Transformers and QKV Attention: A Primer

Luca Erzegovesi

2026-05-31

Table of contents

*First in the series "Understanding LLMs to use them better in management and finance." The three notes describe one machine, a Transformer language model, from three complementary angles. This note opens the machine up and lays out the **attention machinery**: how a Transformer moves information between words. The second note is about the **embeddings**, the vocabulary of vectors the machine reads from and writes to, and about how to look at a space with hundreds of dimensions without fooling yourself. The third note watches the **final step**, the moment all that work collapses into a single next word, which turns out to be a relentless search. Read in order, the three move from mechanism, to dictionary, to read-out.*

*This note is meant as the technical booklet that ships with an electronic appliance: it aims to give an accessible but complete view of the machinery and its inner workings. The end user normally need not open the booklet; the technician must. But in business applications of AI the boundary between technician and end user is blurring fast. Anyone who deploys these models in management or finance increasingly has to look inside the machine to judge what it can and cannot reliably do. This first note is accordingly longer and more abstract than the two that follow, which are focused and example-driven. The architecture it describes is the **Transformer**, introduced by Vaswani et al. (2017) in the paper whose title became a slogan, "Attention Is All You Need"; the technical terms are collected in Appendix C and the references at the very end.*

1. Transformers, an evolutionary step from neural networks

Before opening up the attention machinery, it helps to see where a Transformer *comes from*. A Transformer is not an exotic invention out of nowhere. It is one more step in a long line of neural networks that all do the same basic thing: convert input data, step by step, into a representation whose position in some space encodes the answer. What changes from one architecture to the next is *how* that conversion is organized. Attention is the organizational idea that made networks good at language.

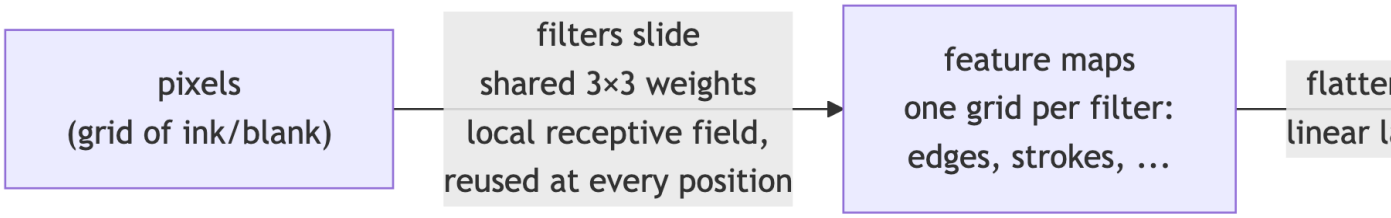
A network you may already picture, the OCR convolutional net

Think of a classic optical-character-recognition (OCR) network, the kind that reads a handwritten digit or letter from a small image. Its workhorse is the **convolution**. The input is a grid of pixels. A small **filter** (say a 3×3 patch of weights) slides across the image; at each location it multiplies the pixels under it by its weights and sums them into a single number. Sweeping the filter over the whole image produces a **feature map**, a new grid that lights up wherever the filter's little pattern (an edge, a curve, a stroke) is present. A layer has many such filters, so it produces many feature maps. Stack a few convolution layers (with pooling in between to shrink the grid), and the network builds up from edges $\rightarrow$ strokes $\rightarrow$ loops $\rightarrow$ whole-character shapes.

A clean version of the underlying operation carries over to attention. A convolution is a **weighted sum over a small local window** of the input: at each position the filter computes a dot product between its fixed weights and the pixels it currently covers. (It is a weighted *sum*, not strictly an average: the weights need not sum to one and are often negative, which is how a filter can

reward ink in one place and *penalise* it in another.) Two things are then layered on top. First, the same weights are reused at every position (**weight sharing**), so the operation is really one small pattern-detector swept across the whole image. Second, a layer stacks many such detectors and a later layer takes **weighted combinations of their feature maps**, and it is *those* combination weights, learned by training, that select which mixtures of low-level features best predict the correct character. So the operation splits in two: each convolution is a *local* weighted sum, and the *cross-feature* mixing that "selects the useful combinations" happens when later layers combine feature maps. (Appendix B works a full convolution through by hand.)

At the very end the features are flattened and a **softmax classifier** turns them into a probability over the possible characters.



Two properties make this work for images. First, each filter has a **fixed, local receptive field**: it only ever looks at a small neighborhood of pixels. Second, the *same* filter is reused at every position (**weight sharing**), so a stroke is recognized the same way wherever it sits on the page. Both properties are exactly right for images, because the clues needed to recognize a stroke are *local* and their meaning does not depend on *where* on the page they appear.

Why language breaks this, and why we need attention

Language does not have those two convenient properties. The piece of context a word depends on can be right next to it or hundreds of words back, and *which* earlier words matter depends on the **content**, not on a fixed offset. To resolve "it" you must find the noun it refers to, and that noun could be anywhere. A fixed, local filter cannot do this: it always looks in the same small window, regardless of what the sentence is about.

Attention is the fix. Instead of a fixed local window, an attention *head* computes, on the fly and for each token, *how much to pull from every other token*, and then pulls. The "receptive field" is no longer fixed by the architecture; it is decided at runtime, from the content, by the query–key matching detailed in Section 2.

> An attention head is the language analogue of a convolutional filter, except its receptive field is **learned, dynamic, and content-dependent** instead of fixed and local.

That single change, a receptive field set by the content rather than by the architecture, is what let neural networks finally handle long-range, content-driven dependencies (Vaswani et al., 2017), and it is the heart of the Transformer.

