their downhill direction, and compare it with the direction from all nine exercises, using the agreement of two arrows from §3.2 (cos of the angle between them; 1 means identical):

exercises per batch	1	2	4	8
average agreement with the full direction	0.272	0.439	0.660	0.949

One exercise points, on average, about 74° away from the full direction; eight point about 18° away. Each step is noisy, but noisy steps that are right on average still go downhill on average, and they are thousands of times cheaper. This is *stochastic gradient descent* (its roots are in Robbins and Monro, 1951). Do you see the coin tosses of §3.8 again? Averaging n noisy arrows cuts the noise roughly like $1/\sqrt{n}$, the same square-root rule.

SAY IT EXACTLY. Each step: pick a batch of exercises, compute the average gradient $\mathbf{g}_{\text{batch}}$ over that batch only, and update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{g}_{\text{batch}}$.

A smarter walker: Adam. Plain steps have two weaknesses. On a long, gently sloping valley floor they creep; and when some knobs are steep and others nearly flat, one learning rate is too big for the first and too small for the second. Adam (Kingma and Ba, 2015) fixes both with two memories. It keeps a running average of recent directions, so consistent directions build up speed

and noisy ones cancel (like a heavy ball rolling downhill). And it keeps a running average of how large each knob’s slope has been, and divides by it, so every knob takes steps of similar size, steep or flat. On our tiny toy it is not faster: plain steps reach a loss below 0.1 after 18 steps and Adam after 22. Its advantage shows on large models, where knobs differ in steepness by many factors of ten. Nearly every model in Chapter 10 was trained with Adam or its variant AdamW (Loshchilov and Hutter, 2019).

WHERE THIS PICTURE BREAKS. The passers-by picture assumes every exercise is equally trustworthy and independent. Real batches are not: nearby text is similar, and some exercises are far more frequent than others. Also, the noise is not only a cost. Keskar and colleagues (2017) found that small, noisy batches tend to end in wider valleys, which carry over better to new text; how much this matters for large models is still debated.

WHY IT HAD TO BE THIS WAY. Why accept noisy steps at all? Because the exact direction costs a pass over the entire dataset, and by the time you had it, a noisy walker would have taken thousands of steps. Cheap, roughly right, and often beats expensive and exact.

A tempting wrong idea (M8.6). *“Each step on a batch moves exactly downhill on the whole dataset.”* With one exercise per batch, the direction is off by about 74° on average. A training step is a vote by a few exercises, not the verdict of all of them.

Check yourself 8.5. Doubling the batch size from 4 to 8 exercises changes the average agreement from 0.660 to 0.949. What does that mean?

- (a) Each step is now exactly downhill.
- (b) Each step’s direction is closer to the full-data direction, at twice the cost per step.
- (c) The learning rate has doubled.
- (d) The loss is now twice as small.

In one sentence: real training steps use small random batches, whose directions are noisy but right on average, and Adam smooths the direction and evens out the step size across knobs.

8.6 A real run: a tiny transformer learns from my lectures

Predict first. Our real model reads text one character at a time. My transcripts use 41 different characters (letters, digits, space and a little punctuation). Before any training, the model knows nothing. What loss should it start with? (Hint: the know-nothing model of §8.1.)

Everything from Chapters 3 to 8 now runs together, at a size a laptop can handle:

- characters as tokens (41 of them), each with an arrow of $d = 64$ numbers, plus a learned position arrow (Chapter 4);
- two blocks, each with 2 attention heads and an MLP (Chapters 5 and 6), 106,816 knobs in all, with the output scores tied to the embedding table as in our toy;
- training text: sessions 1–8 (961,408 characters); held-out text, never trained on: session 9 (324,250 characters);
- 1,500 steps of AdamW, each on a batch of 32 pieces of 64 characters: 3,072,000 next-character exercises in total;

- one CPU thread and fixed random seeds, so running `code/ch8_train.py` again gives identical numbers.

Sanity check. A know-nothing guess over 41 characters has loss $\ln 41 = 3.714$. Our model starts at 3.816, close to that, as it should. My first attempt did not: with the default starting weights of the software library, the starting loss was 39.970, ten times worse than guessing. That one number said something was broken before a single step was taken. (The cause: with the output tied to large starting embeddings, every character’s arrow agreed strongly with itself, so the model began confidently repeating whatever it had last seen. Starting the embeddings small, with spread 0.02 as in Radford and colleagues’ first GPT, 2018, fixed it.) *Always check the starting loss against the know-nothing value.*

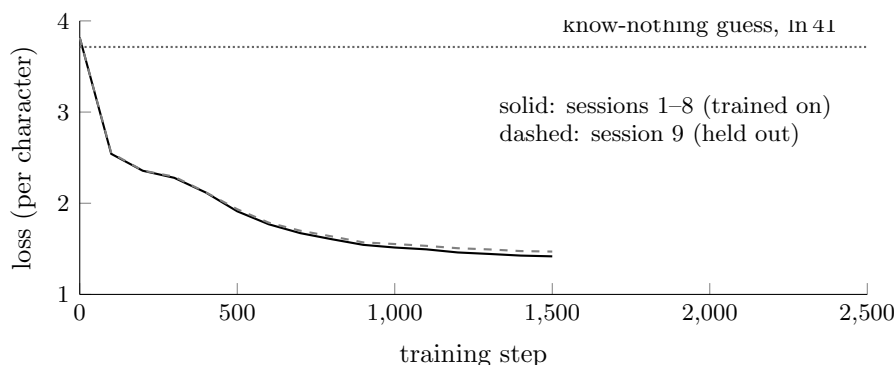


Figure 53: The real loss curve of our tiny transformer, measured every 100 steps.

After 1,500 steps the loss is 1.418 on the training text and 1.470 on session 9, which it never saw. A loss is easier to feel as a *perplexity*, e^{loss} : the model is as unsure as if it were choosing evenly among that many characters. It started at 45.4 and ended at 4.3. Here is what the model writes after the prompt `waste of time`, at three moments (same random seed for the sampling each time):

```

step 0: waste of timeb5,5v'sjvicctp7yuapguv0m4?mul85955g?gh5e9x7i74ne9rh'1.9d0nm.p1h
uejowim18wih7vd
step 800: waste of time st very icall if nout what look well to have not he apeck aprt
aeposing with ad
step 1,500: waste of time session in termuch uses whuloop wer another in i hour the
end ineed in tright?

```

At step 0 it is noise. By the end it writes words, spaces, full stops, and one phrase you may recognise from my sessions: *for example, if you*. It has never been told what a word is. Everything it knows came from nudging 106,816 knobs downhill on next-character surprise.

When the loss lies: overfitting. Now train the same model on a tiny slice, the first 4,000 characters of the transcripts, for 1,200 steps.

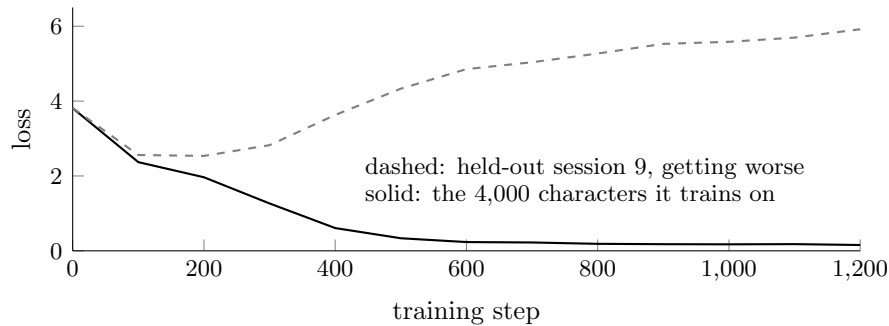


Figure 54: Overfitting, measured. Held-out loss is best at step 200 (2.535) and then rises while training loss keeps falling.

The training loss falls to 0.155: the model has nearly memorised those 4,000 characters. The loss on session 9 is best at step 200 and then climbs to 5.920, worse than guessing. It is the wild run of §8.4 again, at full size.

PICTURE IT. A student who memorises last year’s answer key scores full marks on last year’s paper and fails this year’s. A student who learns the method does a little worse on last year’s paper and well on this year’s. The held-out text is this year’s paper. As I said in one session: “it will overfit. you need to have a substantial training data with consistent patterns into it”.

WHERE THIS PICTURE BREAKS. The exam picture suggests memorising is always bad. Large language models do memorise some text, and some of that is useful (facts, names, code idioms). The problem is memorising *instead of* learning patterns that carry over, and the only way to tell the two apart is a held-out measurement.

WHY IT HAD TO BE THIS WAY. This is why every training run in this book reports two curves. The training curve tells you whether the walker is going downhill. Only the held-out curve tells you whether downhill is where you wanted to go. It is the same reason the fraud model in my sessions was checked on transactions it had not trained on.

Check yourself 8.6. Training loss is falling but held-out loss has started rising. Which reading is right?

- (a) The model is improving; training loss is what counts.
- (b) The model is starting to memorise its training text; it is time to stop, add data, or shrink the model.
- (c) The learning rate is too small.
- (d) Held-out loss always rises at the end; ignore it.

In one sentence: a real 106,816-knob transformer, trained on next-character surprise alone, went from noise to word-like text with a held-out perplexity of 4.3, and a run on too little text showed training loss falling while held-out loss rose.

Abstraction — learning is a loop of guess, surprise, small correction. Strip away the transformer and this chapter describes a loop: make a prediction, measure how surprised you should be by what actually happened, nudge your settings a little in the direction that would have reduced the surprise, repeat on fresh cases, and keep some cases aside to check that the nudges carry over. The same loop runs whenever you improve an agent’s prompts against

an evaluation set, tune a retriever on past queries, or adjust a decision rule from outcomes. So does its failure: tune against the same evaluation cases for long enough and you overfit to them, exactly like the 4,000-character run. The defence is the same too: a held-out set you do not tune against.

Looking back.

- (1) (Chapter 1) A sentence's probability is a product of next-word probabilities. What does the loss of this chapter do to that product, and why does it matter for a computer?
- (2) (Chapter 3) Why did dividing scores by $\sqrt{d_k}$ matter for learning, not only for the starting shares? (Appendix A, §A.5, measures it.)
- (3) (Chapter 3) Which part of attention lets one pass over 10 tokens give 9 honest exercises?

Exercises

Exercise 8.1. A model gives the right next word a probability of 0.25. What is the surprise? Compare it with the know-nothing loss for a vocabulary of 12 words, and say which is better.

Exercise 8.2. A training step uses a batch of 32 sequences, each 64 tokens long. How many next-token exercises does one step contain? Which idea from Chapter 3 makes that possible?

Exercise 8.3. In §8.3, the slope at 90° is 0.0085 loss per degree. Using the update rule, how far does the first step of the good walker (learning rate 4000) move the angle? Check your answer against the text.

Exercise 8.4. The wild run of §8.4 reached a loss of 0.000 while ignoring **waste**. Describe one measurement that would have caught this before you shipped the model.

Exercise 8.5. You train a model for a support team. Training loss keeps falling; held-out loss is lowest at step 200 and rises after. Give three different things you could change, and say what each assumes about the cause.

Answers to Chapter 8

Check yourself 8.1. (b): $-\ln 0.5 = 0.693$. (a) reads the loss as the share of wrong probability (M8.1). (c) scores only the top answer, like a test (M8.1); the loss would still be 0.693 because half the probability went elsewhere. (d) subtracts prediction from answer as for a price (M8.2), which has no meaning for words, and a surprise can never be negative.

Check yourself 8.2. (b). The mask hides every later token from row 3. (a) is false: **Sorry** is not the top word (**been** is). (c) is true but irrelevant: the equality holds for any weights. (d) is M8.5: there are nine exercises in that one pass, not one.

Check yourself 8.3. (b). Lower is to the right, so step right, by an amount proportional to how steep it is. (a) climbs uphill. (c) is M8.3: too large a step overshoots. (d) confuses “one side is lower” with “this is the bottom” (M8.7); at the bottom, both sides are higher.

Check yourself 8.4. (b). No one instructed the model (a). Arrows are never moved directly; the machines change (c is M8.4). (d) is M8.8: the wild run reached zero loss while ignoring **waste**.

Check yourself 8.5. (b). More exercises per step give a direction that agrees better with the full data, and each step costs more. (a) is M8.6: still not exact. (c) and (d) mix up batch size with the learning rate and with the loss.

Check yourself 8.6. (b). (a) is M8.8. (c) does not follow: a too-small learning rate shows up as both curves falling slowly. (d) is false: on enough data, as in the main run, both curves fall together.

Exercise 8.1. $-\ln 0.25 = 1.386$. The know-nothing loss is $\ln 12 = 2.485$, so 0.25 on the right word is better: smaller surprise.

Exercise 8.2. $32 \times 64 = 2,048$ exercises. The causal mask (§3.6, Step 3) lets every position be an honest exercise in the same pass.

Exercise 8.3. Step = learning rate $\times$ slope = $4000 \times 0.0085 \approx 34.1$ degrees, from 90° down to about 55.9° , as in the text.

Exercise 8.4. Measure the attention on a different sentence that also needs **waste** (for example **waste of money**), or measure loss on held-out exercises. A model that solved one exercise by memorising fails both checks.

Exercise 8.5. Any three of: stop at the step with the best held-out loss (assumes later steps only memorise); add more, varied training text (assumes too little data); use a smaller model (assumes too many knobs for the data); lower the learning rate or add regularisation (assumes the walker is jumping into a memorising valley). A good answer says which assumption it is testing.

What you now know.

- The loss is the average surprise $-\ln p$ at the right next word; it rewards moving probability towards the right word, and logs turn the chain of Chapter 1 into a sum.
- Thanks to the mask, one pass gives an exercise at every position.
- Training is walking downhill on the loss landscape, with a learning rate that must be neither too small nor too large.
- Trained only on surprise, the toy's query turned from 95.9° to 9.4° off **waste**'s key: the arrows of Chapter 3 really do turn and stretch.
- Real steps use noisy batches; Adam smooths and evens out the steps.
- A real tiny transformer learned word-like text from my transcripts, and a held-out curve is the only way to see overfitting.

9 Inference: the KV cache and its bill

Map of this chapter. Up to now we have trained the model to answer T exercises in one pass. Using it is a different job: the reply is written one token at a time, and each new token has to wait for the one before it. This chapter asks what that loop costs. We find that most of the work in it is repeated, we remove the repetition with the *KV cache*, and we prove on the toy that the answer does not change by a single digit. Then we count the bill the cache brings with it, in real gigabytes for GPT-2 and Qwen3, and look at two ideas that shrink it: sharing keys and values between heads, and never writing the big score table to memory. It ends with you able to price the memory of any model's conversation from its configuration.

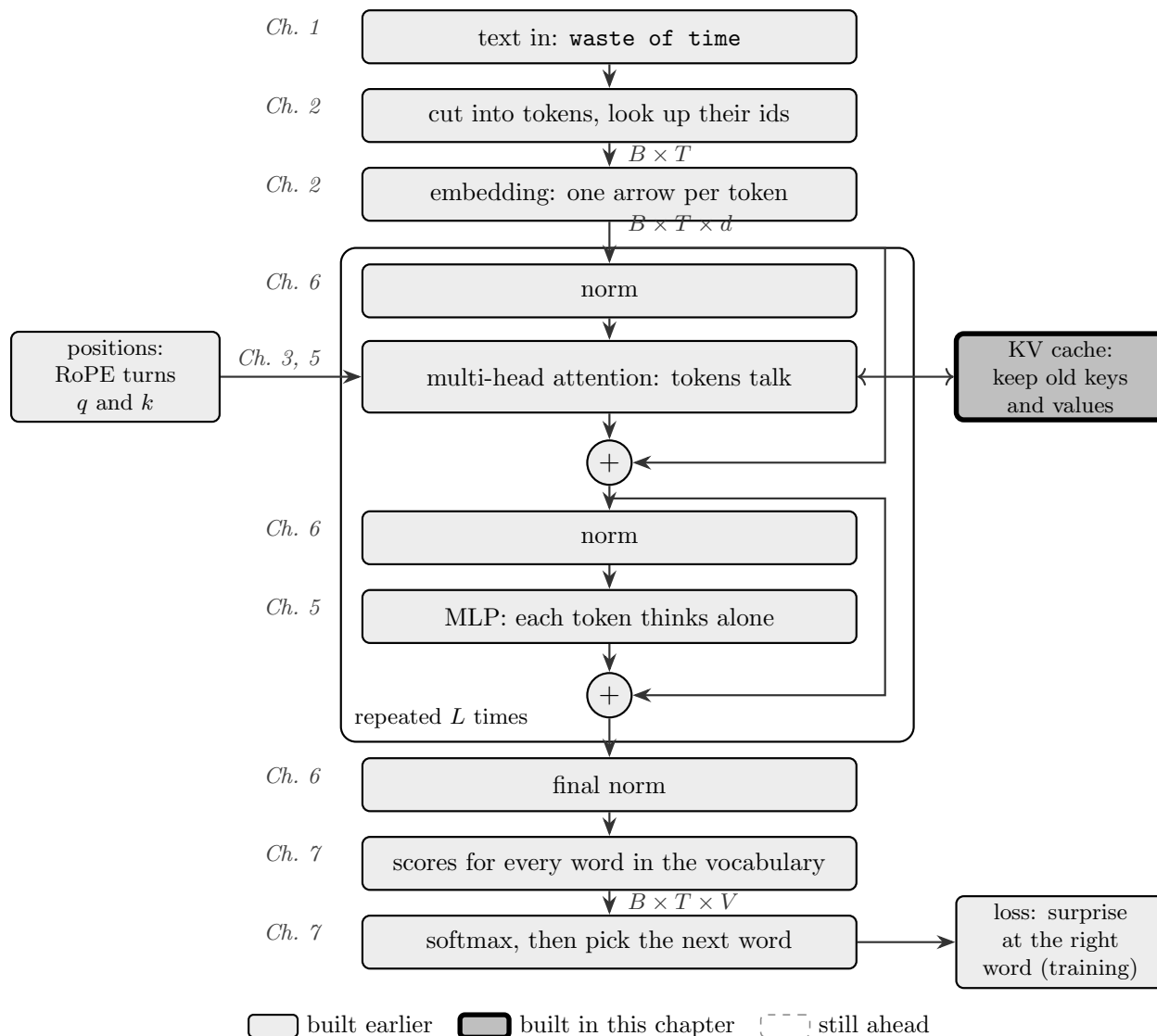


Figure 55: Where we are in the machine. You are here: the *KV cache* box, on the right of attention. It is not a new layer and it changes no answer. It is a store next to every attention block that keeps the keys and values of the tokens already read, so that writing the next token does not redo their work.

Before you start.