Attention is the *new* idea, though, not the *only* idea. A Transformer interleaves two kinds of block. The **attention heads** are the novelty just described, the content-driven, dynamic receptive field that moves information *between* positions. Alongside them sit ordinary **MLP** (multi-layer perceptron) blocks, the same fully-connected, learned weighted-combination machinery that does the cross-feature mixing in a CNN, except that here each MLP refines one token's representation *in place*, with no reference to its neighbours. The two blocks divide the labour cleanly: attention is the *only* channel that lets tokens talk to each other, while the MLP is where each token's representation is reshaped on its own. Sections 5 and 6 build both explicitly, and Appendix A shows what is lost if you switch the attention off and keep only the MLP.

What the output data actually *is*

The same description of what these networks produce fits both the OCR net and the language model.

In both cases the network turns its input into a **representation**: a vector whose *position in a high-dimensional space* encodes the answer. In OCR, the final feature vector's location says "this image sits in the region of the space that means the letter *e*." In a Transformer, the inputs are pieces of text, or **tokens**, and each token is represented by an **embedding**: a vector of numbers giving its coordinates in a high-dimensional "language space," where direction and proximity stand in for meaning. The vector the model processes and maintains for each token, the **residual stream** (formally introduced in Section 4), is best read as a **modified embedding**: it starts life as the raw token embedding and each layer nudges it to a new position that encodes everything the model has worked out about that token *in its context*. How a stream a few thousand numbers wide comes to hold many more distinguishable features than it has dimensions is the subject of the second note's section on distribution and superposition (Elhage et al., 2022).

So the output data are representations of entities, describing each entity's most likely position in a solution space. From that position the model reads out one of two things:

- **A single crisp solution.** The one best answer (take the highest-scoring class, i.e. the arg max, equivalently temperature $\to 0$).
- **A probability distribution over solutions.** A graded set of plausible answers with weights.

The readout is the same machine in both networks: a linear projection followed by a softmax. OCR projects the final feature vector onto the alphabet and gets a distribution over letters; a language model projects the final token representation onto the vocabulary and gets a distribution over *next tokens*. The structure is the same and the solution space differs. (Section 5 shows exactly how the final representation's **direction** encodes *which* tokens are likely and its **magnitude** encodes *how confident* the model is, the geometric version of "position in a solution space.")

The logic of the processing, params convert inputs into entity representations

The route from raw symbols to these context-aware representations is the through-line of this whole note:

1. **Look up.** Each input token ID is replaced by its embedding: a first, context-free guess at its position in the space (a row of the embedding matrix E).
2. **Read, transform, write, repeatedly.** A stack of parameterized blocks then reads the current representations and writes adjustments back. Two kinds of block alternate:
   - **Attention blocks** move information *between* tokens (the dynamic receptive field above), letting each token's representation absorb what it needs from the others.
   - **MLP blocks** refine each token's representation *in place*, one position at a time (detailed in Section 6).
3. **Read out.** After L such rounds (model **layers**) the representation is "finished," and the unembedding projects it into the solution space to give the crisp answer or the distribution.

The **parameters** (W_Q, W_K, W_V, W_O in attention, W_1, W_2 in the MLP) are frozen after training. They *are* the learned knowledge: they encode the rule for *how to move* each representation, step by step, from a bare embedding toward its correct final position. Training is just the search for parameter values that put every entity in the right place in the solution space.

> Because all the "thinking" lives in these incrementally-modified representations flowing through the residual stream, sharp questions can be asked about a trained model: *where* is a particular piece of behavior carried, and *which* block put it there? In small, fully-understood models one can even disable a single block and watch a specific behavior appear or vanish, which is the basis of the interpretability experiments these notes are meant to accompany (Appendix A sets out the method; the experiments themselves are in The clockwork and the dice and the notes that follow it in the ABC of language models series). The rest of this note builds the mechanism precisely enough to make those questions answerable.

2. The QKV trio

Every token, at every attention layer, produces three vectors derived from the token's current hidden state via three learned weight matrices (W_Q, W_K, W_V):

- **Q (query).** "what am I looking for?"
- **K (key).** "what do I offer to be matched against?"
- **V (value).** "what information do I carry, if matched?"

Attention is the operation that uses these three to mix information across tokens. For a given token's query Q, the model computes a similarity score against every previous token's K (a dot product, scaled, then softmaxed into weights). Those weights are then used to take a weighted sum of the corresponding V vectors. The result is the attention output for that token, a blend of *values* from earlier tokens, weighted by how well their keys matched this token's query.

In compact form (Q, K, and V are the corresponding W applied to tokens' embeddings, see Section 4):

$$\text{attention }(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) \cdot V$$

This formula is per head: $Q, K \in \mathbb{R}^{T \times d_k}$ and $V \in \mathbb{R}^{T \times d_v}$, giving an output in $\mathbb{R}^{T \times d_v}$. In multi-head attention, the same formula is applied h times in parallel on independent Q/K/V slices, and the h resulting $T \times d_v$ blocks are concatenated and then mixed by W_O into the residual. That is standard multi-head attention; some recent models let several heads share one K and V slice.

So Q and K together produce the *attention pattern* (who attends to whom, and how strongly), and V is what actually flows along those attention edges. Q combines with K to produce attention weights, which then select and blend the V vectors. V is the payload; QK is the routing.

3. Why only K and V are cached, not Q

During generation, when the model is producing token $N + 1$, it needs:

- **The Q of the new token only.** Q is computed fresh each step and discarded; it has no use beyond this single attention computation.
- **The K and V of every previous token.** The new token's Q has to attend back to all of them.

That's why the cache is called the **KV cache** and not "QKV cache". K and V are the *historical* state that accumulates as the conversation grows; Q is ephemeral, recomputed and thrown away every step.

This also explains the asymmetry between **prefill** (prompt processing) and **generation** (prompt continuation). During prefill, Q is computed for every input token too, but it is used immediately to compute attention for that token and then discarded. Only K and V get written into the cache for future reuse. Generation is just prefill-of-one-token, repeated, with the cache growing by one row of K and one row of V at each layer per step.

4. Dimensionality, a complete inventory

Let me define the symbols once and then track every shape through the network.

Symbol	Meaning	Typical value (GPT-2 small)	Typical value (modern 7B)
V	vocabulary size	50,257	32,000–150,000
L	number of transformer layers	12	32
T	sequence length (tokens in the context)	up to 1024	up to 128K
d_{model}	residual stream / embedding dimension	768	4096
h	number of attention heads	12	32
$d_k = d_v$	per-head Q/K/V dimension, often d_{model}/h	64	128
d_{ff}	MLP hidden dimension, usually $\sim 4 \cdot d_{\text{model}}$	3072	11008

Model parameters (frozen weights)

Weight	Shape	What it does
Token embedding E	$V \times d_{\text{model}}$	maps token IDs to vectors
Positional embedding (if learned)	$T_{\max} \times d_{\text{model}}$	adds positional information; a fixed sinusoidal encoding (Section 6) has no parameters
Per layer ℓ: $W_Q^{(\ell)}$	$d_{\text{model}} \times (h \cdot d_k)$	projects hidden → all heads' Q
Per layer: $W_K^{(\ell)}$	$d_{\text{model}} \times (h \cdot d_k)$	projects hidden → all heads' K
Per layer: $W_V^{(\ell)}$	$d_{\text{model}} \times (h \cdot d_v)$	projects hidden → all heads' V
Per layer: $W_O^{(\ell)}$	$(h \cdot d_v) \times d_{\text{model}}$	mixes head outputs back to residual
Per layer: MLP W_1, W_2	$d_{\text{model}} \times d_{\text{ff}}$, $d_{\text{ff}} \times d_{\text{model}}$	non-linear transformation
Final LayerNorm	d_{model}	normalization scale/shift
Unembedding U (often $E^\top$)	$d_{\text{model}} \times V$	maps final hidden → logits

Total parameters scale roughly as $12 \cdot L \cdot d_{\text{model}}^2$ (the result of Kaplan et al.).

Activations (recomputed every forward pass)

Activation	Shape	Notes
Input token IDs	T	integers in $[0, V)$
Initial residual stream $h^{(0)}$	$T \times d_{\text{model}}$	token embedding + positional encoding, row by row
Hidden state / residual stream $h^{(\ell)}$	$T \times d_{\text{model}}$	the running "meaning" per token, threaded through layers
$Q^{(\ell)}$, $K^{(\ell)}$, $V^{(\ell)}$ per head	$T \times d_k$ each, per head	computed from $h^{(\ell)}$ via W_Q, W_K, W_V
Attention scores $QK^\top$	$T \times T$ per head	the routing pattern

Activation	Shape	Notes
Attention weights (post-softmax)	$T \times T$ per head	row-stochastic, lower-triangular (causal mask)
Attention output (single head)	$T \times d_v$	weighted sum of V rows
Attention output (all heads concat)	$T \times (h \cdot d_v)$	usually $h \cdot d_v = d_{\text{model}}$
Output of W_O added to residual	$T \times d_{\text{model}}$	written back into the residual stream
MLP output	$T \times d_{\text{model}}$	also written back into the residual stream
Final hidden state (after layer L, LayerNorm)	$T \times d_{\text{model}}$	input to the unembedding
Logits	$T \times V$	one row per token position
Logits of the *last* token	V	the one that matters for next-token prediction
Probability distribution over vocabulary	V, sums to 1	softmax of last-token logits

From last-token logits to the next-token distribution

After the final layer, the hidden state of the last position $h_T^{(L)} \in \mathbb{R}^{d_{\text{model}}}$ passes through the final LayerNorm of the parameter table and is multiplied by the unembedding matrix:

$$\ell = \text{LN}\ (h_T^{(L)}) \cdot U \in \mathbb{R}^V$$

These are the **logits**: one real number per vocabulary token, unnormalized. Softmax converts them into probabilities:

$$p(\text{next token} = i \mid \text{context}) = \frac{\exp(\ell_i / \tau)}{\sum_{j=1}^{V} \exp(\ell_j / \tau)}$$

where τ is the sampling temperature. At $\tau \to 0$ this becomes greedy ($\arg\max$). At $\tau = 1$ you get the raw model distribution. The sampler then draws the next token from this distribution, appends it to the sequence, and the loop continues.

KV cache size

The KV cache has total size:

$$\text{KV cache size} = 2 \cdot L \cdot T \cdot h \cdot d_k \cdot (\text{bytes per element})$$

The factor 2 is for K and V together. For a 7B model with $L = 32$, $h = 32$, $d_k = 128$, at FP16 (2 bytes), 32K context:

$$2 \cdot 32 \cdot 32768 \cdot 32 \cdot 128 \cdot 2 \approx 17 \ \ \text{GB}$$

This is why long-context inference is memory-hungry.

5. A worked example with a tiny model and a tiny vocabulary

Let me build a deliberately small model so every matrix fits on a page, and walk one attention step end-to-end.