1. How many bytes are in a kibibyte (KiB), a mebibyte (MiB) and a gibibyte (GiB)?
2. A number stored in “fp16” takes 16 bits. How many bytes is that?
3. What is 12×64 ? What is $2 \times 28 \times 8 \times 128$?
4. From Chapter 3: which of Q , K , V does a token use to *ask*, and which two does it use to *be found* and to *hand over*?

Refreshers. (1) Computers count memory in powers of two: 1 KiB = 1024 bytes, 1 MiB = 1024 KiB, 1 GiB = 1024 MiB = $2^{30} \approx 1.07$ billion bytes. (2) 8 bits make a byte, so fp16 is 2 bytes. (3) 768 and 57,344. (4) The query asks; the key is what it can be found by; the value is what it hands over. If any of these felt shaky, reread §3.5 before going on.

9.1 Back to the loop

Chapter 1 drew the loop that writes a reply: read the words so far, guess the next word, append it, repeat. Our customer wrote **waste of time**. To write *Sorry, this has not been useful* the model goes round the loop 7 times, and after the last turn the sequence is 10 tokens long.

Predict first. When the model writes the 8th token, it has already processed the first 7 tokens on earlier turns. Tell me: how much of the work for those 7 tokens do you expect to be done *again*? None, a little, or almost all of it? Write down your guess before reading on.

PICTURE IT. Imagine keeping a diary where, every evening, before writing today’s line, you copy out every earlier page by hand and then add one sentence. Nothing on the old pages has changed. You are paying again for work you already did.

In one sentence: Writing is a loop, one token per turn, and each turn sees a longer sequence than the last.

9.2 What gets repeated

Let us count it on the toy instead of guessing. Take the attention of Chapter 3 and suppose that on every turn we run it on the whole sequence so far, exactly as during training. For a sequence of t tokens that means making t queries, t keys and t values ($3 \times t \times d \times d_k$ multiplications), then the $t \times t$ score table, then mixing the values ($2 \times t \times t \times d_k$ more).

Now look at what is new on each turn. Only one token arrived. Its key and value are new, and its question is new. Every earlier token’s key and value are the same numbers they were last turn, because a token’s key is $\mathbf{x}W_K$ and neither its arrow nor W_K has changed. And thanks to the causal mask (§3.6, Step 3) the earlier tokens’ outputs cannot change either: they are not allowed to look at the newcomer.

length t	newest token	multiplications, recompute all	multiplications, only the new token
4	Sorry	320	80
5	,	440	88
6	this	576	96
7	has	728	104
8	not	896	112
9	been	1080	120
10	useful	1280	128

Over the 7 turns (plus reading the prompt once) that is 5,536 multiplications for recomputing everything against 944 if we only compute what is new: 5.9 times less work, on a reply of seven tokens. Notice how the left column grows: the score table part grows with t^2 , so for a reply of a thousand tokens the waste is far worse.

A tempting wrong idea (M9.1). “Writing each new token means running the whole sentence through the model again.” It is tempting because the model does *read* its whole context every turn. But reading is not recomputing. The new token needs the old tokens’ keys and values, and those were already computed on earlier turns. What has to be recomputed is only what depends on the new token.

In one sentence: On every turn, everything about the old tokens is the same as last turn; only the newest token brings new work.

9.3 The KV cache

PICTURE IT. A librarian files a card for each book as it arrives: its label and a summary. When a new reader comes with a question, the librarian does not reprint every card; they read through the existing cards with the new question. Each new book adds one card.

SAY IT EXACTLY. Keep two growing tables, K_{cache} and V_{cache} , one row per token so far. On each turn, for the newest token $\mathbf{x}_t$:

$$\mathbf{q}_t = \mathbf{x}_t W_Q, \quad K_{\text{cache}} \leftarrow \begin{bmatrix} K_{\text{cache}} \\ \mathbf{x}_t W_K \end{bmatrix}, \quad V_{\text{cache}} \leftarrow \begin{bmatrix} V_{\text{cache}} \\ \mathbf{x}_t W_V \end{bmatrix} \quad [1 \times d_k, t \times d_k, t \times d_k]$$

$$\mathbf{o}_t = \text{softmax} \left(\frac{\mathbf{q}_t K_{\text{cache}}^\top}{\sqrt{d_k}} \right) V_{\text{cache}} \quad [(1 \times t)(t \times d_k) = 1 \times d_k]$$

One query row meets every cached key: one row of the tournament table, not the whole table. Figure 56 shows the two tables on our toy as the reply starts. Each turn adds one row at the bottom (highlighted) and leaves every row above it untouched.

Worked example: the turn that reads Sorry. The prompt **waste of time** has been read, so the cache holds the three keys and values of Chapter 3. Now **Sorry** arrives, with arrow (1, 1, 0, 0).

Predict first. Before computing: will **Sorry**’s query, (1, 1, 0, 0), agree more with the key of **waste**, (0, 0, 2, 2), or with the key of **of**, (1, 0, 0, 0)? Look at where each has its non-zero numbers.

1. New rows. Key (1, 0, 1, 1), value (1, 0, 0, 1), query (1, 1, 0, 0). Only these three are computed for this turn.
2. Append the key and value to the cache, which now has 4 rows.
3. Scores: one query row meets the 4 cached keys, stood up as columns ($K_{\text{cache}}^\top$). The picture below works out the highlighted cell; the other three are the same row meeting a different column. The scores are (0, 1, 1, 1).
4. Divide by $\sqrt{4} = 2$ and apply softmax: shares (0.168, 0.277, 0.277, 0.277).
5. Mix the cached values with these shares (second picture): $\mathbf{o}_{\text{Sorry}} = (0.445, 0.445, 0.277, 0.555)$.

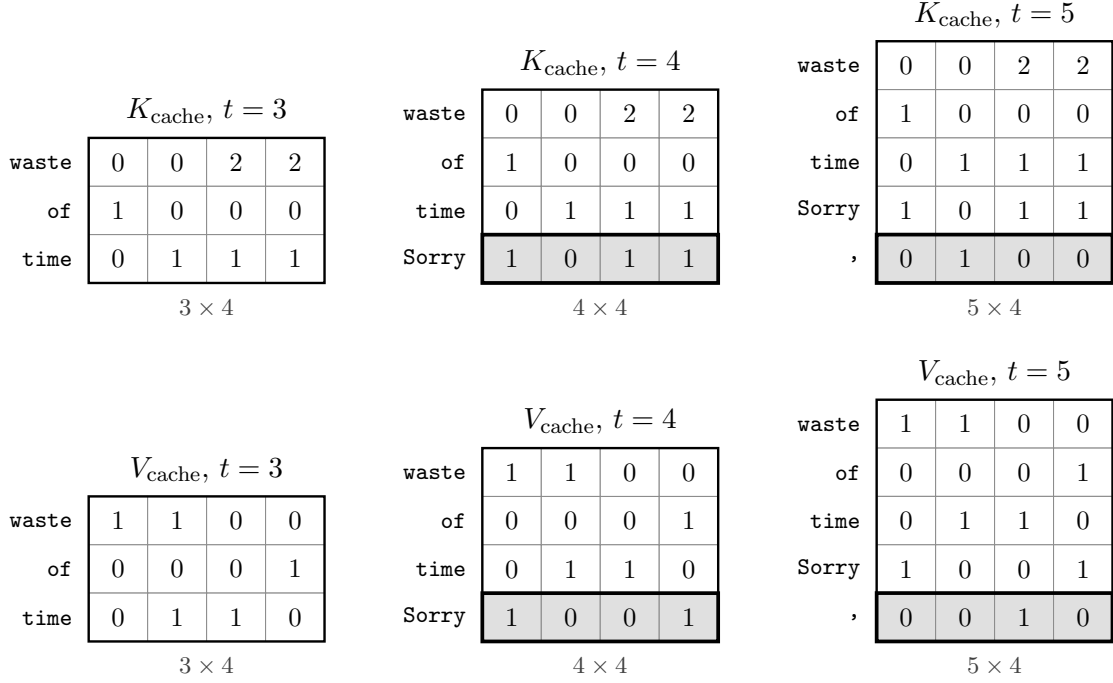
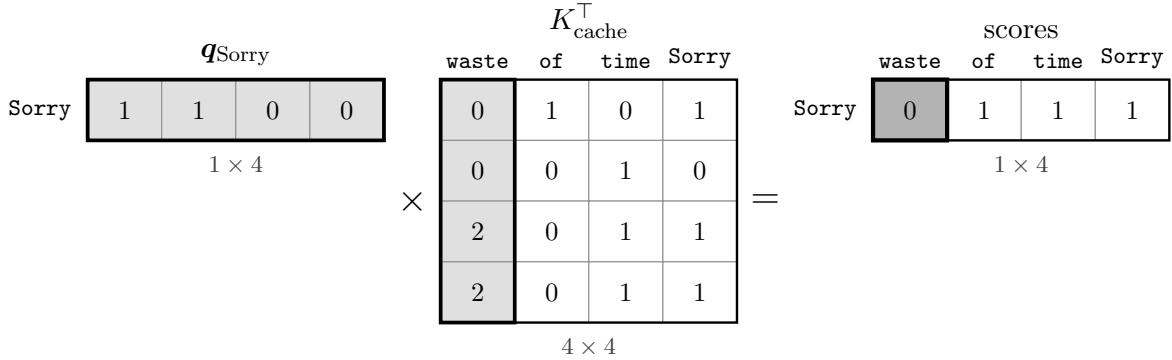
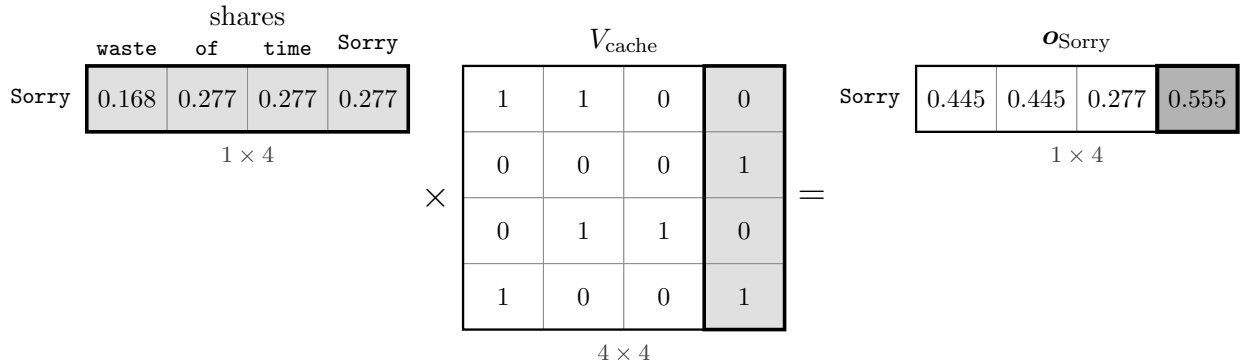


Figure 56: The KV cache growing. After the prompt the cache holds 3 rows; reading **Sorry** adds a fourth, reading the comma a fifth. Rows are only ever added, never changed, because a token's key $\mathbf{x}W_K$ and value $\mathbf{x}W_V$ depend on nothing but that token.



highlighted row (**Sorry's** query) $\cdot$ highlighted column (**waste's** key): $(1)(0) + (1)(0) + (0)(2) + (0)(2) = 0$



highlighted row (the shares) $\cdot$ highlighted column (column 4 of V_{cache}):
 $(0.168)(0) + (0.277)(1) + (0.277)(0) + (0.277)(1) = \mathbf{0.555}$

Figure 57: The turn that reads **Sorry**, as two products. Top: the one new query times the transposed key cache gives one row of 4 scores. Bottom: the 4 shares times the value cache give the output. Shapes: $(1 \times 4)(4 \times 4) = 1 \times 4$ both times. Nothing here is 4×4 work on the old tokens.

waste gets the smallest share here: **Sorry**’s query points along the first two directions, where **waste**’s key has nothing. (Did your prediction hold?)

Sanity check. We ran the whole reply both ways: the full recompute of Chapter 3 on every turn, and the cache. The largest difference between any output number of the two methods, over all 7 turns, was 0. Not small: zero. The cache is not an approximation. It does the same arithmetic, and skips only arithmetic whose result it already has.

WHERE THIS PICTURE BREAKS. A librarian’s card can go out of date if the book is edited. A cached key cannot: once a token is written, its arrow and the weights never change during the reply. The picture breaks the other way too: the cards take up shelf space, and in §9.5 that shelf turns out to be the main cost.

WHY IT HAD TO BE THIS WAY. Why cache keys and values, and not queries? A query is only used on the turn its token arrives, to ask its one question. Keys and values are used by every *later* token. What is reused is worth storing; what is used once is not. Remember the question from Chapter 3, why does **time** need a key and a value of its own? Here is the second half of the answer: so that every token after it can find it and read it.

A tempting wrong idea (M9.3). “The cache must also store the queries.” It is tempting because Q , K and V are made together. But look at the formula: old queries never appear in it. The only query on a turn is the newest one.

Check yourself 9.1. After the cache is added, the toy’s outputs change from (0.665, 0.910, 0.245, 0.090) to something else for **time**. What went wrong, if anything?

- (a) Nothing; a cache always changes outputs a little.
- (b) Something is broken: with a correct cache the outputs are identical.
- (c) The cache was missing the queries.
- (d) The prompt is too short for a cache to be accurate.

In one sentence: The KV cache keeps every earlier token’s key and value, so each turn computes one new query, key and value and one row of scores, and gets exactly the same answer.

9.4 Two phases: prefill and decode

Generation has two phases with very different shapes.

PICTURE IT. A shop opening for the day. In the morning delivery (*prefill*) a whole lorry of boxes is unloaded at once, many hands working in parallel. After that, customers come one at a time (*decode*), and each one needs the shopkeeper to walk the whole shop.

SAY IT EXACTLY. *Prefill* reads the prompt: all prompt tokens go through the model together, as in training, and their keys and values fill the cache. This is big matrix times big matrix, the work GPUs love (§3.9). *Decode* writes the reply: one token per turn, so every weight matrix is multiplied by a single row. For each new token the hardware must read *all* the weights and the whole cache from memory, and do only a little arithmetic with each number it reads.

How little? Take Qwen3-0.6B (Chapter 10). One decode step does about $2N \approx 1.19$ billion multiply-adds and must read the weights, about 1.11 GiB in fp16. That is roughly 1.0 arithmetic operation per byte read. A modern GPU can do dozens to hundreds of operations in the time

it reads one byte from its main memory. So during decode the arithmetic units mostly wait for memory. Decode speed is set by how many bytes must be read per token, and the cache is part of those bytes.

A tempting wrong idea (M9.4). “What a model costs to run is set by its weights alone; context length is free.” The number in a model’s name is its weight count, so it is the number everyone quotes. You raised the right question in the session: “should it run in gpu, cpu, mobile phone?” (S9). The answer depends on the weights *and* on how many tokens of conversation it must hold, as the next section shows.

In one sentence: Prefill is parallel and arithmetic-heavy; decode goes one token at a time and is limited by how many bytes it must read, cache included.

9.5 The bill

Predict first. A model has 28 layers. Each layer keeps a key and a value per token, for 8 heads of width 128, each number in 2 bytes. How much memory for one token of conversation? Guess an order of magnitude: bytes, kilobytes, megabytes?

SAY IT EXACTLY. For every token, every layer stores one key and one value per key-value head:

$$\text{KV bytes} = 2 \times \text{layers} \times \text{kv heads} \times \text{head width} \times \text{tokens} \times \text{bytes per number}.$$

The 2 counts K and V. Drawn as a product, for Qwen3-0.6B:

$$\begin{array}{ccccccccc} \boxed{2} & \times & \boxed{28} & \times & \boxed{8} & \times & \boxed{128} & \times & \boxed{2} & = & \boxed{114,688 \text{ bytes per token}} \\ \text{K and V} & & \text{layers} & & \text{key-value} & & \text{numbers} & & \text{bytes per} & & \\ & & & & \text{heads} & & \text{per head} & & \text{number} & & \\ & & & & & & & & \text{(fp16)} & & \end{array}$$

Figure 58: The KV-cache bill for one token of conversation in Qwen3-0.6B. Change any one box and the bill changes in proportion; §9.6 shrinks the third box.

For Qwen3-0.6B that is $2 \times 28 \times 8 \times 128 \times 2 = 114,688$ bytes, about 112 KiB *per token*. GPT-2 small (12 layers, 12 heads of 64) needs 36,864 bytes, 36 KiB. Our script computes the same formula as the calculator `code/calc.py`:

model	weights (fp16)	cache, 1,024 tokens	8,192 tokens	32,768 tokens
GPT-2 small	0.23 GiB	0.04 GiB	0.28 GiB*	1.12 GiB*
Qwen3-0.6B	1.11 GiB	0.11 GiB	0.88 GiB	3.50 GiB

*GPT-2 cannot read past 1,024 tokens; these columns show what the same design would cost.

Read the Qwen row. At about 10,395 tokens of conversation, the cache is as large as the whole model. At 32,768 tokens it is three times the model. And this is for *one* conversation: a server answering 16 such conversations at once needs 56.0 GiB of cache. For long conversations, the bill is the cache.

WHY IT HAD TO BE THIS WAY. The cache trades memory for arithmetic. That trade was a bargain in §9.2, and it stays a bargain, but memory is the scarce thing on a GPU and the cache grows with

every token. So the next ideas all attack the same formula: fewer layers, fewer key-value heads, narrower heads, or fewer bytes per number. Which of those would you touch first, and what would each cost you?

Check yourself 9.2. You double the context from 8,192 to 16,384 tokens. What happens to Qwen3-0.6B’s cache?

- (a) It doubles.
- (b) It quadruples, because attention is quadratic in length.
- (c) It stays the same; the cache size is fixed by the weights.
- (d) It grows a little, because only new tokens are added.

In one sentence: The cache costs $2 \times \text{layers} \times \text{kv heads} \times \text{head width} \times \text{bytes per token}$, and for long conversations it outgrows the model itself.

9.6 Paying less: sharing keys and values between heads

Chapter 5 gave each attention head its own queries, keys and values. In the formula above, the number of *key-value* heads is what matters. What if several query heads shared one set of keys and values?

PICTURE IT. A class of 16 students each asking their own question, but sharing 8 textbooks instead of carrying 16. Every student still asks what they want; they read the answers from shared books.

SAY IT EXACTLY. *Multi-query attention* (Shazeer, 2019) keeps all the query heads but gives the whole layer a single key head and a single value head. *Grouped-query attention* (Ainslie et al., 2023) sits in between: query heads are split into groups, and each group shares one key head and one value head. Qwen3-0.6B uses 16 query heads and 8 key-value heads, so each pair of query heads shares its keys and values. For 32,768 tokens:

Qwen3-0.6B with ...	key-value heads	cache at 32,768 tokens
full multi-head attention	16	7.00 GiB
grouped-query attention (what it uses)	8	3.50 GiB
multi-query attention	1	0.44 GiB

WHERE THIS PICTURE BREAKS. Students sharing a book all read the same pages. Heads sharing keys and values all see the same “labels” and the same “contents” for every token; only their questions differ. Something is lost: fewer independent ways to describe each token. Ainslie and colleagues found that grouping kept quality close to full multi-head attention while multi-query lost more; how much is lost is an empirical question for each model, not a law.

A tempting wrong idea (M9.5). “Fewer key-value heads means the model looks at fewer things.” The number of *ways of asking* is the number of query heads, and that is unchanged. What shrinks is the number of independent ways each token can be found and read.

In one sentence: Grouped-query attention lets several query heads share one key and value head, halving Qwen3’s cache compared with full multi-head attention at little cost in quality.

9.7 Never writing the big table: FlashAttention

The cache is about memory that *stays*. There is a second memory cost, during prefill and training: the $T \times T$ score table of §3.4. At 32,768 tokens, one head in one layer has 1.07 billion scores, 2.0 GiB in fp16, before softmax has even been applied.

PICTURE IT. You need the total of a long column of numbers written in a ledger in the basement. One way: carry each page upstairs, copy it into a giant table on your desk, then add. Another: go down once, add each page as you read it, and carry up only the running total.

SAY IT EXACTLY. FlashAttention (Dao et al., 2022) computes attention in tiles: a block of queries meets a block of keys inside the GPU’s small fast memory, the softmax is kept as a running sum, and the full $T \times T$ table is never written to the large slow memory. The arithmetic is the same and the result is exact (up to floating-point rounding). What changes is the number of bytes moved, and in §9.4 we saw that bytes moved are what limit speed.

In one sentence: FlashAttention gets the same answer while never writing the full score table to slow memory.

Abstraction — pay once, reuse; and move fewer bytes. The KV cache is *memoisation*: store a result that will be asked for again, so you pay for it once. It works here for a precise reason: the causal mask guarantees that old results never change. Before you cache anything in a system you build, ask the same question: what guarantees the stored answer is still right? Agent platforms use the same trade when they cache a long system prompt across calls, and a memory system makes it when it stores a summary instead of rereading a whole conversation. The second lesson, from §9.4 and §9.7, transfers too: when a system is slow, count the bytes it moves before counting the arithmetic it does.

Looking back.

1. (Chapter 3) Why does the causal mask guarantee that earlier tokens’ outputs do not change when a new token is appended?
2. (Chapter 1) Hallucination was described as a consequence of writing one token at a time. Does the KV cache make it more likely, less likely, or neither? Why?

Exercises

Exercise 9.1. Using the formula of §9.5, compute GPT-2 small’s cache for one conversation of 1,024 tokens in fp16. Then compute it again with numbers stored in 1 byte each (8-bit). Check the first against the table.

Exercise 9.2. (Fading worked example.) The turn after **Sorry** reads the token `,` with arrow $(0, 0, 0, 1)$. Fill in the gaps: its key is $(0, 0, 0, 1)W_K = (_, _, _, _)$; the cache now has $_$ rows; its query is $(0, 0, 0, 1)$, so its score against each cached key is that key’s $_$ -th entry.

Exercise 9.3. A model has 32 layers, 32 query heads, 8 key-value heads and head width 128. A product team wants 100 simultaneous conversations of 8,192 tokens in fp16. How much cache memory is that? What would it be with full multi-head attention?

Exercise 9.4. Explain to a friend, in three sentences and without symbols, why decode is limited by memory rather than arithmetic.

Answers to Chapter 9

Check yourself 9.1. (b). With a correct cache the outputs are identical; our script measured a largest difference of 0. (a) is misconception M9.2, the cache as a lossy shortcut. (c) is M9.3: old queries are never used. (d) treats the cache as an approximation that needs enough data; it is not an approximation at all.

Check yourself 9.2. (a). The cache has one row per token per layer per head, so it grows in proportion to length. (b) confuses the cache with the score table, which is $T \times T$ (§9.7). (c) is M9.4. (d) is true per token added but the cache still ends up twice as large.

Looking back. (1) Each earlier token's row of the score table has $-\infty$ in every position to its right, so a new token is invisible to it; its shares and output are unchanged. (2) Neither: the cache gives exactly the same outputs as recomputing, so it changes nothing about what is written.

Exercise 9.1. $2 \times 12 \times 12 \times 64 \times 1,024 \times 2 = 37,748,736$ bytes = 36 MiB, which the table rounds to 0.04 GiB. With 1 byte per number, half: 18 MiB.

Exercise 9.2. Row 4 of W_K is the key: (0, 1, 0, 0). The cache has 5 rows. The query (0, 0, 0, 1) picks the 4th entry of each key. The cached keys are (0, 0, 2, 2), (1, 0, 0, 0), (0, 1, 1, 1), (1, 0, 1, 1) and (0, 1, 0, 0), so the scores are (2, 0, 1, 1, 0).

Exercise 9.3. Per token: $2 \times 32 \times 8 \times 128 \times 2 = 131,072$ bytes = 128 KiB. Times 8,192 tokens and 100 conversations: 107,374,182,400 bytes = 100 GiB. With 32 key-value heads, four times as much: 400 GiB.

Exercise 9.4. One possible answer: to write each word, the computer has to fetch every number in the model and the whole conversation so far from its memory. It then does only about one sum with each number. Its calculators are far faster than its memory, so they spend most of the time waiting for numbers to arrive.

What you now know

- Writing a reply is a loop, and almost all of the work in it would be repeated without a cache.
- The KV cache stores each token's key and value; each turn computes one query and one row of scores, and the outputs are exactly the same.
- Prefill is parallel; decode is one token at a time and limited by bytes read.
- The cache costs $2 \times \text{layers} \times \text{kv heads} \times \text{head width} \times \text{bytes per token}$: 112 KiB for Qwen3-0.6B, larger than the model after about 10,395 tokens.

- Grouped- and multi-query attention shrink the cache by sharing keys and values; FlashAttention avoids writing the score table to slow memory.

10 The toy scaled up: GPT-2, then Qwen3

Map of this chapter. Every chapter so far used a model with arrows of 4 numbers and a vocabulary of 12 tokens, small enough for a pencil. This chapter replays the same machine at real sizes, twice: GPT-2 small (2019), and Qwen3-0.6B (2025). We read each model’s configuration, predict from it how many parameters the model has, and check the prediction against the model itself, built from the same configuration in the reference code library. Then we follow a real batch through Qwen3’s layers by their shapes, count the arithmetic per token, and end with a line-by-line table: what changed from the toy to GPT-2 to Qwen3, and which earlier chapter explains why.

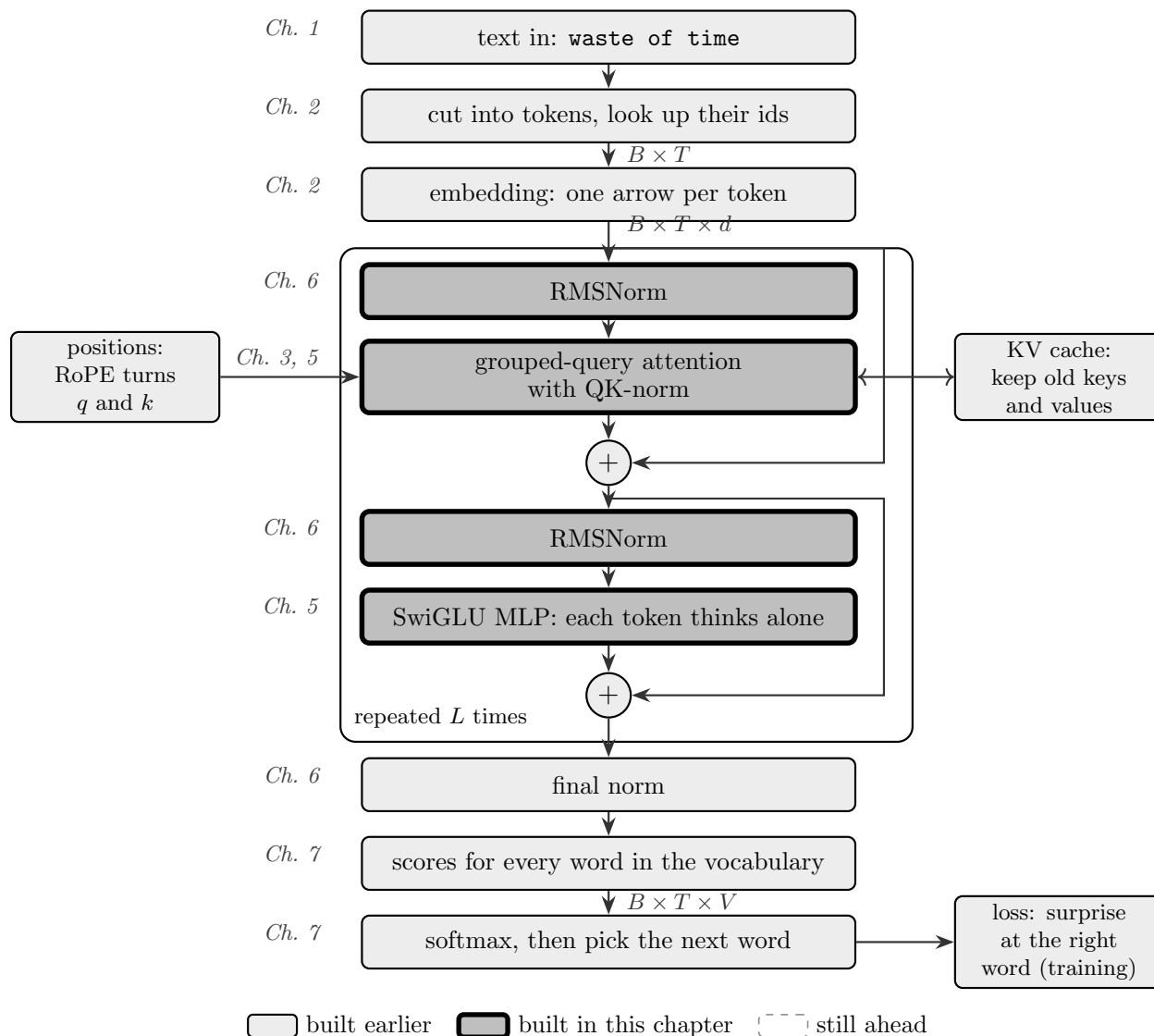


Figure 59: Where we are in the machine. You are here: every block is built. This chapter swaps in the versions real models use, so three blocks are highlighted: the SwiGLU MLP (§10.4), RMSNorm for the norms (Chapter 6) and grouped-query attention with QK-norm (§10.3). The frame repeats 12 times in GPT-2 small and 28 times in Qwen3-0.6B.

Before you start.

1. A matrix has 768 rows and 2,304 columns. How many numbers does it hold?
2. What is 1.5×10^8 written out in full? What is 0.596 billion in millions?
3. From Chapter 3: the query matrix W_Q has shape $d \times d_k$. If $d = 4$ and $d_k = 4$, how many numbers is that?
4. From Chapter 5: what are the two parts of every transformer block?

Refreshers. (1) Rows times columns: 1,769,472. (2) 150,000,000; 596 million. (3) 16. (4) Attention, where tokens talk, and the MLP, where each token thinks on its own; each sits on the residual stream with a normalisation in front.

10.1 Same machine, bigger numbers

In the session on model internals you counted one piece of GPT-2 aloud: “768 mult* 2304 that’s the number of a parameters” (S9). That piece is the matrix that makes queries, keys and values together, 768 inputs to $3 \times 768 = 2,304$ outputs: 1,769,472 numbers. Our toy’s three attention machines held 48.

Predict first. GPT-2 small is advertised as having about 124 million parameters. Where do you think most of them live: in the attention matrices, in the MLPs, or in the embedding table? Write down a rough split, say “40% attention, 40% MLP, 20% embeddings”, before reading §10.4.

PICTURE IT. A city and its model railway. The model railway has the same kinds of things as the city: stations, tracks, signals. The city has thousands of each, and some new kinds of things the model never needed. This chapter is a walk through the city with the model railway in your pocket.

In one sentence: GPT-2 and Qwen3 are the toy with bigger numbers, plus a few changes, each of which an earlier chapter has already motivated.

10.2 Reading the configuration

A model’s *configuration* is a short list of numbers that fixes its shape. Here are the two we use. GPT-2 small’s comes from the reference library’s default GPT-2 configuration, which is GPT-2 small. Qwen3-0.6B’s comes from the model table of the Qwen3 technical report (Yang et al., 2025); it has not yet been checked against the published `config.json` file, which could not be downloaded from where these notes were built.

setting	toy	GPT-2 small	Qwen3-0.6B
vocabulary (rows of the embedding table)	12	50,257	151,936
width d	4	768	1,024
layers (blocks)	1	12	28
query heads $\times$ head width	1×4	12×64	16×128
key-value heads	1	12	8
MLP inner width	—	3,072	3,072
positions	—	learned, 1,024 slots	RoPE, base 1,000,000

The vocabulary. GPT-2’s 50,257 tokens are the 256 possible bytes, 50,000 merges learned by byte-pair encoding, and one end-of-text token (Radford et al., 2019). Qwen3’s embedding table

has 151,936 rows. In the session, the tokenizer was counted at “around 151,669 address” (S9). The two numbers differ by 267.

A tempting wrong idea (M10.3). “The embedding table has exactly one row per tokenizer entry.” It is tempting because each token needs a row. But a table may have spare rows that no token ever uses. Here $151,936 = 128 \times 1,187$: a multiple of 128. Tables are commonly padded to such round sizes because GPU matrix arithmetic runs fastest on them. The spare rows are trained toward never being predicted, since no training text contains them.

In one sentence: A configuration is a handful of numbers that fixes every shape in the model; read it before anything else.

10.3 Heads at scale, and a surprise in the shapes

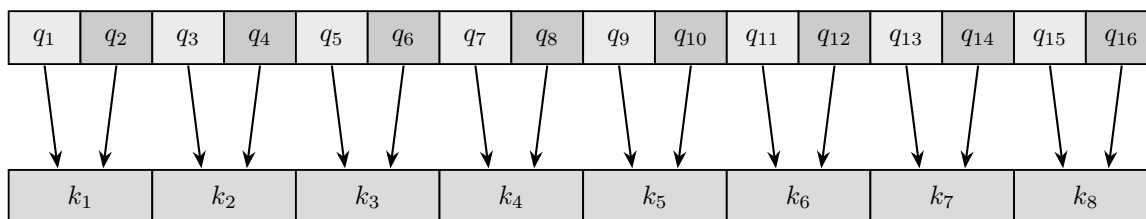
In the toy, one head of width 4 covered the whole width 4. GPT-2 does the same at scale: 12 heads of width 64 cover exactly 768. Now look at Qwen3: 16 query heads of width 128 make 2,048, twice the model’s width of 1,024.

Predict first. If you feed a batch of 2 sentences of 3 tokens into Qwen3, what shape will come out of its query projection: $2 \times 3 \times 1,024$ or $2 \times 3 \times 2,048$? And out of its key projection?

We checked by running a real batch through the Qwen3 architecture (random weights, so the numbers mean nothing, but the shapes are exactly those of the real model). Queries come out $2 \times 3 \times 2,048$, keys $2 \times 3 \times 1,024$, values $2 \times 3 \times 1,024$.

SAY IT EXACTLY. The query projection W_Q has shape $1,024 \times 2,048$: sixteen heads, each asking its question with 128 numbers. The key and value projections are $1,024 \times 1,024$ each: eight heads of 128, the grouped-query sharing of §9.6. After attention, the output projection W_O ($2,048 \times 1,024$) brings the result back to the width of the residual stream, so the block can add it in (Chapter 5).

queries of one token: 16 heads $\times$ 128 numbers = 2,048



keys of one token: 8 heads $\times$ 128 numbers = 1,024 (values are shared the same way)

Figure 60: Grouped-query attention in one Qwen3-0.6B layer, for one token. Follow the arrows: query heads q_1 and q_2 both compare themselves with key head k_1 , q_3 and q_4 with k_2 , and so on. Only the 8 key heads and 8 value heads go into the cache (Chapter 9), half of what 16 would need.

Predict first. Two query heads share one key head. Will they give the same scores? Decide before looking at the toy version below, where the toy’s 4 query numbers are split into two heads of 2, and both heads use the last two columns of the toy’s keys as their one shared key head.

Sharing keys does not make the heads agree, because each head still multiplies the shared keys by its own questions. Head 1’s query for **time** is (0,0): in this head, **time** asks nothing at all. What is shared is the *storage*: one key head to keep in the cache instead of two.

$$\begin{array}{c}
Q^{(1)} \\
\begin{array}{cc}
\text{waste} & \begin{array}{|c|c|} \hline 1 & 0 \\ \hline \end{array} \\
\text{of} & \begin{array}{|c|c|} \hline 0 & 1 \\ \hline \end{array} \\
\text{time} & \begin{array}{|c|c|} \hline 0 & 0 \\ \hline \end{array} \\
3 \times 2
\end{array}
\times
\begin{array}{c}
K_{\text{shared}}^{\top} \\
\begin{array}{ccc}
\text{waste} & \text{of} & \text{time} \\
\begin{array}{|c|c|c|} \hline 2 & 0 & 1 \\ \hline \end{array} \\
2 \times 3
\end{array}
=
\begin{array}{c}
S^{(1)} \\
\begin{array}{ccc}
\text{waste} & \text{of} & \text{time} \\
\begin{array}{|c|c|c|} \hline 2 & 0 & 1 \\ \hline \end{array} \\
3 \times 3
\end{array}
\end{array}$$

highlighted row (time) · highlighted column (waste) = (0)(2) + (0)(2) = 0

$$\begin{array}{c}
Q^{(2)} \\
\begin{array}{cc}
\text{waste} & \begin{array}{|c|c|} \hline 1 & 0 \\ \hline \end{array} \\
\text{of} & \begin{array}{|c|c|} \hline 0 & 0 \\ \hline \end{array} \\
\text{time} & \begin{array}{|c|c|} \hline 1 & 1 \\ \hline \end{array} \\
3 \times 2
\end{array}
\times
\begin{array}{c}
K_{\text{shared}}^{\top} \\
\begin{array}{ccc}
\text{waste} & \text{of} & \text{time} \\
\begin{array}{|c|c|c|} \hline 2 & 0 & 1 \\ \hline \end{array} \\
2 \times 3
\end{array}
=
\begin{array}{c}
S^{(2)} \\
\begin{array}{ccc}
\text{waste} & \text{of} & \text{time} \\
\begin{array}{|c|c|c|} \hline 2 & 0 & 1 \\ \hline \end{array} \\
3 \times 3
\end{array}
\end{array}$$

highlighted row (time) · highlighted column (waste) = (1)(2) + (1)(2) = 4

Figure 61: Two query heads, one shared key head, real toy numbers. Same keys, different questions: for time looking at waste, head 1 scores 0 and head 2 scores 4. Grey cells are hidden by the causal mask.