Setup

- Vocabulary size $V = 20$. Say the vocabulary is `{the, cat, dog, sat, ran, on, mat, floor, big, small, red, blue, fast, slow, a, and, ., is, was, .EOS}`, 20 tokens indexed 0–19.
- Embedding dimension $d_{\text{model}} = 5$.
- One attention head, so $d_k = d_v = 5$.
- One layer (MLPs are ignored here for clarity).
- Context: 3 tokens. The attention computed below is for the sequence `the cat sat`, token IDs $[0, 1, 3]$.
- No positional encoding and no LayerNorm. The attention pattern below depends on the tokens' content alone, and the read-out multiplies the last hidden state directly. Section 6 adds both.

Step 1: token embedding

The embedding matrix E is 20×5. After training it might look like (showing only the three rows needed):

```
E[0]  = [ 0.10, -0.20,  0.05,  0.40,  0.15]   "the"
E[1]  = [ 0.30,  0.50, -0.10,  0.20,  0.00]   "cat"
E[3]  = [-0.40,  0.10,  0.60, -0.30,  0.20]   "sat"
```

After looking up these three rows, the input to layer 1 is a 3×5 matrix, three tokens each as a 5-dimensional embedding:

```
X = [[ 0.10, -0.20,  0.05,  0.40,  0.15],
     [ 0.30,  0.50, -0.10,  0.20,  0.00],
     [-0.40,  0.10,  0.60, -0.30,  0.20]]
```

This is the initial residual stream $h^{(0)}$, shape $T \times d_{\text{model}} = 3 \times 5$.

Step 2: compute Q, K, V

The weight matrices W_Q, W_K, W_V are each 5×5 here.

General shape: $d_{\text{model}} \times (h \cdot d_k)$ for W_Q, W_K and $d_{\text{model}} \times (h \cdot d_v)$ for W_V. In this toy example there is one head with $d_k = d_v = d_{\text{model}} = 5$, so the shape collapses to 5×5. In a real multi-head model the per-head V slice has shape $T \times d_v$, not $T \times d_{\text{model}}$; the equality holds only after concatenating all h heads.

W_V is also called the IN matrix.

Take these values:

```
W_Q = [[ 1.0,  0.0,  0.0,  0.5,  0.0],
       [ 0.0,  1.0,  0.0,  0.0,  0.5],
       [ 0.0,  0.0,  1.0,  0.0,  0.0],
       [ 0.5,  0.0,  0.0,  1.0,  0.0],
       [ 0.0,  0.5,  0.0,  0.0,  1.0]]

W_K = [[ 0.8,  0.2,  0.0,  0.0,  0.0],
       [ 0.2,  0.8,  0.0,  0.0,  0.0],
       [ 0.0,  0.0,  0.9,  0.1,  0.0],
       [ 0.0,  0.0,  0.1,  0.9,  0.0],
       [ 0.0,  0.0,  0.0,  0.0,  1.0]]

W_V = [[ 0.5,  0.5,  0.0,  0.0,  0.0],
       [ 0.5, -0.5,  0.0,  0.0,  0.0],
       [ 0.0,  0.0,  1.0,  0.0,  0.0],
       [ 0.0,  0.0,  0.0,  1.0,  0.0],
       [ 0.0,  0.0,  0.0,  0.0,  1.0]]
```

Compute $Q = X \cdot W_Q$, $K = X \cdot W_K$, $V = X \cdot W_V$. Each is $3 \times 5 = T \times d_k$ in this single-head example. In general, per head, the shapes are $T \times d_k$ for Q and K, and $T \times d_v$ for V. In the three matrices below, each row is a token and each column indexes one coordinate of the per-head Q/K/V vector.

```
Q = [[ 0.300, -0.125,  0.050,  0.450,  0.050],   # for "the"
     [ 0.400,  0.500, -0.100,  0.350,  0.250],   # for "cat"
     [-0.550,  0.200,  0.600, -0.500,  0.250]]   # for "sat"

K = [[ 0.040, -0.140,  0.085,  0.365,  0.150],   # for "the"
     [ 0.340,  0.460, -0.070,  0.170,  0.000],   # for "cat"
     [-0.300,  0.000,  0.510, -0.210,  0.200]]   # for "sat"

V = [[-0.050,  0.150,  0.050,  0.400,  0.150],   # for "the"
     [ 0.400, -0.100, -0.100,  0.200,  0.000],   # for "cat"
     [-0.150, -0.250,  0.600, -0.300,  0.200]]   # for "sat"
```

(I have not rounded these. At this size the products divide out cleanly, so all three matrices are exact at three decimals; the rounding starts with the scores in Step 3.)

A **fourth weight matrix**, W_O (OUT matrix), is also a learned parameter of the attention block. Its job is to project the concatenated per-head attention outputs back into the residual stream's dimension and (in multi-head attention) to mix information across heads.

General shape: $(h \cdot d_v) \times d_{\text{model}}$. Here, with one head and $d_v = d_{\text{model}} = 5$, it collapses to 5×5. Note: $h \cdot d_v$ is the *concatenated* head-output dimension, which in most architectures equals d_{model} by design, which is why W_O usually looks like a $d_{\text{model}} \times d_{\text{model}}$ square in implementations.

```
W_O = [[ 1.0,  0.0,  0.0,  0.0,  0.0],
       [ 0.0,  1.0,  0.0,  0.0,  0.0],
       [ 0.0,  0.0,  1.0,  0.0,  0.0],
       [ 0.0,  0.0,  0.0,  1.0,  0.0],
       [ 0.0,  0.0,  0.0,  0.0,  1.0]]
```

(For simplicity W_O is set to the identity matrix here, so the attention output passes through unchanged. In a trained model W_O would be a learned dense matrix that performs the head-mixing and projection described above.)