A tempting wrong idea (M10.6). “Number of heads times head width must equal the model width.” It does in the original transformer and in GPT-2, which is why it looks like a rule. It is a choice. The residual stream must keep width d , and W_O guarantees that whatever the heads produce is brought back to d . Qwen3-0.6B spends more numbers on asking (2,048) than it keeps in the stream (1,024).

Two more changes inside attention. Qwen3 normalises every query and every key before the dot product (QK -norm; Henry et al., 2020; Yang et al., 2025). Remember the two knobs of §3.2: direction and length. QK -norm fixes the length knob, so scores cannot grow without limit as training goes on; the model has to express agreement mostly through direction. And positions enter by rotating queries and keys (RoPE, Chapter 4) instead of GPT-2’s learned table of 1,024 position vectors.

A tempting wrong idea (M10.2). “More attention heads always help.” As one of my live sessions put it, “it is not always it helps” (S9). Heads split a fixed budget of numbers into more, narrower questions. Narrower heads describe each token with fewer numbers. Qwen3 chose 16 query heads but only 8 key-value heads; the right count is something to test, not assume (Chapter 11).

Check yourself 10.1. In Qwen3-0.6B, what shape is the output projection W_O of one layer?

- (a) $1,024 \times 1,024$, because heads times width must equal d .
- (b) $2,048 \times 1,024$, from the 16 query heads back to the stream width.

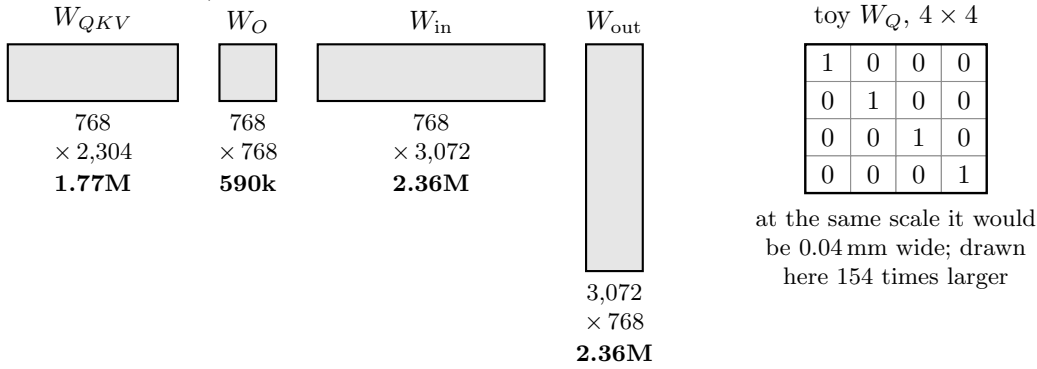
- (c) $1,024 \times 2,048$, because outputs are wider than inputs.
 (d) It has no W_O ; the heads are added straight into the stream.

In one sentence: Qwen3 asks with 16 heads of 128 numbers, shares 8 key-value heads, fixes query and key lengths, and rotates them for position; W_O brings everything back to the stream width.

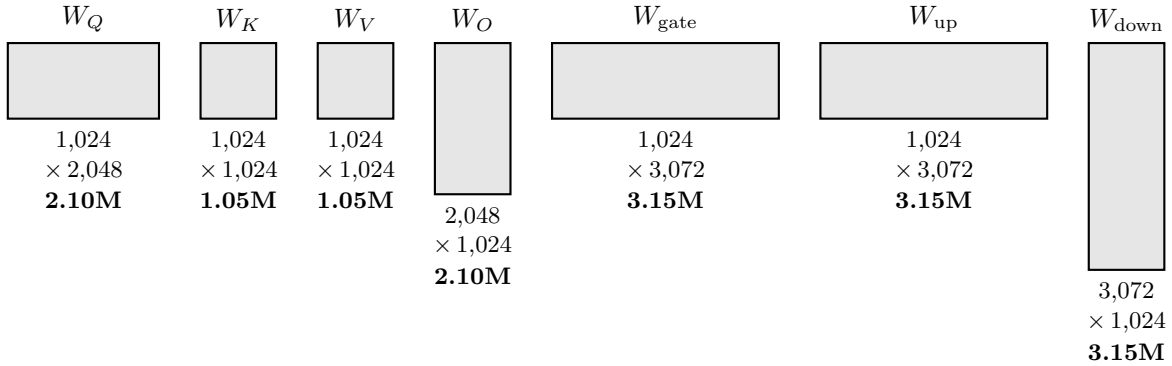
10.4 Counting every parameter, then checking

Before counting, look. Figure 62 draws every weight matrix of one block to the same scale, with the shapes read from the reference code library’s own model classes. Each rectangle has one row per input number and one column per output number, so its area is its parameter count.

GPT-2 small, one block



Qwen3-0.6B, one block



— this bar is 1,024 rows or columns long

Figure 62: One block’s weight matrices, drawn to scale (the bar at the bottom left is 1,024 rows or columns long). Under each: rows $\times$ columns and the count in millions (M). Notice how much of the area the three MLP matrices on the right take, and that the toy’s whole query matrix would be a dot.

Now count. Each component’s count follows from its shape:

- embedding table: vocabulary $\times d$; GPT-2 adds a position table, $1,024 \times d$;
- attention: W_Q , W_K , W_V , W_O (GPT-2 also has a bias for every output number); Qwen3 adds two QK-norm scales of 128;
- MLP: GPT-2 has two matrices, $d \times 3,072$ and $3,072 \times d$, plus biases; Qwen3’s SwiGLU (Shazeer, 2020) has three, a gate, an up and a down matrix;

- norms: a LayerNorm has a scale and a shift for each of the d numbers ($2d$); an RMSNorm has only the scale (d). Two per block, one at the end;
- unembedding: both models reuse the embedding table (*tied* weights, Chapter 7), so it adds nothing.

SwiGLU on the toy. Qwen3’s MLP has three matrices where GPT-2’s has two. Two of them read the same input: W_{up} proposes values, and W_{gate} decides, entry by entry, how much of each value gets through.

Predict first. If a gate entry is negative, does its value get through, get flipped, or get mostly stopped? Below, `time`’s attention output from Chapter 3, $\mathbf{x} = (0.665, 0.910, 0.245, 0.090)$, goes through a pencil-sized SwiGLU (inner width 4, weights chosen by hand).

Sanity check. The third gate entry is negative, and its entry of $\mathbf{h}$ comes out small and of the opposite sign. A gate is a dial on each value, which a plain MLP (Chapter 5) does not have; that is the whole idea, at the cost of a third matrix.

component	GPT-2 small	Qwen3-0.6B
token embeddings	38,597,376	155,582,464
position embeddings	786,432	0
attention, per block	2,362,368	6,291,712
MLP, per block	4,722,432	9,437,184
norms, per block	3,072	2,048
one block	7,087,872	15,730,944
all blocks	85,054,464	440,466,432
final norm	1,536	1,024
total, by hand	124,439,808	596,049,920

Sanity check. We then built both models in the reference library from the same configurations and asked them to count their own parameters. GPT-2 small reports 124,439,808; Qwen3-0.6B reports 596,049,920. Both match the hand count to the last digit. If your formula and the real model disagree, the disagreement tells you exactly which component you misunderstood.

Now check your prediction from §10.1. The MLP holds 67% of each GPT-2 block and 60% of each Qwen3 block. The embedding tables hold 32% of GPT-2 and 26% of Qwen3.

A tempting wrong idea (M10.4). “Most of a transformer’s parameters sit in attention.” Attention is the famous part, and the chapter titles make it feel central. By count, the MLPs are larger: in both models, about two thirds or more of each block. Where the *talking* happens and where the *numbers* are stored are different places.

WHERE THIS PICTURE BREAKS. The model railway in your pocket matched the city in kind. It did not prepare you for proportions: in the toy, attention looked like most of the machine because the toy had little else. Scaling up changes which part is big.

In one sentence: Every parameter follows from a shape in the configuration, and the hand count matches the real models exactly: 124,439,808 and 596,049,920.

$$\begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \mathbf{x} \\ \hline 0.665 & 0.910 & 0.245 & 0.090 \\ \hline \end{array} \begin{array}{c} 1 \times 4 \end{array} \times \begin{array}{|c|c|c|c|} \hline W_{\text{gate}} \\ \hline 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ \hline \end{array} \begin{array}{c} 4 \times 4 \end{array} = \begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \mathbf{g} \\ \hline 0.910 & 0.820 & -0.665 & -0.820 \\ \hline \end{array} \begin{array}{c} 1 \times 4 \end{array}$$

$$\text{highlighted row} \cdot \text{highlighted column} = (0.665)(-1) + (0.910)(0) + (0.245)(0) + (0.090)(0) = -0.665$$

$$\begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \mathbf{x} \\ \hline 0.665 & 0.910 & 0.245 & 0.090 \\ \hline \end{array} \begin{array}{c} 1 \times 4 \end{array} \times \begin{array}{|c|c|c|c|} \hline W_{\text{up}} \\ \hline 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ \hline \end{array} \begin{array}{c} 4 \times 4 \end{array} = \begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \mathbf{u} \\ \hline 0.910 & 0.665 & 0.335 & 0.245 \\ \hline \end{array} \begin{array}{c} 1 \times 4 \end{array}$$

$$\text{highlighted row} \cdot \text{highlighted column} = (0.665)(0) + (0.910)(0) + (0.245)(1) + (0.090)(1) = 0.335$$

$$\begin{array}{|c|c|c|c|} \hline \text{SiLU}(\mathbf{g}) \\ \hline 0.649 & 0.569 & -0.226 & -0.251 \\ \hline \end{array} \odot \begin{array}{|c|c|c|c|} \hline \mathbf{u} \\ \hline 0.910 & 0.665 & 0.335 & 0.245 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline \mathbf{h} \\ \hline 0.590 & 0.379 & -0.076 & -0.061 \\ \hline \end{array}$$

entry by entry, e.g. third entry: $\text{SiLU}(-0.665) \times 0.335 = -0.226 \times 0.335 = -0.076$; a negative gate shrinks its entry

$$\begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \mathbf{h} \\ \hline 0.590 & 0.379 & -0.076 & -0.061 \\ \hline \end{array} \begin{array}{c} 1 \times 4 \end{array} \times \begin{array}{|c|c|c|c|} \hline W_{\text{down}} \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ \hline \end{array} \begin{array}{c} 4 \times 4 \end{array} = \begin{array}{c} \text{time} \end{array} \begin{array}{|c|c|c|c|} \hline \text{MLP output} \\ \hline 0.529 & 0.379 & 0.303 & -0.061 \\ \hline \end{array} \begin{array}{c} 1 \times 4 \end{array}$$

$$\text{highlighted row} \cdot \text{highlighted column} = (0.590)(0) + (0.379)(1) + (-0.076)(0) + (-0.061)(0) = 0.379$$

Figure 63: SwiGLU with real numbers. Gate and up are two matrix products of the same vector; $\text{SiLU}(g) = g/(1 + e^{-g})$ squashes the gate (close to g when g is large, close to 0 when g is very negative); the two are multiplied entry by entry ($\odot$); W_{down} brings the result back to width d . Output: $(0.529, 0.379, 0.303, -0.061)$.

10.5 What “0.6B” means

Qwen3-0.6B has 0.596 billion parameters, which rounds to the 0.6 in its name. Of those, 0.440 billion sit outside the embedding table; this is the number scaling studies usually count (Kaplan et al., 2020), because the embedding table is a lookup, not arithmetic done on every token.

A tempting wrong idea (M10.5). “A model with billions of weights will overfit.” In the session the worry came up as “billions of weights means too much” (S9), and you answered it there: it depends on the data. Classic machine learning taught that more parameters than examples means memorising. Count the examples here. The Qwen3 report states about 36 trillion pre-training tokens, and each token is one next-word exercise (§1.2). That is about 60,397 exercises per parameter. For comparison, the Chinchilla study (Hoffmann et al., 2022) found about 20 tokens per parameter to be the best use of a fixed training budget; Chapter 11 returns to why a small model might be trained far past that.

In one sentence: The name counts all parameters; the number that matters for arithmetic is the non-embedding count, 0.440 billion.

10.6 Following a batch through Qwen3

Chapter 1 drew the forward pass as an assembly line of shapes. Here are the real shapes, recorded by hooks on each stage of the Qwen3 architecture for a batch of $B = 2$ sentences of $T = 3$ tokens:

stage	shape	chapter
token ids	2×3	2
embeddings	$2 \times 3 \times 1,024$	2
queries / keys / values	$2 \times 3 \times 2,048$ / $2 \times 3 \times 1,024$ / $2 \times 3 \times 1,024$	3, 5, 9
attention output after W_O	$2 \times 3 \times 1,024$	5
MLP gate (inner width)	$2 \times 3 \times 3,072$	5
MLP output	$2 \times 3 \times 1,024$	5
after one whole block	$2 \times 3 \times 1,024$	5, 6
after the final norm	$2 \times 3 \times 1,024$	6
scores for every word	$2 \times 3 \times 151,936$	7

Notice the rhythm: the stream stays $B \times T \times 1,024$ from the first block to the last, and only reaches out to other widths inside a block, for asking (2,048), for thinking (3,072), and once at the very end, for scoring every word (151,936). That last shape is the $B \times T \times V$ at the end of the forward-pass figure in §1.9: a guess at every position of every sentence, in one pass.

The same shape at pencil size is the batch of Chapter 3: two sentences of three tokens, each token an arrow of 4 numbers. Every tensor in Figure 64 is this, with 1,024 (or more) columns instead of 4.

		0	1	0	1	← not very useful
waste	1	0	1	0	0	
of	0	1	0	0	0	
time	0	0	1	1		← waste of time

$X: B \times T \times d = 2 \times 3 \times 4$

In one sentence: A real batch keeps the shape $B \times T \times d$ on the residual stream and widens only inside attention, inside the MLP, and at the final scores.

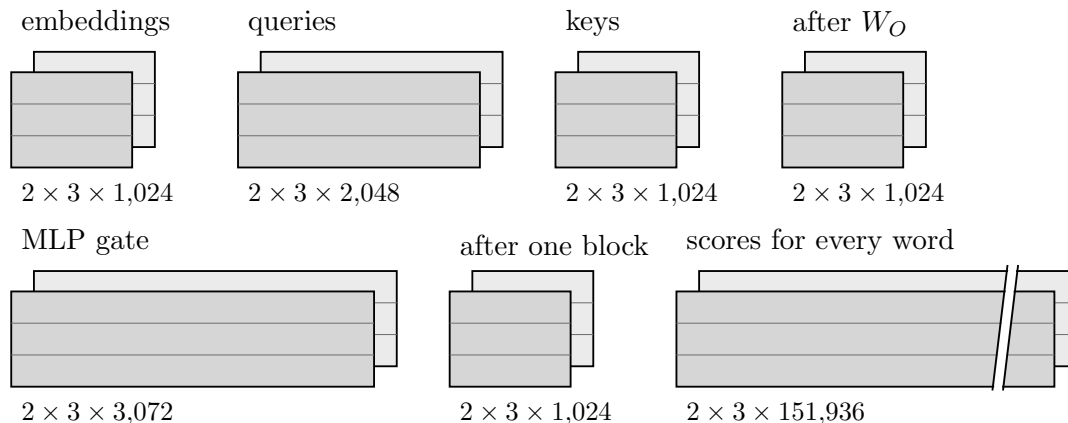


Figure 64: The tensors of a real batch inside Qwen3-0.6B, drawn from the recorded shapes. Each sheet is one sentence ($B = 2$), each band one token ($T = 3$), and the length is proportional to the width of each token’s arrow. The stream (embeddings, after W_O , after one block) never changes shape; queries and the MLP gate are wider; the scores, 151,936 per token, are broken off, since at this scale they would be over two metres long.

10.7 The arithmetic per token

PICTURE IT. Every parameter in a matrix is used once for every token that passes: one multiplication and one addition. So the work per token is about twice the number of parameters it passes through.

SAY IT EXACTLY. Forward work per token is about $2N$ operations, with N the non-embedding parameters, plus the attention scores, $2 \times \text{layers} \times \text{context} \times \text{heads} \times \text{head width}$ (Kaplan et al., 2020). Training costs about three times the forward pass (the backward pass of Appendix A does about twice the forward work), so about $6N$ per token.

model, context	$2N$ part	attention part	forward, per token
GPT-2 small, 1,024	170 million	19 million	189 million
Qwen3-0.6B, 4,096	0.88 billion	0.47 billion	1.35 billion
Qwen3-0.6B, 32,768	0.88 billion	3.76 billion	4.64 billion

At long context, the attention part overtakes the weights: the price of letting every token look at every earlier token (Requirement 3 of Chapter 1). And training: 6×0.440 billion parameters $\times$ 36 trillion tokens $\approx 9.5 \times 10^{22}$ operations.

Check yourself 10.2. Doubling the context length of Qwen3-0.6B from 4,096 to 8,192 tokens roughly does what to the forward work per token?

- (a) Doubles all of it.
- (b) Leaves the $2N$ part unchanged and doubles the attention part.
- (c) Leaves it unchanged; work depends only on parameters.
- (d) Quadruples all of it, because attention is quadratic.

In one sentence: Per token, forward work is about twice the non-embedding parameters plus an

attention term that grows with context; training is about three times that.

10.8 Line by line: toy, GPT-2, Qwen3

component	toy	GPT-2 small (2019)	Qwen3-0.6B (2025)	why it changed
tokenizer	whole words, 12	byte-level BPE, 50,257	byte-level BPE, 151,936 rows	Ch. 2: no unknown words; more languages need more tokens
embeddings	12×4 table	$50,257 \times 768$	$151,936 \times 1,024$	Ch. 2
positions	none (order lost, §3.11)	learned table, 1,024 slots	RoPE rotation, no parameters	Ch. 4: relative positions, longer contexts
norm	none	LayerNorm before each sublayer	RMSNorm before each sublayer, plus QK-norm	Ch. 6: stable depth; §10.3
attention	1 head of 4	12 heads of 64	16 query heads of 128, 8 key-value heads	Ch. 5, Ch. 9: cache bill
MLP	—	GELU, $768 \rightarrow 3,072 \rightarrow 768$	SwiGLU, gate and up $1,024 \rightarrow 3,072$, down	Ch. 5
biases	none	on every linear map	none	dropped in Qwen3 (Yang et al., 2025)
residual stream	—	yes	yes	Ch. 5
final norm	—	LayerNorm	RMSNorm	Ch. 6
unembedding	—	tied to embeddings	tied to embeddings	Ch. 7

WHERE THIS PICTURE BREAKS. This table compares two published designs. It does not tell you that each change made the model better on its own; the reports describe what was built, and only some changes were tested one at a time. Chapter 11 shows how to test a change yourself.