Step 3: attention scores $QK^\top$

This is a 3×3 matrix where entry (i, j) is the dot product of token i's query with token j's key, a similarity score *between tokens* (not between heads), measuring how strongly token i wants to attend to token j.

```
QK^T = [[ 0.2055,  0.1175, -0.1490],
        [ 0.1028,  0.4325, -0.1945],
        [-0.1440, -0.2220,  0.6260]]
```

Now divide by $\sqrt{d_k} = \sqrt{5} \approx 2.24$:

```
QK^T / sqrt(5) = [[ 0.092,  0.053, -0.067],
                  [ 0.046,  0.193, -0.087],
                  [-0.064, -0.099,  0.280]]
```

Step 4: causal mask + softmax

Since this is autoregressive language modeling, each token can only attend to itself and earlier tokens. Mask the upper triangle to $-\infty$ (so softmax gives them weight 0):

```
masked = [[ 0.092,  -inf,  -inf],
          [ 0.046,  0.193,  -inf],
          [-0.064, -0.099,  0.280]]
```

Apply softmax row by row:

```
attention_weights = [[1.000, 0.000, 0.000],
                     [0.463, 0.537, 0.000],
                     [0.296, 0.286, 0.418]]
```

The third row says that when computing the output for "sat", the model attends about 30% to "the", 29% to "cat", and 42% to itself. This is the attention pattern, the "who attends to whom" that QK produced.

Step 5: weighted sum of V

Multiply attention_weights $\cdot V$. Per head this is $(T \times T) \cdot (T \times d_v) = T \times d_v$. With one head and $d_v = 5$, the result is 3×5:

```
attention_output = [[-0.050,  0.150,  0.050,  0.400,  0.150],   # "the" attends
only to itself
                    [ 0.192,  0.016, -0.031,  0.293,  0.069],   # "cat" blends
"the" + "cat"
                    [ 0.037, -0.089,  0.237,  0.050,  0.128]]   # "sat" blends
all three
```

The third row is the interesting one: it's a weighted blend (about 30%/29%/42%) of the three V vectors. This is the V payload being routed along the attention edges that QK set up. The output for "sat" now incorporates information drawn from "the" and "cat", which is how the model learns long-range dependencies.

(With multiple heads, there would be h such $T \times d_v$ blocks, one per head, computed in parallel from independent $Q^{(j)}, K^{(j)}, V^{(j)}$ slices. They get concatenated along the last axis into a single $T \times (h \cdot d_v)$ tensor before the next step. Section 6 carries out exactly this multi-head case numerically.)

Step 6: through W_O, back into the residual stream

The concatenated attention output (here just one head, so "concatenation" is a no-op) is projected by W_O:

$h^{(1)} = h^{(0)} + \text{attention\_output} \cdot W_O$

Shape arithmetic: $(T \times (h \cdot d_v)) \cdot ((h \cdot d_v) \times d_{\text{model}}) = T \times d_{\text{model}}$. In this single-head toy, that's $(3 \times 5) \cdot (5 \times 5) = 3 \times 5$.

This is the **residual addition**: the attention output is *added* to the previous residual stream, not replacing it. The hidden state $h^{(1)}$ is still shape $T \times d_{\text{model}} = 3 \times 5$.

In a real model, MLP layers would now run on top, also reading from and writing to the residual stream, and the whole thing would repeat L times. Here there is one layer, so the next step is the output. (Section 6 adds the MLP and a second layer explicitly.)

Step 7: final hidden state → logits → next-token distribution

Only the next token matters here, so take the last row of $h^{(1)}$: a single 5-vector, the final hidden state for "sat". With W_O set to the identity, that is $h^{(0)}$'s last row plus the attention output's:

```
h_last = [-0.363,  0.011,  0.837, -0.250,  0.328]
```

Multiply by the unembedding matrix U, shape 5×20 (one column per vocabulary token). The toy skips the final LayerNorm a real model applies first; the notes below describe it and Section 6 applies it:

$\ell = h_{\text{last}} \cdot U \in \mathbb{R}^{20}$

Result might look like:

```
logits = [-0.8, -0.2,  0.1,  0.5,  1.2,  2.4,  3.1,  2.8,  0.4, -0.1,
          -0.3,  0.0, -0.5, -0.4,  0.6,  0.2,  1.0, -0.2,  0.3, -1.5]
            the  cat  dog  sat  ran   on   mat floor  big small  red ...
```

The three highest logits are "on" (2.4), "mat" (3.1) and "floor" (2.8), the most likely continuations of "the cat sat". Applying softmax at temperature 1:

```
P("mat"   | "the cat sat") ≈ 0.31
P("floor" | "the cat sat") ≈ 0.23
P("on"    | "the cat sat") ≈ 0.16
P(others)                  ≈ remainder
```

This is the conditional distribution $p(\text{next} \mid \text{context})$ that the language model has been trained to approximate. The sampler picks a token from this (greedy would pick "mat"), appends it to the sequence, and the next forward pass begins.

Final step calculations in detail

The setup

After the last transformer layer (L), the residual stream is a $T \times d_{\text{model}}$ tensor, one d_{model}-dimensional vector per token position. Call it $h^{(L)}$.

At inference time, when you want to predict the *next* token, you only care about the last row: $h_T^{(L)} \in \mathbb{R}^{d_{\text{model}}}$. This vector is the model's final, fully-processed representation of "what comes next" given the entire context so far.