Abstraction — read the configuration, not the name. A model’s name gives one rounded number. Its configuration gives every shape, and from the shapes you can derive the parameter count, the arithmetic per token, the memory per token of conversation (Chapter 9) and where each cost comes from. The same habit pays off in any system you build or buy: an agent’s tool list, a memory store’s schema, a pipeline’s config file. Derive the costs from the declared shapes, then check your derivation against the running system. When the two disagree, the disagreement shows you where your understanding was wrong.

Looking back.

1. (Chapter 3) Why did we divide scores by $\sqrt{d_k}$? With Qwen3’s head width of 128, what is the divisor?
2. (Chapter 9) Using Qwen3’s configuration, recompute its cache per token.
3. (Chapter 1) Which of the four requirements of Chapter 1 does each of these meet: embeddings, RoPE, attention, the causal mask?

Exercises

Exercise 10.1. Recount GPT-2 small’s attention parameters per block from the shapes: W_{QKV} is $768 \times 2,304$ with 2,304 biases, and W_O is 768×768 with 768 biases. Check your answer against the table.

Exercise 10.2. (Fading.) Fill in the Qwen3 MLP count: gate $1,024 \times __$, up $1,024 \times __$, down $__ \times 1,024$; total $__$.

Exercise 10.3. Suppose Qwen3-0.6B did *not* tie its unembedding to its embedding table. How many parameters would it have, and what share of them would be the two tables together?

Exercise 10.4. Design question. You must halve Qwen3-0.6B’s cache per token without changing its width or number of layers. Name two configuration changes that do it, and for each, say what you expect to lose.

Answers to Chapter 10

Check yourself 10.1. (b). The 16 query heads produce 2,048 numbers, and W_O maps them back to the stream width 1,024. (a) is misconception M10.6. (c) has the direction reversed. (d) forgets that heads’ outputs must be brought back to width d before being added to the residual stream (Chapter 5).

Check yourself 10.2. (b). The $2N$ part depends only on the parameters; the attention part is proportional to context. (a) treats all work as attention. (c) ignores the scores. (d) confuses the work for one new token with the work for the whole sequence, which does grow quadratically.

Looking back. (1) To keep random scores about one unit wide at any head width, so softmax does not start saturated (§3.8); $\sqrt{128} \approx 11.3$. (2) $2 \times 28 \times 8 \times 128 \times 2 = 114,688$ bytes. (3) Embeddings meet Requirement 1 (similarity), RoPE Requirement 2 (order), attention Requirement 3 (direct reach) and, with the mask, Requirement 4 (all positions at once, honestly).

Exercise 10.1. $768 \times 2,304 + 2,304 + 768 \times 768 + 768 = 2,362,368$, as in the table.

Exercise 10.2. 3,072, 3,072, 3,072; $3 \times 1,024 \times 3,072 = 9,437,184$.

Exercise 10.3. Add another $151,936 \times 1,024 = 155,582,464$: 751,632,384 parameters, of which 41% would be the two tables.

Exercise 10.4. Halve the key-value heads (8 to 4): fewer independent ways to describe each token. Or halve the head width (128 to 64) for keys and values: each key and value carries fewer numbers. A third option, storing the cache in 8 bits instead of 16, changes no configuration number but loses precision.

What you now know

- How to read a configuration and derive every shape in the model from it.
- How to count parameters by component; the counts match the real models exactly (124,439,808 and 596,049,920).
- That most parameters live in the MLPs and the embedding table, not in attention.
- What changed from the toy to GPT-2 to Qwen3, and which chapter motivates each change.
- That forward work is about $2N$ per token plus an attention term that grows with context, and training about $6N$.

11 Designing your own transformer

Map of this chapter. Ten chapters took one model apart. This one puts a model together from nothing, and the hard part is no longer the mechanism but the choices. **The problem:** you have a budget (of compute, of memory, of time) and a job. **The steps:** name every decision (§11.1); price the three budgets, compute (§11.2), parameters (§11.3) and running memory (§11.5); see what heads and key–value sharing really buy (§11.4); look at what our own experiments showed and what they could not show (§11.6); then make one complete design and defend every line (§11.7). **Where it ends:** a table of trade-offs you can reuse (§11.8), and a look back over the whole notes.

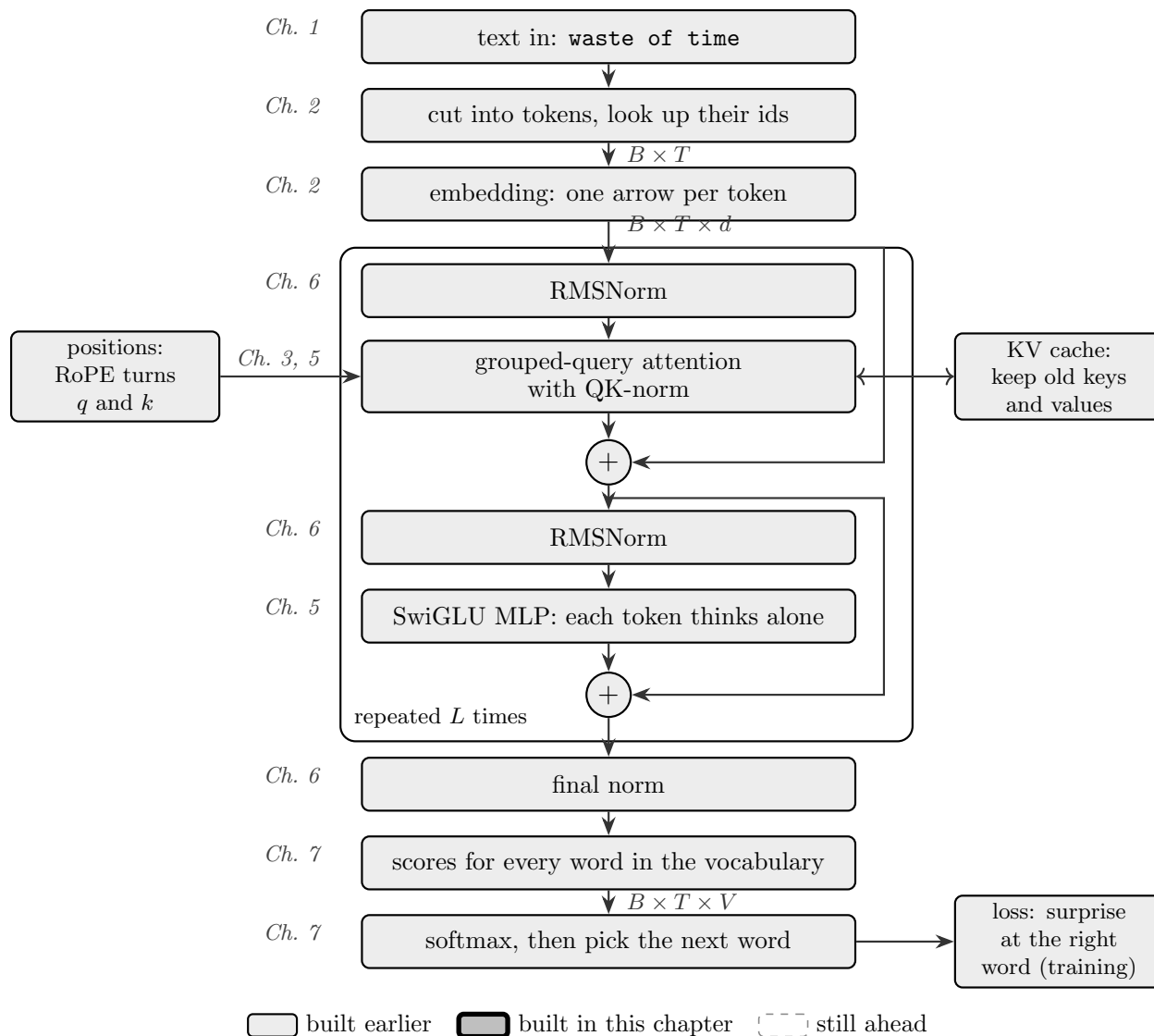


Figure 65: Where we are in the machine. The whole machine, built. Designing is setting its dials, and every dial sits on one block: the vocabulary V on the token and embedding blocks (and the scores at the bottom); the width d on every arrow of the stream; the depth L on the frame, which is repeated L times; the query heads h and the key–value heads inside attention; the MLP width d_{ff} inside the MLP; and the context length on the KV cache at the side, which grows with every token.

Before you start.

1. How many zeros are in one billion? *Refresher: thousand = 10^3 , million = 10^6 , billion = 10^9 , trillion = 10^{12} .*
2. What is $10^3 \times 10^6$? *Refresher: multiplying powers of ten adds the exponents, so it is 10^9 .*
3. What is $\sqrt{10^{18}}$? *Refresher: a square root halves the exponent: 10^9 .*
4. A file is 2,000,000,000 bytes. How many gigabytes (GB) is that? *Refresher: here 1 GB = 10^9 bytes, so 2 GB.*
5. If a cost grows “as the square of the width”, what happens to it when the width doubles? *Refresher: $2^2 = 4$, so it becomes four times bigger.*

11.1 Design is a list of decisions

Near the end of the ninth session I asked the room this:

“how many attention heads we should have? what should be my feed forward dimension? would it be 192? would it be 256? how many blocks should i have?”

and a little later, “what should be your dimension of token? how do you determine the token dimension?” That is the whole of this chapter. Every one of those questions is a *decision*, and a decision is only as good as the *assumptions* under it and the *trade-off* it accepts.

Predict first. Before reading on, write down every number you would have to choose to build a model like the one in Chapter 10. How many did you get? Most people find five or six. There are about ten.

PICTURE IT. You are building a house with a fixed amount of money. You choose how wide the rooms are, how many floors to stack, how many doors each room has, how big the storage cupboard is, and how many guests it must hold at once. Every extra metre in one place is a metre you cannot spend somewhere else.

SAY IT EXACTLY. A decoder-only transformer is fixed by these choices:

Decision	Symbol	Where you met it
vocabulary size	V	Chapter 2
width (numbers per token)	d	Chapters 2, 3
depth (number of blocks)	L	Chapters 5, 6
query heads and head size	H, d_h with $Hd_h = d$	Chapters 3, 5
key-value heads	H_{kv}	Chapter 9
MLP width and type	d_{ff} , GELU or SwiGLU	Chapter 5
normalisation and where it sits	LayerNorm or RMSNorm, before or after	Chapter 6
positional scheme	learned, sinusoid, RoPE	Chapter 4
context length	n_{ctx}	Chapters 4, 9
training tokens	D	Chapter 8

WHERE THIS PICTURE BREAKS. A house’s rooms do not talk to each other. In a transformer the choices interact: the head size is the width divided by the number of heads, and the memory needed at run time depends on depth, key-value heads and context together. The picture gives you the budget; it does not give you the interactions.

WHY IT HAD TO BE THIS WAY. There is no formula that outputs “the best transformer”. There are budgets, and every budget turns some of these choices into consequences you can calculate. So the method of this chapter is always the same: state the assumption, compute the consequence, then decide.

In one sentence: Designing a transformer means fixing about ten numbers, each one a decision whose cost you can calculate before you train anything.

11.2 Budget 1: compute

Predict first. You double the number of parameters and also double the amount of training text. How much more compute do you need: twice as much, three times, or four times?

PICTURE IT. Every token of training text has to pass through every weight. Think of a factory where each item visits each machine: the work is (number of items) $\times$ (number of machines). Double both and the work is four times larger.

SAY IT EXACTLY. A weight touched by a token costs one multiply and one add, so two operations (FLOPs) on the way forward. Learning needs a backward pass that costs about twice the forward one (Chapter 8). So training costs about $2 + 4 = 6$ FLOPs per parameter per token (Kaplan et al., 2020):

$$C \approx 6ND \quad C: \text{training FLOPs}, \quad N: \text{parameters}, \quad D: \text{training tokens}$$

So the answer to the prediction is four times. The formula ignores the attention scores themselves, which add a little for long contexts (Chapter 10 counts them).

How to split a compute budget between N and D . If C is fixed, a bigger model sees less text and a smaller model sees more. Hoffmann et al. (2022) trained hundreds of models and found that, to reach the lowest loss for a fixed C , parameters and tokens should grow together, with roughly *20 training tokens per parameter*. Put $D = 20N$ into $C = 6ND$:

$$C = 120N^2 \implies N_{\text{opt}} \approx \sqrt{C/120}, \quad D_{\text{opt}} \approx 20N_{\text{opt}}.$$

Worked example A: one CPU thread for one hour. Our ablation models in §11.6 have 108,928 parameters and each saw 3,072,000 tokens in about 27 seconds on one thread of our build machine. That is about 7.41×10^{10} training FLOPs per second, so one hour gives $C \approx 2.67 \times 10^{14}$. The rule of thumb then says: about 1,490,843 parameters trained on 29.8 million tokens. Our ablation models used 28 tokens per parameter, close to the rule.

Worked example B: 1,000 hours of a data-centre GPU. Let’s assume a GPU whose data sheet promises 989 trillion FLOPs per second (NVIDIA’s figure for the H100 at 16-bit precision), and let’s assume your code keeps it 40% busy, a common real-world figure. Then $C \approx 1.42 \times 10^{21}$, and the rule of thumb gives about 3.44 billion parameters trained on 68.9 billion tokens. We design that model in §11.7.

A tempting wrong idea (M11.3). “Compute depends on the number of parameters.” It depends on parameters *and* tokens, in equal measure. A 1-billion-parameter model trained on a thousand times more text costs a thousand times more.

A tempting wrong idea (M11.2). *“Bigger is always better.”* At a fixed compute budget, a model that is too big for its data ends up worse than a smaller one trained on more text. That is the finding of Hoffmann et al. (2022): many earlier large models were under-trained for their size.

Why modern small models ignore the 20-to-1 rule. Qwen3-0.6B has 596,049,920 parameters. The rule of thumb would train it on about 11.9 billion tokens. The Qwen3 technical report (2025) says the family was pre-trained on about 36 trillion tokens: roughly 60,398 tokens per parameter, a few thousand times the rule. Is that a mistake? Ask what the rule minimises: *training* compute for a given loss. But a small model is trained once and then run billions of times. Every extra token of training makes the model better without making it bigger, and a smaller model is cheaper on every single request forever after. When serving costs dominate, you pay more at training time to save at run time. (We take the 36-trillion figure from the report and could not check it ourselves.)

WHERE THIS PICTURE BREAKS. The factory picture says the cost is items times machines. It does not say whether more items or more machines gives a better product; that question needed experiments (Hoffmann et al.), and the answer changes once you count what the product costs to use.

WHY IT HAD TO BE THIS WAY. Compute is the one budget you cannot borrow against. Writing $C \approx 6ND$ first turns a vague wish (“a good model”) into two numbers you can argue about.

Check yourself 11.1. You have a fixed compute budget and you decide to make the model 4 times bigger. By the rule of thumb in this section, what should happen to the number of training tokens?

- (a) Stay the same; compute only depends on parameters.
- (b) Shrink to a quarter, since $C = 6ND$ is fixed.
- (c) The plan breaks the 20-to-1 balance; with C fixed you should not grow N alone.
- (d) Grow 4 times, because bigger models always need more data, whatever the budget.

In one sentence: Training compute is about $6ND$, and for a fixed budget the loss-optimal split is roughly 20 tokens per parameter, unless running the model will cost more than training it.

11.3 Budget 2: parameters, and where they live

Predict first. In Qwen3-0.6B, which part holds more parameters in each block: the attention or the MLP? And what fraction of the whole model do you think is the vocabulary table?

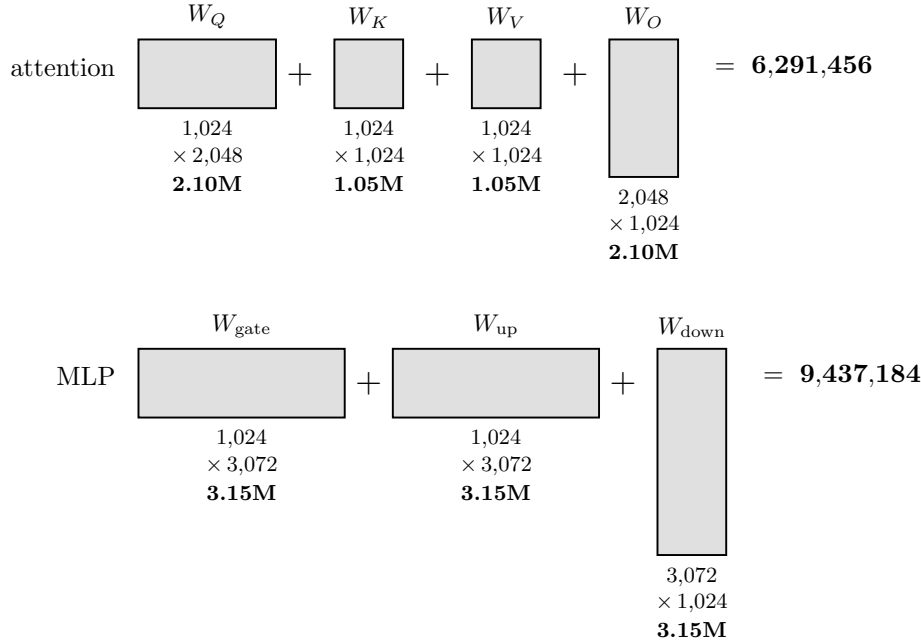
PICTURE IT. A recipe card has one number for each ingredient. In Chapter 3 a $d \times d_k$ matrix was d_k recipe cards with d ingredients each. So a matrix costs (inputs) $\times$ (outputs) numbers, and every design choice is a choice of how many cards, with how many ingredients.