The two operations

1. Final LayerNorm (or RMSNorm). Before unembedding, virtually all modern transformers apply one last normalization to the residual stream. It rescales the vector so its components have controlled magnitude, then applies a learned per-dimension scale (and sometimes shift):

$\tilde{h} = \text{LayerNorm}\ (h_T^{(L)})$

The shape stays d_{model}. This step is easy to forget but it matters: without it, the magnitudes coming out of the residual stream would be wild, since every layer has been adding to it.

2. Unembedding (the linear projection to vocabulary). The normalized vector is multiplied by the unembedding matrix $U \in \mathbb{R}^{d_{\text{model}} \times V}$:

$\ell = \tilde{h} \cdot U \in \mathbb{R}^{V}$

Each column of U is a d_{model}-dimensional vector, one per vocabulary token. The matrix multiplication computes, for every vocabulary token i, the dot product between the final hidden state and that token's column:

$\ell_i = \tilde{h} \cdot U_{:,i}$

So each logit ℓ_i is literally a similarity score between the final hidden state and the i-th vocabulary token's representation in U. Tokens whose column points in the same direction as $\tilde{h}$ get high logits; tokens whose column points elsewhere get low logits.

Interpreting the similarity score

It is tempting to read this one step further: the model is trained to produce a last-token vector that *points in the same direction* as the embedding vectors of the likely next tokens. That intuition is sound, and it is exactly the geometry the third note develops in full, so the result is only stated here.

Training (gradient descent on the cross-entropy loss) shapes $\tilde{h}$ to have **high dot product with the columns of likely next tokens** and **low dot product with the rest**. Under weight tying ($U = E^\top$, below), those columns *are* the input embedding vectors, so the picture is almost literal. Two refinements keep it honest, both elaborated in Note 3: the vector does not point at *one* token but positions itself among *many* plausible continuations at once (which is how the model expresses uncertainty), and its **magnitude**, not just its direction, matters, because a longer vector sharpens the softmax (more confident) and a shorter one flattens it (less sure).

> The model learns to produce a final residual-stream vector whose **direction** encodes *which* next tokens are likely and whose **magnitude** encodes *how confident* the prediction is.

This is the foundation of the **logit lens** (nostalgebraist, 2020), which applies the unembedding U to *intermediate* residual streams to ask what the model would predict if forced to commit early; it works because the residual stream lives, throughout the network, in the same space the final readout uses. Note 3 turns this whole picture, the last step as a similarity search over the vocabulary, into its central theme.

Weight tying

In many models (GPT-2 and Gemma among them), U is the same matrix as the token embedding E, specifically $U = E^\top$. This is called **weight tying**. The embedding matrix maps *token ID* $\longrightarrow$ *vector* (each row is a token's representation); the unembedding maps *vector* $\longrightarrow$ *token logits* (each column is a token's representation). It is the same dictionary, used in two directions.

Weight tying cuts parameter count noticeably, since that matrix is $V \times d_{\text{model}}$ and can run to hundreds of millions of parameters for large vocabularies, and empirically it often improves quality (Press & Wolf, 2017; Inan et al., 2017).

Not all models tie weights (some recent ones keep them separate to give the unembedding more flexibility), but it's a very common default.

From logits to a distribution

Logits are just real numbers, unnormalized. They can be negative and they can be huge, and they don't sum to anything meaningful on their own. To turn them into a probability distribution over the vocabulary, apply softmax:

$$p_i = \frac{\exp(\ell_i/\tau)}{\sum_{j=1}^{V} \exp(\ell_j/\tau)}$$

where τ is the sampling temperature. The sampler then draws the next token ID from this distribution (or picks arg max for greedy decoding), and the next forward pass begins.

The compact picture

$$\underbrace{h_T^{(L)}}_{d_{\text{model}}} \xrightarrow{\text{LayerNorm}} \underbrace{\tilde{h}}_{d_{\text{model}}} \xrightarrow{\times U} \underbrace{\ell}_{V} \xrightarrow{\text{softmax}} \underbrace{p}_{V}$$

Three operations separate the last residual vector from a probability distribution over the entire vocabulary: a normalisation, one matrix multiplication, and a softmax. Only the multiplication by U touches the vocabulary. The residual stream did the heavy lifting; the unembedding just reads out the answer.

Two things to keep in mind

The residual stream "decides" everything before the unembedding. The unembedding is a fixed linear readout: it has no capacity to think, only to project. By the time you reach U, all the work of conditioning on the context has already been done by the L transformer layers writing into $h_T^{(L)}$. The unembedding is a translator from "internal representation space" to "vocabulary space."

During training, you compute logits for every position. At inference you only need the last row, but training computes ℓ for all T positions in parallel, each row predicting its successor, so the loss can be evaluated everywhere at once (teacher forcing). That's why the full logits tensor in training has shape $T \times V$, while at inference you typically only materialize the last row.

6. A second worked example with two heads, two layers and an MLP

The first example deliberately stripped the model down to a single head and a single layer, and skipped the MLP entirely. That was the right move for seeing attention clearly, but it hides three things a real Transformer does on every forward pass: it splits the work across several heads, it refines each token with an MLP, and it stacks layers so the residual stream is processed again and again. This example puts all three back, kept just small enough to do by hand.

This section is about dimensionality: watching the shape of the data at every step as it flows through two complete layers. The numbers are read off a trained model. I built a Transformer of exactly this shape on my research bench, 736 parameters, and trained it on a small English world: 64 sentence types over the vocabulary below, such as `the big cat sat on the red mat` and `the mat is blue`. Every matrix printed here is a rendering of that model's forward pass, produced by one script from the same weights; a second script re-renders them and compares them with

this page. The final token is therefore meaningful: after `<BOS> the cat sat` the model puts 0.997 on `on`, the only continuation its world allows. Every matrix is shown rounded to three decimals, so re-adding the displayed intermediates by hand may differ from a shown result by ± 0.001; the full-precision computation is consistent.