SAY IT EXACTLY. Count one block of a model with width d , H query heads and H_{kv} key-value heads of size d_h , and an MLP of width d_{ff} :

$$\underbrace{d H d_h}_{W_Q} + \underbrace{2 d H_{kv} d_h}_{W_K, W_V} + \underbrace{H d_h d}_{W_O} + \underbrace{3 d d_{ff}}_{\text{SwiGLU MLP}} + (\text{small norm vectors}).$$

The model adds the vocabulary table Vd (and a second one if the output layer is not tied to it, Chapter 7). For Qwen3-0.6B this gives 6,291,712 attention parameters and 9,437,184 MLP parameters per block: the MLP is 60% of each block. The vocabulary table alone is 26% of the whole model; in GPT-2 the token and position tables are 32%.

Qwen3-0.6B: one block, every weight matrix, drawn to scale



one block = attention 6,291,456 + QK-norm 256 + MLP 9,437,184 + two norms $2 \times 1,024 = \mathbf{15,730,944}$

whole model = embedding table $151,936 \times 1,024$ (155,582,464) + $28 \times$ one block + final norm 1,024 = **596,049,920**

Figure 66: The formula above, drawn. Each rectangle is one weight matrix of a Qwen3-0.6B block, with one row per input and one column per output, so its area is its parameter count. Add the rectangles and you get the block; multiply by 28 and add the embedding table and you get the model. Compare the area of the MLP row with the attention row before reading the numbers.

Sanity check. Our calculator is checked two ways. Building the real Hugging Face model classes on an empty (“meta”) device, with no weights downloaded, gives exactly 124,439,808 parameters for GPT-2 and 596,049,920 for Qwen3-0.6B, the same as our formula. A second calculator written for Chapter 10 gives the same totals: the check says it agrees.

Width versus depth. Doubling the width d multiplies every matrix in every block by about four (both sides of each matrix grow). Doubling the depth L only doubles the block count. So width is the expensive direction for parameters, depth the expensive direction for *time*: blocks run one after another, so a deep model is slower per token even at equal size. Kaplan et al. (2020) found that, at a fixed parameter count, loss depends only weakly on the exact shape across a wide range. In practice designers keep d/L somewhere between about 50 and 150 and spend their attention elsewhere.

A tempting wrong idea (M11.7). “*Width is free: a wider vector holds more information.*” In the ninth session the room’s answer was the right one: “if the hidden dimensions are too much, then your more parameters will come into the place or more computations”. Width costs quadratically.

Vocabulary size. A bigger vocabulary means a bigger table (Vd numbers) but shorter sequences, since each token covers more text (Chapter 2). Shorter sequences make attention and the KV cache cheaper. For a small model the table is a large share of the budget (26% here), which is one reason small models tie their input and output tables.

A tempting wrong idea (M11.5). “*The vocabulary is free; it is only a lookup.*” The lookup costs no FLOPs, but it costs parameters, memory and, at the output end, a full $d \times V$ multiplication for every generated token.

Check yourself 11.2. You double the width d of a model and keep everything else in proportion ($d_{ff} = 3d$, d_h fixed). Roughly what happens to the parameters inside the blocks?

- (a) They double, like the width.
- (b) They grow about four times.
- (c) They stay the same; width only changes how tokens are described.
- (d) They halve, because each head gets smaller.

In one sentence: A matrix costs inputs times outputs, so width costs quadratically, depth linearly, and the vocabulary table can be a quarter of a small model.

11.4 Heads, head size, and sharing keys and values

In the ninth session I asked what more heads means, and the room answered “more ... ways to look at things”, then found the catch: “what is a trade-off general wider compute expensive”. Let’s make that exact.

Predict first. Our ablation models have width 64. How many parameters does the attention gain if you go from 1 head of size 64 to 4 heads of size 16?

PICTURE IT. In Chapter 3 one head had 64 recipe cards. Four heads do not get 64 cards each; they split the same 64 cards into four packs of 16. Each pack asks its own, smaller question. You buy variety of questions with detail per question.

SAY IT EXACTLY. With $Hd_h = d$ fixed, W_Q and W_O stay $d \times d$ whatever H is, so the parameter count does not move: our 1-, 2- and 4-head models all have 108,928 parameters. What changes is the shape of the computation: H separate score tables of size $T \times T$ instead of one.

Sharing is a separate decision. Each query head needs keys and values to compare against, but several query heads can *share* one set. With $H_{kv} < H$ key-value heads this is *grouped-query attention* (GQA, Ainslie et al., 2023); with $H_{kv} = 1$ it is *multi-query attention* (MQA, Shazeer, 2019). It shrinks W_K , W_V and, more importantly, the KV cache of Chapter 9 by a factor H/H_{kv} . Our 4-head model with 2 shared key-value heads has 100,736 parameters instead of 108,928.

A tempting wrong idea (M11.1). *“More heads means more capacity.”* At fixed width, more heads means the same parameters cut into more, smaller pieces. It can help (more kinds of question) or hurt (each question has fewer numbers); §11.6 shows that in our runs the difference was smaller than the noise.

WHERE THIS PICTURE BREAKS. Packs of recipe cards suggests each head learns a tidy, nameable job. Some heads do; many share work in ways no one has a name for. The picture is about cost, not about meaning.

WHY IT HAD TO BE THIS WAY. Heads trade the number of questions against the detail of each; sharing keys and values trades a little flexibility for a lot of memory at run time. The second trade is almost always worth it, which is why Qwen3-0.6B uses 16 query heads but only 8 key–value heads.

In one sentence: At fixed width, more heads change the shape of attention, not its size; sharing keys and values is the decision that saves memory.

11.5 Budget 3: memory when the model runs

Predict first. A phone has 4 GB free. Qwen3-0.6B has about 0.6 billion parameters at 2 bytes each. Does it fit, with a conversation of 8,000 tokens?

PICTURE IT. The weights are the furniture: always there, same size. The KV cache is the guests: it grows with every token of the conversation and must be kept until the conversation ends.

SAY IT EXACTLY. Memory to run $\approx$ weights + KV cache. From Chapter 9, the cache stores a key and a value for every layer, key–value head and token:

$$\text{KV bytes} = 2 \cdot L \cdot H_{kv} \cdot d_h \cdot n_{\text{tokens}} \cdot \text{bytes per number}.$$

For Qwen3-0.6B at 2 bytes per number that is 114,688 bytes per token. So:

	8,192 tokens	32,768 tokens
weights, 16-bit	1.19 GB	1.19 GB
weights, 4-bit	0.30 GB	0.30 GB
KV cache, 16 heads (no sharing)	1.88 GB	7.52 GB
KV cache, 8 heads (Qwen3’s GQA)	0.94 GB	3.76 GB
KV cache, 1 head (MQA)	0.12 GB	0.47 GB

So the answer to the prediction: 16-bit weights (1.19 GB) plus an 8,192-token cache (0.94 GB) fit in 4 GB with room for the phone’s own needs. At 32,768 tokens the cache alone is 3.76 GB and the plan fails, unless you shorten the context, store the cache in fewer bytes, or share more heads. *Context length is a memory decision.*

A tempting wrong idea (M11.4). *“The parameter count tells you the memory.”* It tells you the furniture. The guests (the KV cache) can outgrow it, and they grow with every token.

Check yourself 11.3. You keep Qwen3-0.6B but want 4 times the context. Which change keeps the KV cache the same size?

- (a) Make the weights 4-bit instead of 16-bit.

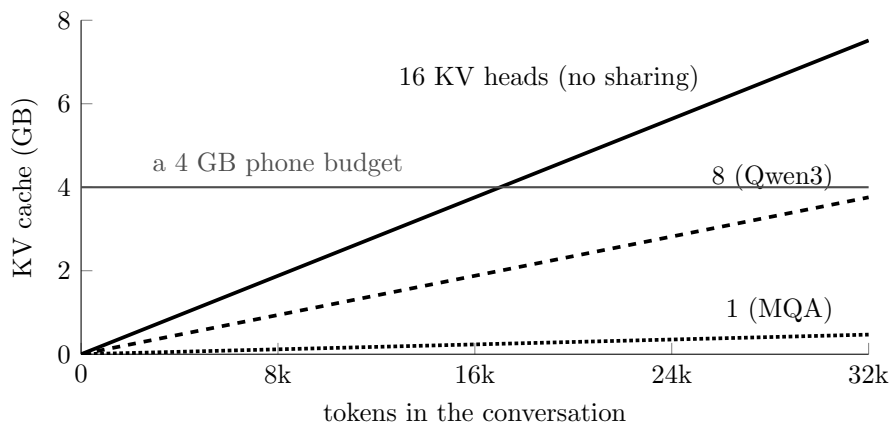


Figure 67: The KV cache of Qwen3-0.6B grows in a straight line with the conversation. The number of key-value heads sets the slope.

- (b) Cut the key-value heads from 8 to 2.
- (c) Nothing; the cache is part of the parameters and does not depend on context.
- (d) Double the number of query heads.

In one sentence: Running memory is weights plus a KV cache that grows with every token, so context length and the number of key-value heads are memory decisions.

11.6 What our own experiments showed, and what they could not

Claims about design are cheap. So we trained tiny transformers ourselves and changed one thing at a time. Each model reads characters (a vocabulary of 41), has width 64, 2 blocks and a context of 64, and is trained for 1,500 steps on sessions 1–8 of my lectures (961,478 characters). We measure the loss on session 9 (324,250 characters), which the models never saw. Lower is better; a model that guesses uniformly would score 3.71.

Predict first. Before you look: which change will hurt most, removing positional information, removing the residual connections, or removing normalisation? Which will make no measurable difference?

Model	parameters	val. loss	change
baseline (2 heads, learned positions, residuals, LayerNorm), seed 1	108,928	1.444	—
baseline, seed 2	108,928	1.394	(noise: 0.050)
no positional information	104,832	1.803	+0.358
no residual connections	108,928	2.283	+0.839
no normalisation	108,288	1.455	+0.011
1 head of size 64	108,928	1.464	+0.020
4 heads of size 16	108,928	1.416	-0.029
4 query heads, 2 shared key-value heads	100,736	1.409	-0.035

Reading the results honestly. The two baselines differ by 0.050 with nothing changed but the random seed. That is our noise level, and any difference of that size tells us nothing.

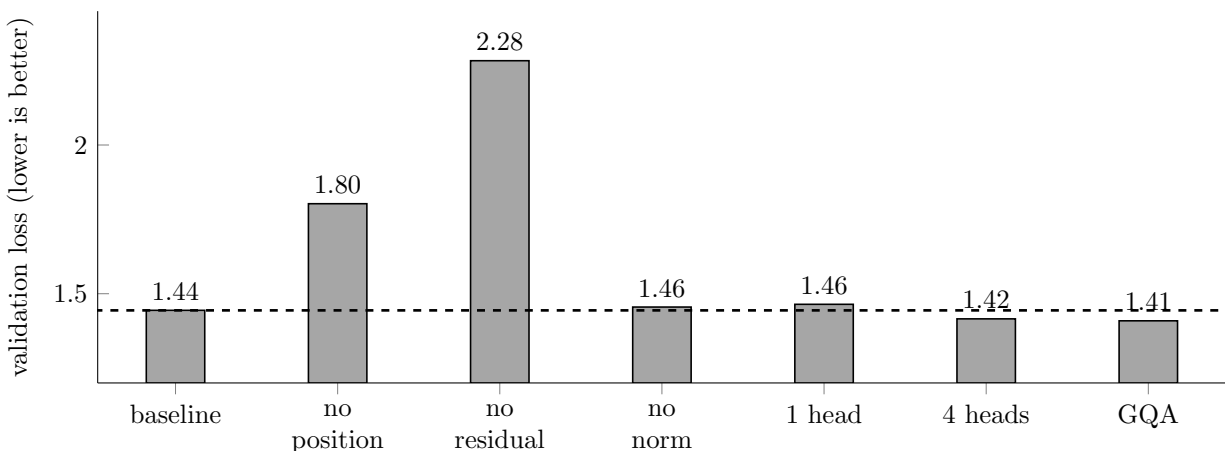


Figure 68: One change at a time. The dashed line is the baseline. Two baselines that differ only in their random seed already differ by 0.050, so only bars far from the line mean something.

- *Removing residual connections* (+0.839) and *removing positional information* (+0.358) are far outside the noise. These are real.
- The no-position model still beats uniform guessing by a wide margin. It can learn something about position because the causal mask lets each token count how many tokens it can see; Haviv et al. (2022) found the same in larger models.
- *Removing normalisation* (+0.011), *changing the number of heads* (+0.020, -0.029) and *sharing key-value heads* (-0.035) are all inside the noise. With 2 blocks we cannot see what normalisation does for depth (Chapter 6). And GQA lost nothing measurable while using fewer parameters, which matches what Ainslie et al. (2023) report at scale.

A tempting wrong idea (M11.6). *“It went down, so it is better.”* The 4-head and GQA models scored lower than the first baseline, and it is tempting to call them better. The second baseline scored lower still. Without a second seed you would have fooled yourself.

WHERE THIS PICTURE BREAKS. These are 0.1-million-parameter models that see each character a few times. Effects that only appear at scale or depth (normalisation, many heads) cannot show up here. A small experiment can show you that something matters; it is much weaker at showing that something does not.

WHY IT HAD TO BE THIS WAY. This is the same discipline as any evaluation: know your noise before you read your result. It is the reason a design is defended with several seeds, or with a larger run, before anyone believes it.

In one sentence: In our runs, residuals and positions mattered far beyond the noise; heads, normalisation and key-value sharing did not, at this tiny scale.

11.7 A complete design, line by line

The brief: *1,000 GPU-hours, a model that will also be run on ordinary hardware with an 8,192-token context.* From §11.2, the compute-optimal size is about 3.44 billion parameters. Here is a design, with the assumption and the reason behind every line. Our script searched widths and

depths for a shape near that size with d/L between 60 and 140.

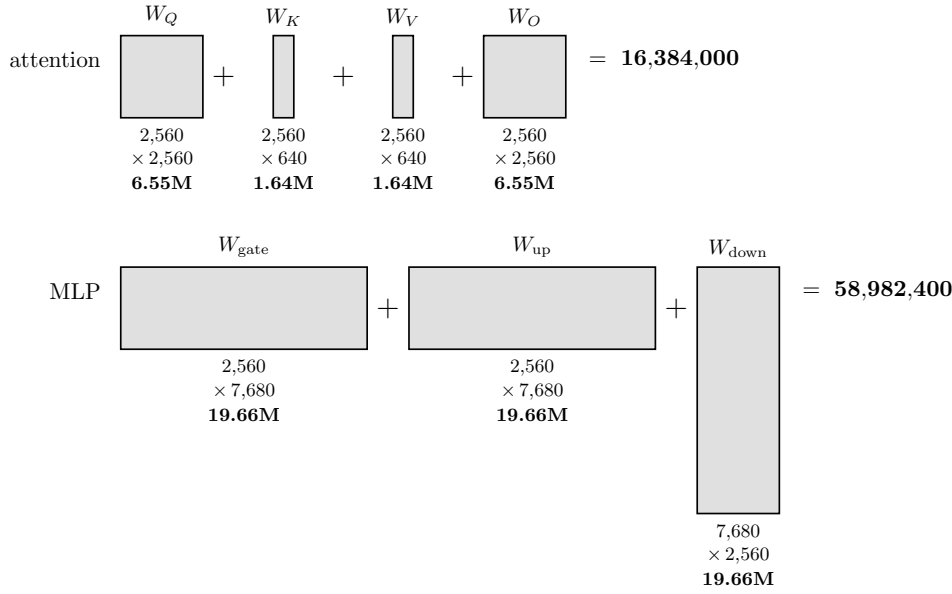
Decision	Choice	Why
width d	2560	near the target size with a sensible width-to-depth ratio (71)
depth L	36	enough blocks for many steps of reasoning; pre-norm makes this depth trainable (Ch. 6)
head size d_h	128	the size Qwen3 uses; large enough for detailed questions
query heads H	20	d/d_h
key-value heads H_{kv}	5	a quarter of H : KV cache at 8,192 tokens is 0.75 GB instead of 3.02 GB
MLP	SwiGLU, $d_{ff}=7,680$	three times the width, as in Qwen3-0.6B
vocabulary	151,936	reuse an existing multilingual tokenizer (Ch. 2); untied tables, 22% of parameters
normalisation	RMSNorm, before each sublayer	stable at depth (Xiong et al., 2020; Ch. 6)
positions	RoPE	relative positions, extends to longer contexts (Ch. 4)
context tokens D	8,192 68.9 billion	the brief; the cache cost is in the KV line 20 per parameter, unless serving costs justify more (§11.2)
total	3.49 billion	target was 3.44 billion

Drawn the same way as Qwen3’s block in §11.3, here is the block of this design. Notice how small W_K and W_V are: with 5 key-value heads instead of 20, they are a quarter of W_Q , and so is the cache.

Your turn, with fewer steps given. Design for the phone of §11.5: 4 GB in total, a 4,096-token context. Assume 4-bit weights and a 16-bit cache. Write down: the largest parameter count that leaves 1 GB for the phone, a width and depth near it, and how many key-value heads you need. Use the formulas of §11.3 and §11.5; Exercise 11.3 walks through one answer.

Abstraction — a design is a set of decisions with written-down assumptions. Every line in the table has the same form: an assumption (“it will run with 8,192 tokens”), a consequence you can compute (“0.75 GB of cache”), and a decision. The same form works for systems built around models. In an agent’s memory system, the context window is a budget measured in tokens, and every token placed in it costs KV memory and attention time; deciding what to retrieve is deciding what is worth that cost. The model does not make these decisions for you, and the arithmetic of this chapter is how you make them in the open.

The design of §11.7: one block, drawn to scale



one block = attention 16,384,000 + MLP 58,982,400 + two norms $2 \times 2,560 = 75,371,520$

whole model = embedding table $151,936 \times 2,560$ (388,956,160) + $36 \times$ one block + final norm 2,560 + output table = **3,491,289,600**

Figure 69: One block of the §11.7 design as a sum of its weight matrices, drawn to scale. The thin W_K and W_V are the price of grouped-query attention paid in reverse: fewer key-value heads, fewer numbers to store.

11.8 The trade-off table

Choice	What you gain	What you pay
wider (d up)	more numbers per token	parameters and FLOPs grow with d^2
deeper (L up)	more steps of processing	time per token; harder to train without pre-norm
more heads, same d	more kinds of question	less detail per question; no parameter change
fewer KV heads (GQA, MQA)	smaller KV cache, faster generation	a little flexibility; little measured loss
bigger vocabulary	shorter sequences	a bigger table and output layer
longer context	more text in view	KV memory and attention time grow with it
pre-norm vs post-norm	stable deep training	a slightly different signal path (Ch. 6)
RoPE vs learned positions	relative positions, longer contexts	nothing measurable in most reports
more training tokens	a better model at the same size	training compute ($6ND$)

In one sentence: Every choice in a transformer buys something specific and pays for it in compute, parameters, memory or time, and you can write down which before you train.

Looking back.

1. (Chapter 3) Why do we divide attention scores by $\sqrt{d_k}$, and what would happen to a model with $d_h = 128$ if we did not?
2. (Chapter 9) Why does the KV cache store keys and values but not queries?
3. (Chapter 1) Which failure of the RNN does the “time per token grows with depth” cost in §11.3 remind you of, and why is it less severe?

Exercises

Exercise 11.1. A team has $C = 6 \times 10^{20}$ FLOPs. Using the 20-tokens-per-parameter rule, find N and D .

Exercise 11.2. Count the parameters inside one block of a model with $d = 512$, $H = 8$, $H_{kv} = 2$, $d_h = 64$ and a SwiGLU MLP with $d_{ff} = 1536$, ignoring the norm vectors. What share is attention?

Exercise 11.3. The phone of §11.7: 4 GB in total, 1 GB kept for the phone, 4-bit weights, a 16-bit cache, 4,096 tokens. If you copy Qwen3-0.6B’s attention shape ($L = 28$, $d_h = 128$, $H_{kv} = 8$), how much is left for the weights, and how many parameters is that?

Exercise 11.4. Your colleague trains one model with 8 heads and one with 2 heads and reports a validation loss of 1.41 versus 1.44. Write the two questions you would ask before believing that 8 heads are better.

Answers to Chapter 11

Check yourself 11.1. (c). With $C = 6ND$ fixed, making N four times bigger forces D to a quarter, which moves you from 20 tokens per parameter to 1.25: far from the balance, so the plan is wrong, not the arithmetic. (a) is M11.3: compute depends on tokens too. (b) is the correct arithmetic but accepts the imbalance as if it were fine. (d) is M11.2: more data would need more compute than you have.

Check yourself 11.2. (b). Every matrix has d on at least one side, and most have it on both (or d and $d_{ff} = 3d$), so the count grows about four times. (a) treats a matrix like a list. (c) is M11.7, width as free. (d) mixes up the number of heads with their size; at fixed d_h doubling d doubles the heads.

Check yourself 11.3. (b). The cache is proportional to H_{kv} times the tokens, so 4 times the tokens and a quarter of the key–value heads cancel. (a) shrinks the weights, not the cache (M11.4). (c) is M11.4 itself. (d) changes the queries, which are never cached (Chapter 9).

Exercise 11.1. $N = \sqrt{C/120} = \sqrt{5 \times 10^{18}} \approx 2.24 \times 10^9$ parameters, and $D = 20N \approx 4.47 \times 10^{10}$ tokens. Check by putting both back into $6ND$: you get 6×10^{20} again.

Exercise 11.2. W_Q : $512 \times 512 = 262,144$; W_K and W_V together: $2 \times 512 \times (2 \times 64) = 131,072$; W_O : 262,144; attention total 655,360. MLP: $3 \times 512 \times 1536 = 2,359,296$. Block total 3,014,656, of which attention is about 22%. The strategy that worked: count each matrix as inputs times outputs, and write the shapes before multiplying.

Exercise 11.3. Cache: $2 \times 28 \times 8 \times 128 \times 4096 \times 2$ bytes ≈ 0.47 GB. Left for weights: $4 - 1 - 0.47 \approx 2.53$ GB. At half a byte per parameter, that is about 5.06 billion parameters, so memory is not the limit here; speed on a phone processor will be. A sensible answer stays much smaller and spends the margin on a longer context.

Exercise 11.4. (1) How large is the difference between two runs of the *same* model with different seeds? In our runs it was 0.050, larger than this gap. (2) Did both models get the same data, steps and parameters? At fixed width they did have the same parameters (§11.4), so if the gap is real it comes from the shape of attention, not size.

What you now know

From this chapter.

- Training compute is about $6ND$; for a fixed budget the loss-optimal split is about 20 tokens per parameter, and small models meant for heavy use are trained far beyond it.
- A matrix costs inputs times outputs: width is quadratic, depth linear, and the vocabulary table can be a quarter of a small model. Our calculator matches the real model classes exactly.
- Heads at fixed width change the shape of attention, not its size; sharing keys and values saves the memory that grows with every token.
- Small experiments need a noise estimate before their results mean anything.

From the whole notes. You started with one message, **waste of time**, and a job: reply well.

- Writing is repeated next-word guessing, and the problem labels itself (Chapter 1).
- Tokens become arrows, and similar tokens get nearby arrows (Chapter 2).
- Attention lets each token ask, be found, and hand over, all at once for a whole batch (Chapter 3).
- Rotating queries and keys gives order back (Chapter 4).
- Heads, the MLP and the residual stream let tokens talk and then think (Chapter 5); normalisation lets you stack many blocks (Chapter 6).
- Scores become words through the output table, softmax and sampling (Chapter 7); training turns and stretches the arrows (Chapter 8).
- The KV cache makes generation fast and costs memory (Chapter 9); GPT-2 and Qwen3 are the same machine at scale (Chapter 10).
- And every number in a real model is a decision you can now price and defend (Chapter 11).

Appendix A The calculus, for readers who want it

Map of this chapter. Chapter 8 found slopes by probing: step a little left, a little right, compare. This appendix shows how the slopes are computed exactly, for every knob at once. We start from the slope of a curve (§A.1), add the one rule that does all the work, the chain rule (§A.2), and then apply it to the three places that matter in a transformer: the loss at the end (§A.3), attention (§A.4) and the residual connection (§A.5). Every formula here was checked by a script, `code/appA.check.py`, against the probing method of Chapter 8. The largest disagreement anywhere is 1.1×10^{-9} (a relative error), which is rounding noise.

Before you start.

- (1) The slope of a straight line is “rise over run”. A line goes through (1, 2) and (3, 8). Slope? *Refresher:* $(8 - 2)/(3 - 1) = 3$.
- (2) What is $(a + b)^2$? *Refresher:* $a^2 + 2ab + b^2$.
- (3) A matrix of shape 3×4 times one of shape 4×2 gives what shape? *Refresher:* 3×2 ; the inner sizes must match and disappear (Chapter 3, §3.4).

A.1 From the slope of a line to the slope of a curve

Predict first. The curve $y = x^2$ is not a straight line, so its steepness changes from place to place. How steep is it at $x = 3$? Guess a number before reading on.

PICTURE IT. Zoom in on a curve far enough and it looks like a straight line. The slope of that tiny straight piece is the slope of the curve at that point.

Let us zoom in on $y = x^2$ at $x = 3$, using the rise-over-run of a short step h :

$$\frac{(3+h)^2 - 3^2}{h} \quad \text{gives} \quad 7.000 \ (h = 1), \quad 6.100 \ (h = 0.1), \quad 6.001 \ (h = 0.001).$$

The numbers settle towards 6.

SAY IT EXACTLY. Expanding, $((3+h)^2 - 9)/h = (6h + h^2)/h = 6 + h$, and as the step h shrinks to nothing this becomes exactly 6. That limit is the *derivative*, written $\frac{dy}{dx}$. Doing the same at a general x gives $\frac{d}{dx}x^2 = 2x$.

WHY IT HAD TO BE THIS WAY. Probing (Chapter 8) is this formula with a small but not-zero h . The derivative is what probing would give with a step too small to measure. It is exact, and it costs nothing extra once you know the formula.

Check yourself A.1. Probing $y = x^2$ at $x = 3$ with $h = 0.1$ gave 6.1, not 6. Why the extra 0.1?

- (a) Rounding error in the computer.
- (b) The curve bends within the step, and the bend adds h to the rise-over-run.
- (c) The derivative of x^2 is $2x + 0.1$.

A.2 The chain rule: slopes multiply

PICTURE IT. Three gears in a row. If the first turns the second 3 times as fast, and the second turns the third 2 times as fast, the first turns the third $3 \times 2 = 6$ times as fast. When one quantity

feeds another, the rates multiply.

SAY IT EXACTLY. If $y = f(u)$ and $u = g(x)$, then

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}.$$

Example: $y = (3x + 1)^2$. Let $u = 3x + 1$, so $y = u^2$. Then $dy/du = 2u$ and $du/dx = 3$, so $dy/dx = 2(3x + 1) \cdot 3$, which at $x = 1$ is $2 \cdot 4 \cdot 3 = 24$. Probing with a tiny step gives 24.0000.

With many knobs, the slope of the loss with respect to each knob is called a *partial derivative*, written $\partial L / \partial w$; the list of all of them is the gradient. *Backpropagation* is the chain rule applied from the loss backwards, layer by layer: each layer receives “how much the loss changes when my output changes” and passes back “how much the loss changes when my input changes”, multiplying by its own local slope. Every intermediate result is computed once and reused, which is why the whole gradient costs about as much as one extra forward pass.

In one sentence: a derivative is the slope of a tiny straight piece of a curve, and the chain rule multiplies local slopes along any path from a knob to the loss.

A.3 The loss at the end: prediction minus answer, after all

Let the scores (logits) be $\mathbf{z}$, the probabilities $\mathbf{p} = \text{softmax}(\mathbf{z})$, and the right word be y . The loss is $L = -\ln p_y$. Write it out:

$$L = -\ln \frac{e^{z_y}}{\sum_j e^{z_j}} = -z_y + \ln \sum_j e^{z_j}.$$

The slope of the first part with respect to z_i is -1 if $i = y$ and 0 otherwise. The slope of the second part, by the chain rule, is $e^{z_i} / \sum_j e^{z_j} = p_i$. So

$$\frac{\partial L}{\partial z_i} = p_i - [i = y] \quad [1 \times V],$$

where $[i = y]$ is 1 for the right word and 0 otherwise. Here is the twist promised in Chapter 8: the correction *is* “prediction minus answer”, one probability at a time, with the answer written as a 1 for the right word and 0 for the rest. For our toy after **waste of time**: the slope on the **Sorry** score is -0.854 (negative, so walking downhill raises it) and on the **been** score 0.186 (positive, so it is lowered). Checked against probing: relative error 2.0×10^{-10} . Here are all three vectors for the toy:

	waste	of	time	not	very	useful	Sorry	,	this	has	been	.
$\mathbf{p}$	0.075	0.075	0.042	0.082	0.059	0.038	0.146	0.033	0.064	0.096	0.186	0.105
$\mathbf{y}$	0	0	0	0	0	0	1	0	0	0	0	0
$\mathbf{p} - \mathbf{y}$	0.075	0.075	0.042	0.082	0.059	0.038	-0.854	0.033	0.064	0.096	0.186	0.105

three vectors of length 12 (one number per word); the highlighted column is **Sorry**

Figure 70: Prediction, answer, and their difference, one number per word. Only the **Sorry** column of $\mathbf{p} - \mathbf{y}$ is negative: walking downhill raises that score and lowers all the others, each in proportion to the probability it was wrongly given.

Check yourself A.2. A model gives the right word probability 1. What is the slope of the loss on its score?

- (a) -1 .
- (b) 0: there is nothing left to correct.
- (c) 1.

A.4 Through attention

Recall the forward pass for one sentence (Chapter 3), with shapes:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \quad [T \times d_k]$$

$$S = \frac{1}{\sqrt{d_k}} QK^\top, \quad A = \text{softmax}_{\text{row}}(S) \quad [T \times T] \quad O = AV \quad [T \times d_k]$$

Suppose the layers above have handed back $G = \partial L / \partial O$, of the same shape as O . Each step backwards is the chain rule plus one rule about shapes: *the gradient of a quantity has the same shape as the quantity*. That rule alone tells you where every transpose goes.

gradient of	formula	shape
V	$\partial L / \partial V = A^\top G$	$(T \times T)(T \times d_k) = T \times d_k$
A	$\partial L / \partial A = GV^\top$	$(T \times d_k)(d_k \times T) = T \times T$
S	$\partial L / \partial S_{ij} = A_{ij}(\partial L / \partial A_{ij} - \sum_k A_{ik} \partial L / \partial A_{ik})$	$T \times T$
Q	$\partial L / \partial Q = \frac{1}{\sqrt{d_k}} (\partial L / \partial S) K$	$T \times d_k$
K	$\partial L / \partial K = \frac{1}{\sqrt{d_k}} (\partial L / \partial S)^\top Q$	$T \times d_k$
W_Q, W_K, W_V	$X^\top (\partial L / \partial Q)$, and the same for K and V	$d \times d_k$

The row for S comes from the slope of softmax, $\partial p_i / \partial s_j = p_i(\delta_{ij} - p_j)$ with $\delta_{ij} = 1$ when $i = j$ and 0 otherwise (probing check: 1.7×10^{-10}). Two consequences:

- *Masked cells pass back nothing.* A cell hidden by the mask has $A_{ij} = 0$, so its row- S formula gives 0. The future cannot be nudged by the past, in training as in prediction.
- *A made-up mind barely learns.* If one share in a row is close to 1 and the rest close to 0, every entry $p_i(\delta_{ij} - p_j)$ is close to 0, so the gradient reaching S (and from there W_Q and W_K) almost vanishes. This is the cost of the saturation measured in §3.8. With $d_k = 1024$ and random arrows, the size of the softmax's slope (all its entries combined) averages 0.0399 without the $1/\sqrt{d_k}$ and 0.3659 with it: about 9 times more responsive to learning.

Worked example: the real gradient of the toy. Take the loss of §A.3, the surprise at **Sorry** after **waste of time**. The scores were $\mathbf{z} = \mathbf{o}_{\text{time}} E^\top$, so going back one step, $\mathbf{g} = \partial L / \partial \mathbf{o}_{\text{time}} = (\mathbf{p} - \mathbf{y})E$, a 1×4 arrow. Only **time**'s row of O feeds this loss, so $G = \partial L / \partial O$ is that arrow in row 3 and zeros in rows 1 and 2. Then the table's first row gives $\partial L / \partial V = A^\top G$:

$$\begin{array}{c}
 E^\top \\
 \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
 \hline
 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
 \hline
 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
 \hline
 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
 \hline
 \end{array} \\
 4 \times 12
 \end{array}
 \times
 \begin{array}{|c|}
 \hline
 (\mathbf{p} - \mathbf{y})^\top \\
 \hline
 0.075 \\
 \hline
 0.075 \\
 \hline
 0.042 \\
 \hline
 0.082 \\
 \hline
 0.059 \\
 \hline
 0.038 \\
 \hline
 -0.854 \\
 \hline
 0.033 \\
 \hline
 0.064 \\
 \hline
 0.096 \\
 \hline
 0.186 \\
 \hline
 0.105 \\
 \hline
 \end{array}
 \begin{array}{|c|}
 \hline
 \mathbf{g}^\top \\
 \hline
 -0.471 \\
 \hline
 -0.311 \\
 \hline
 0.542 \\
 \hline
 0.326 \\
 \hline
 \end{array}
 \begin{array}{|c|}
 \hline
 4 \times 1 \\
 \hline
 \end{array}
 =
 \begin{array}{|c|}
 \hline
 12 \times 1 \\
 \hline
 \end{array}$$

highlighted row (row 1 of $E^\top$) · highlighted column ($\mathbf{p} - \mathbf{y}$):

$$(1)(0.075) + (0)(0.075) + (0)(0.042) + (0)(0.082) + (1)(0.059) + (0)(0.038) + (1)(-0.854) + (0)(0.033) + (1)(0.064) + (0)(0.096) + (1)(0.186) + (0)(0.105) = -0.471$$

$$\begin{array}{c}
 A^\top \\
 \begin{array}{|c|c|c|}
 \hline
 \text{waste} & 1.000 & 0.500 & 0.665 \\
 \hline
 \text{of} & 0.000 & 0.500 & 0.090 \\
 \hline
 \text{time} & 0.000 & 0.000 & 0.245 \\
 \hline
 \end{array} \\
 3 \times 3
 \end{array}
 \times
 \begin{array}{c}
 G \\
 \begin{array}{|c|c|c|c|}
 \hline
 0.000 & 0.000 & 0.000 & 0.000 \\
 \hline
 0.000 & 0.000 & 0.000 & 0.000 \\
 \hline
 -0.471 & -0.311 & 0.542 & 0.326 \\
 \hline
 \end{array} \\
 3 \times 4
 \end{array}
 =
 \begin{array}{c}
 \partial L / \partial V \\
 \begin{array}{|c|c|c|c|}
 \hline
 \text{waste} & -0.313 & -0.207 & 0.360 & 0.217 \\
 \hline
 \text{of} & -0.042 & -0.028 & 0.049 & 0.029 \\
 \hline
 \text{time} & -0.115 & -0.076 & 0.133 & 0.080 \\
 \hline
 \end{array} \\
 3 \times 4
 \end{array}$$

highlighted row (waste's column of A) · highlighted column (column 3 of G):

$$(1.000)(0.000) + (0.500)(0.000) + (0.665)(0.542) = 0.360$$

Figure 71: Two steps of the backward pass with the toy's real numbers. Top: $E^\top$ (one column per word, in vocabulary order) times $(\mathbf{p} - \mathbf{y})^\top$ gives $\mathbf{g}$, the gradient reaching **time**'s attention output (shown standing up). Bottom: $A^\top G$ gives the gradient of every value arrow. Row **waste** is the biggest because **time** gave **waste** the biggest share in the forward pass: whoever was listened to most gets corrected most. Our script checks the resulting $\partial L / \partial W_V = X^\top (\partial L / \partial V)$ against probing all 16 numbers.

We checked every row of the table on the toy of Chapter 3, with a random G , against probing each of the 48 numbers in W_Q , W_K , W_V : relative errors 5.6×10^{-10} , 1.1×10^{-9} and 1.7×10^{-10} .

Check yourself A.3. Why is it $A^\top G$ and not AG or GA ?

- (a) AG is also correct; the order does not matter.
- (b) The result must have the shape of V ($T \times d_k$), and $A^\top G$ is the only product of these two that does, while summing over the right index.
- (c) Transposes always appear in the backward pass.

A.5 Through a residual connection

Chapter 5 adds each block’s output to its input: $\mathbf{y} = \mathbf{x} + f(\mathbf{x})$ (He and colleagues, 2016). By the chain rule,

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} + \frac{\partial L}{\partial \mathbf{y}} J_f,$$

where J_f is the table of slopes of f . The first term passes the gradient back *untouched*. Without the “ $\mathbf{x} +$ ”, the gradient is multiplied by a J_f at every layer, and Chapter 1’s fading memory returns: we stacked layers of $\tanh(\mathbf{h}W)$ with small random weights and measured the size of the gradient that reaches the input.

layers	without residual	with residual
10	1.5×10^{-5}	7.50
50	1.2×10^{-26}	15.73

Without the residual path, after 50 layers essentially nothing reaches the first layer, so it cannot learn. With it, the signal arrives intact. Probing check on a 5-layer residual stack: 5.7×10^{-10} .