Setup

- Vocabulary size $V = 20$: the sixteen words of Section 5′s vocabulary, with `a`, `and`, `.` and `.EOS` replaced by four special tokens, `<PAD>`, `<BOS>`, `<EOS>` and `<SEP>`. Every sequence the model sees starts with `<BOS>`.
- $d_{\text{model}} = 6$.
- $h = 2$ heads, so the per-head dimension is $d_k = d_v = d_{\text{model}}/h = 6/2 = 3$.
- $L = 2$ layers, each one a full **attention block + MLP block**.
- MLP hidden size $d_{\text{ff}} = 8$. (Real models use $\sim 4 \cdot d_{\text{model}} = 24$; it is shrunk here so the matrices stay on the page.)
- **GELU** nonlinearity in the MLP: $\text{GELU}(z) = z \cdot \Phi(z)$, with Φ the standard normal distribution function. It is a smooth relative of ReLU: large negatives go to zero, large positives pass through, and values near zero are damped rather than cut. GPT-2 and most models since use it.
- Positions are added as sinusoidal encodings, the fixed sine and cosine pattern of Vaswani et al. (2017), so the positional part carries no parameters. Step 1 prints it and explains how one vector can hold both a word and its place.
- Normalisation before every sub-block. Each layer applies a **LayerNorm** to the residual stream before its attention block (LN_1) and again before its MLP block (LN_2), and a final LayerNorm precedes the read-out. This is the pre-LN arrangement of GPT-2 and later models. Section 5 skipped it. Every projection here also carries a bias vector (b_Q, b_K, b_V, b_O, b_1, b_2).
- The unembedding is tied: $U = E^\top$.
- Context: `<BOS> the cat sat`, so $T = 4$.

The parameter count: $20 \times 6 = 120$ for E (shared with U); 302 per layer, namely $4 \times 36 + 4 \times 6 = 168$ for the attention block, $6 \times 8 + 8 \times 6 + 8 + 6 = 110$ for the MLP and $2 \times 12 = 24$ for the two LayerNorms; and 12 for the final LayerNorm. In all, $120 + 2 \times 302 + 12 = 736$.

Here is the whole journey as a **shape table**. Everything below just fills in the numbers for these rows.

Step	Operation	Output shape
token IDs	lookup	$[T] = [4]$
$h^{(0)}$	embedding + positional	$T \times d_{\text{model}} = 4 \times 6$
$\text{LN}_1(h^{(0)})$	LayerNorm, row by row	4×6
Q, K, V (all heads)	$\text{LN}_1(h^{(0)})W_Q + b_Q$, etc.	4×6 each
per-head Q, K, V	slice into $h = 2$ blocks	4×3 each, $\times 2$
scores $QK^\top/\sqrt{d_k}$	per head	4×4 per head

Step	Operation	Output shape
weights	mask + softmax	4×4 per head
head output	weights $\cdot\, V$	4×3 per head
concat	join 2 heads	$4 \times (h \cdot d_v) = 4 \times 6$
attn write-back	concat $\cdot\, W_O + b_O$	4×6
h_{mid}	$h^{(0)} + \text{attn}$	4×6
$\text{LN}_2(h_{\text{mid}})$	LayerNorm, row by row	4×6
MLP pre-activation	$\text{LN}_2(h_{\text{mid}})W_1 + b_1$	$4 \times d_{\text{ff}} = 4 \times 8$
MLP activation	GELU	4×8
MLP output	$\cdot\, W_2 + b_2$	4×6
$h^{(1)}$	$h_{\text{mid}} + \text{MLP}$	4×6
… repeat for layer 2 …		
$h^{(2)}$	final hidden state	4×6
final LayerNorm	row by row	4×6
last row $\cdot\, U$	unembedding	$[V] = [20]$
softmax	distribution	$[V] = [20]$

Step 1: embedding + positional $\longrightarrow h^{(0)}$

Attention has no notion of order on its own: the scores of Step 3 compare vectors, and the same four vectors in any order would give the same set of scores. The position has to be written into the vectors. Two ingredients are added, row by row. First the embedding rows of the four tokens, looked up in E (20×6):

```
E[ids] = [[ 0.115,  0.381,  0.375, -0.196,  0.163, -0.128],   # <BOS>
          [-0.490,  0.522, -0.239,  0.656, -1.010,  0.604],   # the
          [ 0.747, -0.693, -0.211, -0.514, -0.593, -0.133],   # cat
          [ 0.048, -0.856,  1.067,  0.482,  0.983, -0.205]]   # sat
```

Second the **positional encoding**, one fixed row per position, the same for every token that ever sits there. Each pair of columns is a sine and a cosine of the position at a fixed frequency, and each pair is slower than the one before (1, $10000^{-1/3}$ and $10000^{-2/3}$ for the three pairs here):

```
PE = [[ 0.000,  1.000,  0.000,  1.000,  0.000,  1.000],   # pos 1
      [ 0.841,  0.540,  0.046,  0.999,  0.002,  1.000],   # pos 2
      [ 0.909, -0.416,  0.093,  0.996,  0.004,  1.000],   # pos 3
      [ 0.141, -0.990,  0.139,  0.990,  0.006,  1.000]]   # pos 4
```

Their sum is the initial residual stream $h^{(0)}$, shape 4×6. The first row is `<BOS>`, whose embedding the model learned like any other token's:

```
h⁰ = [[ 0.115,  1.381,  0.375,  0.804,  0.163,  0.872],   # <BOS> (pos 1)
      [ 0.351,  1.062, -0.193,  1.655, -1.008,  1.604],   # the   (pos 2)
      [ 1.656, -1.109, -0.118,  0.482, -0.588,  0.867],   # cat   (pos 3)
      [ 0.189, -1.846,  1.206,  1.472,  0.989,  0.795]]   # sat   (pos 4)
```

How one vector carries both the word and its place

The sum looks as if it destroys information: six numbers went in for the word, six for the position, and six come out. Three facts make both recoverable.