In one sentence: the gradient through attention follows from the chain rule and the shape rule, saturation and masking both show up as zeros in it, and a residual connection gives the gradient a path that is never multiplied away.

Looking back.

- (1) (Chapter 3) The transpose in $QK^\top$ arranges keys as columns. Where does a transpose reappear in the backward pass, and what does the shape rule say about it?
- (2) (Chapter 1) Which break of the RNN does the residual table above echo?

Exercises

Exercise A.1. Find the slope of $y = x^3$ at $x = 2$, first by the rise-over-run of a tiny step, then by expanding $(2 + h)^3$.

Exercise A.2. Probabilities over three words are $(0.7, 0.2, 0.1)$ and the right word is the second. Write the slope of the loss on each of the three scores. Which score goes up after a downhill step?

Exercise A.3. For a sentence of $T = 5$ tokens, what are the shapes of $\partial L / \partial S$ and $\partial L / \partial W_K$ when $d = 8$ and $d_k = 4$?

Exercise A.4. Explain in one sentence, using A.4, why a token never gets a gradient from a later token’s score.

Answers to Appendix A

Check yourself A.1. (b). The extra is exactly h : $(6h + h^2)/h = 6 + h$. It is not rounding (a), and the derivative is $2x = 6$ (c).

Check yourself A.2. (b). $p_y = 1$ gives $p_y - 1 = 0$. (a) would push further towards a word that already has all the probability.

Check yourself A.3. (b). The shape rule decides. (a) is false: AG sums over the wrong index. (c) is false in general: many backward steps have no transpose.

Exercise A.1. $(2 + h)^3 = 8 + 12h + 6h^2 + h^3$, so the rise-over-run is $12 + 6h + h^2 \rightarrow 12$. Probing gives 12.0000.

Exercise A.2. $\mathbf{p} - \mathbf{y} = (0.7, -0.8, 0.1)$. Only the second slope is negative, so only the second score goes up.

Exercise A.3. $\partial L / \partial S$ has the shape of S , 5×5 ; $\partial L / \partial W_K$ has the shape of W_K , 8×4 .

Exercise A.4. Masked cells have $A_{ij} = 0$, and the formula for $\partial L / \partial S_{ij}$ is multiplied by A_{ij} , so it is zero there.

Appendix B Reproducing every number

These notes make one promise: no number in them was typed by hand. How would you check a promise like that? This appendix tells you how. Every number comes from a script in the `code/` folder of the companion repository; every script writes its numbers into a small file of L^AT_EX macros, and the text only ever uses those macros. If you change a script and rebuild, the text changes with it.

B.1 Rebuilding everything

From the top folder of the repository:

Command	What it does
<code>make</code>	runs every number script whose source changed, then builds the PDF with <code>latexmk</code>
<code>make clean</code>	deletes the generated macro files and the build, so the next <code>make</code> redoes everything
<code>python3 lint/lint.py tex/ch*.tex</code>	runs the style checks of the design pack on every chapter
<code>python3 code/ch11_ablate.py</code> <code>--retrain</code>	retrains the Chapter 11 models (about 5 CPU minutes)
<code>python3 code/ch11_appB_index.py</code>	regenerates the script table below from the code itself

The slow step, training the small models of Chapters 8 and 11, caches its results in `tex/data_ch8/` and `tex/data_ch11/`, so an ordinary rebuild takes about a minute. To check that a rebuild is faithful, run `make clean`, retrain, rebuild, and compare the two PDFs’ text: with the fixed seeds the numbers should agree to the digits printed, except the training times.

B.2 Every script, what it writes, and what it checks

The table is generated from the scripts themselves. “Checks” counts the `assert` statements in each script: places where the script stops the build if something disagrees, for example a hand-calculation against NumPy, a transcript quote against the transcript, or our parameter calculator against the real Hugging Face model classes. In total there are 42 scripts and 98 checks.

Script	Purpose	Writes	Checks
<code>ch1_figs.py</code>	Chapter 1 picture: one pass over a batch makes a guess at every position (inputs vs. targets).	—	0
<code>ch1_numbers.py</code>	Small numbers used in Chapter 1’s prior-knowledge check and worked examples.	<code>num_ch1.tex</code>	0
<code>ch2_bpe.py</code>	Chapter 2 numbers: characters vs words vs BPE on the transcripts, and word vectors from co-occurrence.	<code>bpe_growth.dat</code> , <code>nn.json</code> , <code>num_ch2.tex</code>	1
<code>ch2_figs.py</code>	Pictures for Chapter 2: token ids, the one-hot lookup as a matrix product, and a batch of ids $\rightarrow$ arrows.	—	1
<code>ch2_mis.py</code>	Misconception inventories for Chapters 2 and 4; every transcript quote is asserted verbatim.	—	1
<code>ch3_figs.py</code>	Pictures of every vector and matrix product in Chapter 3 (numbers from <code>code/vocab.py</code>).	—	3
<code>ch3_numbers.py</code>	Extra numbers for Chapter 3: the churn-score picture of a dot product and the softmax-shift question.	<code>num_ch3.tex</code>	1

Script	Purpose	Writes	Checks
ch4_figs.py	Chapter 4 pictures: positions as matrices, the rotation acting on a pair, q and k before/after RoPE,	num_ch4.tex	3
ch4_rope.py	Chapter 4 numbers: order blindness, sinusoidal positions, and RoPE on the toy. Writes tex/num_ch4.tex.	config.json, num_ch4.tex, wavelengths.dat, waves.dat	5
ch5_figs.py	Chapter 5 pictures: one head's blend, each head's products, stitching the heads, the MLP, the residual	—	4
ch5_misconceptions.py	Misconception inventories for Chapters 5, 6 and 7 → design/mis_ch5.json, mis_ch6.json, mis_ch7.json.	—	1
ch5_numbers.py	Numbers for Chapter 5 (many heads, the MLP, the residual stream). Writes tex/num_ch5.tex.	identity.dat, num_ch5.tex	3
ch5_random.blocks.py	Random (untrained) transformer blocks for the depth experiments of Chapters 5 and 6.	—	0
ch6_figs.py	Vector and matrix pictures for Chapter 6. Writes tex/fig_ch6_*.tex (uses code/figlib.py).	—	3
ch6_numbers.py	Numbers for Chapter 6 (normalisation and depth). Writes tex/num_ch6.tex and tex/data_ch6/*.dat.	depth.dat, num_ch6.tex	4
ch7_figs.py	Vector and matrix pictures for Chapter 7. Writes tex/fig_ch7_*.tex (uses code/figlib.py).	—	4
ch7_numbers.py	Numbers for Chapter 7 (from scores to words). Writes tex/num_ch7.tex.	num_ch7.tex, temperature.dat	3
ch8_figs.py	Pictures of the matrices and vectors in Chapter 8 (and a helper reused by ch9_figs / appA_figs).	—	4
ch8_misconceptions.py	Misconceptions for Chapter 8, with transcript quotes checked verbatim. Writes design/mis_ch8.json.	—	1
ch8_numbers.py	All numbers and figure data for Chapter 8. Run after code/ch8_train.py.	landscape.dat, num_ch8.tex, train.json	7
ch8_toy.py	Chapter 8 toy: the skeleton model (embeddings → one attention head → tied unembedding),	—	0
ch8_train.py	Chapter 8: train a tiny decoder-only transformer on the lecture transcripts (character level).	curve.dat, overfit.dat, train.json	0
ch9_figs.py	Pictures for Chapter 9: the KV cache growing row by row, the new query meeting the cached keys,	—	2
ch9_mis.py	Write and verify design/mis_ch9.json and design/mis_ch10.json (quotes checked verbatim).	—	1
ch9_numbers.py	Chapter 9 numbers: KV cache on the toy, cost counts, and the memory bill of real models.	num_ch9.tex	2
ch10_figs.py	Chapter 10 figures: weight matrices drawn to scale (shapes read from the real model classes),	num_ch10.tex, num_ch10figs.tex	5
ch10_numbers.py	Chapter 10 numbers: parameter counts checked against transformers, shapes from a forward pass, FLOPs.	num_ch10.tex	5
ch10_verify.py		—	0
ch11_ablate.py	Chapter 11 ablations: tiny decoder-only transformers trained on the author's transcripts.	ablations.json	0
ch11_appB_index.py	Builds the script index for Appendix B from the code/ directory itself, so the appendix never goes	—	0
ch11_budget.py	Chapter 11 budget calculator and design scenarios. Writes tex/num_ch11.tex.	ablations.dat, ablations.json, num_ch11.tex	6
ch11_figs.py	Chapter 11 figures: the parameter count as a sum of labelled matrix blocks, drawn to scale, for	num_ch11.tex	4
ch11_mis.py	Misconceptions for Chapter 11, with transcript quotes checked verbatim.	—	1
appA_check.py	Appendix A: every derivative in the appendix, checked against finite differences.	num_appA.tex	7
appA_figs.py	Pictures for Appendix A: the gradient at the end (p - y) and the gradient of V (A ^T G) for the toy's	—	2
calc.py	Parameter, FLOP and KV-cache calculator from a config dict (Chapters 9, 10, 11).	config.json	0
corpus.py	Real-corpus numbers for Chapter 1: n-gram sparsity on the author's own lecture transcripts.	corpus.json, num_corpus.tex	0
figlib.py	Draw matrices, vectors and their multiplication as TikZ, with real numbers.	—	0
history.py	Numbers for the history section of Chapter 1 (all computed, none typed).	corpus.json, num_history.tex	2

Script	Purpose	Writes	Checks
<code>toy.py</code>	Toy transformer numbers for the sample chapter.	<code>num_toy.tex</code>	8
<code>toy_block.py</code>	The toy transformer block of Chapters 5-7, built on <code>code/vocab.py</code> ($d = 4$, 12-token vocabulary).	—	3
<code>vocab.py</code>	The toy vocabulary and embedding table shared by every chapter ($d = 4$).	—	1

Checks you can rely on. Ask yourself, for any number you doubt: what would have caught it if it were wrong? A few checks deserve a sentence each.

- The toy attention of Chapter 3 is computed twice, once with NumPy and once with exact fractions, and the two must agree to 10^{-12} .
- Every quote from a lecture transcript is checked to appear word for word in that transcript.
- The parameter calculator of Chapters 10 and 11 must equal the parameter count of the real model classes for GPT-2 (124,439,808) and Qwen3-0.6B (596,049,920), built without downloading any weights; two calculators written separately must agree.
- Derivatives in Appendix A are checked against small finite differences.
- Every chapter must pass `lint/lint.py` with no hits.

B.3 Software

Tool	Version used for this edition
Python	3.13.16
NumPy	2.5.3
PyTorch	2.14.1+cu130
Transformers	5.20.1
TeX	pdfTeX 3.141592653-2.6-1.40.25 (TeX Live 2023/Debian)

Training runs used one CPU thread (`torch.set_num_threads(1)`) and fixed seeds. A different machine can give slightly different losses in the last digit; the conclusions in §11.6 are stated against a noise level measured with two seeds for exactly this reason.

B.4 What needs the network, and what could not be done here

Building the notes needs no network at all. Three things would, and in the environment where this edition was made they were blocked:

- **Hugging Face** (huggingface.co): the published `config.json`, tokenizer and weights of Qwen3-0.6B. Until it is reachable, the Qwen3 configuration comes from the Qwen3 technical report (2025) and is marked as such in Chapter 10. Our parameter count for that configuration matches the real model class exactly, which checks our arithmetic but not the configuration itself.
- **Wikipedia** (en.wikipedia.org, dumps.wikimedia.org): the live page *Signs of AI writing*, against which the lint rules must be re-checked, and Simple English Wikipedia text as a

second corpus for Chapters 1 and 2. Until then, the corpus numbers come from the lecture transcripts only.

- **Large models:** anything that needs a GPU (running Qwen3-0.6B itself, longer training) was out of scope. The scripts are written so that the same code runs with a larger budget.

B.5 How we will know the notes teach

Numbers that regenerate show that the notes are *consistent*. Do they show that the notes *teach*? No. For that we need readers, and the plan before publication is:

1. **Pre-test and post-test** with 5 to 10 readers, at least two of whom have been out of education for a long time. Each test uses questions written to the chapter objectives. We report the normalised gain (Hake, 1998): the share of the possible improvement that each reader actually made.
2. **Item analysis** of every *Check yourself* question: how many readers got it right (questions everyone gets right or wrong teach us nothing), whether it separates stronger from weaker readers, and whether every wrong option was chosen by at least one reader. A wrong option nobody chooses means the misconception behind it was not real, and the option is replaced.
3. **Think-aloud readings** with 3 to 5 readers who say out loud what they are thinking as they read (Ericsson and Simon, 1993). Every place where a reader stalls, rereads or guesses wrongly goes into a list, and every item on the list is fixed and tested again.

Which of these would convince you, if you were the reader? The results, and the changes they cause, will be published with the next edition alongside an errata page.

References

- Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., & Sanghai, S. (2023). GQA: Training generalized multi-query transformer models from multi-head checkpoints. *EMNLP*.
- Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. arXiv:1607.06450.
- Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv:1409.0473 (ICLR 2015).
- Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3, 1137–1155.
- Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157–166.
- Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000). *How People Learn*. National Academy Press.
- Cauchy, A. (1847). Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendus de l'Académie des Sciences*, 25, 536–538.
- Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations. *Cognitive Science*, 13(2), 145–182.
- Cho, K., et al. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP*.
- Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *NeurIPS 2022*.
- Dunlosky, J., Rawson, K. A., Marsh, E. J., Nathan, M. J., & Willingham, D. T. (2013). Improving students' learning with effective learning techniques. *Psychological Science in the Public Interest*, 14(1), 4–58.
- Elhage, N., et al. (2021). A mathematical framework for transformer circuits. *Transformer Circuits Thread*, Anthropic.
- Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179–211.
- Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis: Verbal reports as data* (rev. ed.). MIT Press.
- Fan, A., Lewis, M., & Dauphin, Y. (2018). Hierarchical neural story generation. *ACL*.
- Firth, J. R. (1957). A synopsis of linguistic theory 1930–1955. In *Studies in Linguistic Analysis*. Blackwell.
- Fyfe, E. R., McNeil, N. M., Son, J. Y., & Goldstone, R. L. (2014). Concreteness fading in mathematics and science instruction. *Educational Psychology Review*, 26(1), 9–25.
- Gage, P. (1994). A new algorithm for data compression. *The C Users Journal*, 12(2).
- Geva, M., Schuster, R., Berant, J., & Levy, O. (2021). Transformer feed-forward layers are key-value memories. *EMNLP*.
- Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. (2017). On calibration of modern neural networks. *ICML*.
- Hake, R. R. (1998). Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1), 64–74.
- Haviv, A., Ram, O., Press, O., Izsak, P., & Levy, O. (2022). Transformer language models without positional encodings still learn positional information. *Findings of EMNLP*.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *CVPR*.
- Hendrycks, D., & Gimpel, K. (2016). Gaussian Error Linear Units (GELUs). arXiv:1606.08415.

- Henry, A., Dachapally, P. R., Pawar, S., & Chen, Y. (2020). Query-Key Normalization for Transformers. Findings of EMNLP 2020.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
- Hoffmann, J., Borgeaud, S., Mensch, A., et al. (2022). Training Compute-Optimal Large Language Models. arXiv:2203.15556.
- Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2020). The curious case of neural text degeneration. *ICLR*.
- Jelinek, F. (1997). *Statistical Methods for Speech Recognition*. MIT Press.
- Kaplan, J., McCandlish, S., Henighan, T., et al. (2020). Scaling Laws for Neural Language Models. arXiv:2001.08361.
- Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. *ICLR*.
- Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *ICLR*.
- Kneser, R., & Ney, H. (1995). Improved backing-off for m-gram language modeling. *ICASSP*.
- Levy, O., & Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. *NeurIPS*.
- Lionni, L. (1970). *Fish Is Fish*. Pantheon Books.
- Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. *ICLR*.
- Mayer, R. E. (2009). *Multimedia Learning* (2nd ed.). Cambridge University Press.
- Michel, P., Levy, O., & Neubig, G. (2019). Are sixteen heads really better than one? *NeurIPS*.
- Mikolov, T., Karafát, M., Burget, L., Černocký, J., & Khudanpur, S. (2010). Recurrent neural network based language model. *Interspeech*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *NeurIPS*.
- Mueller, C. M., & Dweck, C. S. (1998). Praise for intelligence can undermine children’s motivation and performance. *Journal of Personality and Social Psychology*, 75(1), 33–52.
- NVIDIA (2022). NVIDIA H100 Tensor Core GPU datasheet.
- Press, O., & Wolf, L. (2017). Using the output embedding to improve language models. *EACL*.
- Qwen Team (2025). Qwen3 technical report. arXiv:2505.09388.
- Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. OpenAI technical report.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. OpenAI technical report.
- Renkl, A., & Atkinson, R. K. (2003). Structuring the transition from example study to problem solving. *Educational Psychologist*, 38(1), 15–22.
- Richland, L. E., Kornell, N., & Kao, L. S. (2009). The pretesting effect: Do unsuccessful retrieval attempts enhance learning? *Journal of Experimental Psychology: Applied*, 15(3), 243–257.
- Robbins, H., & Monro, S. (1951). A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3), 400–407.
- Roediger, H. L., & Karpicke, J. D. (2006). Test-enhanced learning. *Psychological Science*, 17(3), 249–255.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536.

- Sennrich, R., Haddow, B., & Birch, A. (2016). Neural machine translation of rare words with subword units. *ACL*.
- Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27, 379–423, 623–656.
- Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell System Technical Journal*, 30, 50–64.
- Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations. *NAAACL*.
- Shazeer, N. (2019). Fast transformer decoding: One write-head is all you need. *arXiv:1911.02150*.
- Shazeer, N. (2020). GLU Variants Improve Transformer. *arXiv:2002.05202*.
- Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2021). RoFormer: Enhanced transformer with rotary position embedding. *arXiv:2104.09864*.
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *NeurIPS*.
- Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1), 59–89.
- Tippett, C. D. (2010). Refutation text in science education: A review of two decades of research. *International Journal of Science and Mathematics Education*, 8(6), 951–970.
- Touvron, H., et al. (2023). LLaMA: Open and efficient foundation language models. *arXiv:2302.13971*.
- Vaswani, A., et al. (2017). Attention is all you need. *NeurIPS*.
- Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2), 270–280.
- Wu, Y., et al. (2016). Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144*.
- Xiong, R., et al. (2020). On layer normalization in the transformer architecture. *ICML*.
- Yang, A., et al. (Qwen Team) (2025). Qwen3 Technical Report. *arXiv:2505.09388*.
- Zhang, B., & Sennrich, R. (2019). Root mean square layer normalization. *NeurIPS*.