The positional rows are fixed and known. Every sequence the model ever sees has the row `[0.000, 1.000, 0.000, 1.000, 0.000, 1.000]` added at position 1, the row for position 2 at position 2, and so on. Training takes place against this constant background, so the weights can learn in which coordinates the position shows up and in which the word does.

The two signals live mostly in different coordinates. Read the `PE` matrix column by column. The first pair moves strongly from one position to the next, since its frequency is 1: the sine goes 0.000, 0.841, 0.909, 0.141 down the four rows. The second pair moves slowly (0.046 to 0.139 in column 3), and the third pair barely moves at all (column 5 reaches 0.006 at position 4). In this model, position is written almost entirely in columns 1 and 2, and columns 3 to 6 carry the word nearly undisturbed. In a large model d_{model} is in the hundreds, the frequencies run from 1 down to $1/10000$, and the same reading applies pair by pair: fast pairs mark fine position, slow pairs coarse position, and the embedding is trained to make its use of every coordinate compatible with the pattern added there.

Every read of the residual stream is a weighted sum of coordinates. W_Q, W_K and W_V are matrices, so a query is a weighted combination of the six numbers in a row. A head whose W_Q and W_K put their weight on columns 1 and 2 computes scores that depend on the positions of query and key and hardly at all on the words; a head that weights columns 3 to 6 does the opposite. Since $h^{(0)}$ is a sum, the score $q \cdot k$ splits into a word-to-word part, a position-to-position part and two mixed parts, and training decides how much of each a head uses. The sinusoidal choice adds one convenience: the row for position $p + k$ is the row for position p with each column pair rotated through an angle that depends on k alone. A head that wants to attend "one token back" can therefore learn a single $W_Q W_K^\top$ that scores position p against $p - 1$ highly for every p, instead of a separate rule per position.

That is the whole mechanism. The vector is one, the coordinates are shared, and the model is trained with the same positional pattern present in every example, so it learns projections that read the part each head needs. GPT-2 learns its position rows instead of fixing them, and most current models rotate the queries and keys by the position inside the attention scores (rotary position embedding), which is the same separation carried out one step later.

Step 2: LayerNorm, then project to Q, K, V and split into heads

Before the projections, LN_1 rescales each row of $h^{(0)}$ to mean 0 and variance 1 and applies a learned scale and shift per column. Section 5 described the operation at the read-out; in this model it also sits at the entrance of every block. The rows below are the same four tokens, normalised:

```
LN1(h⁰) = [[-1.315,  2.156, -0.671,  0.367, -1.235,  0.519],   # <BOS>
           [-0.250,  0.650, -0.968,  1.052, -1.972,  1.013],   # the
           [ 1.926, -1.750, -0.438,  0.263, -1.043,  0.678],   # cat
           [-0.269, -2.584,  0.741,  0.850,  0.522,  0.246]]   # sat
```

W_Q is now 6×6 (general shape $d_{\text{model}} \times (h \cdot d_k) = 6 \times 6$), and likewise W_K, W_V; each comes with a bias of length 6. Multiplying $Q = \mathrm{LN}_1(h^{(0)}) \cdot W_Q + b_Q$ gives a 4×6 matrix. Those 6 columns are two heads' worth of Q stacked side by side: columns 1–3 are head 1's query, columns 4–6 are head 2's. The vertical bar marks the split:

```
Q = [[-0.723, -1.114, -0.010 |  1.062, -0.797, -0.920],   # <BOS>
     [ 0.102, -1.531,  0.432 |  0.296, -0.261,  0.082],   # the
     [ 1.230, -0.892,  0.567 | -1.161,  0.795,  1.502],   # cat
     [ 1.078,  1.850,  0.871 | -0.828,  0.639,  0.471]]   # sat
      └──────── head 1 ────────┘   └──────── head 2 ────────┘
```

K and V are computed and split the same way. Each head now has its own 4×3 query, key, and value. This is what "multi-head" means: one matrix multiply, then carve the result into h independent lanes, each running the attention formula on its own slice.

Step 3: attention inside each head

Run Steps 3–5 of the previous example *separately* in each lane.

Head 1. Scaled scores $Q_1 K_1^\top / \sqrt{3}$, then causal mask and softmax:

```
scores₁ = [[ 0.033,   -inf,   -inf,   -inf],      weights₁ = [[1.000, 0.000,
0.000, 0.000],
           [-0.586, -0.787,   -inf,   -inf],  →               [0.550, 0.450,
0.000, 0.000],
           [-0.992, -0.477,  0.573,   -inf],                  [0.134, 0.224,
0.641, 0.000],
           [-0.122,  1.233,  2.304, -0.161]]                  [0.058, 0.226,
0.660, 0.056]]
```

Weighted sum of head 1's V gives head 1's output, shape 4×3:

```
head₁_out = [[-0.551, -1.031, -0.818],
             [-0.611, -0.515, -0.300],
```

```
             [-0.466,  0.863,  0.963],
             [-0.395,  1.036,  1.118]]
```

Head 2. Same procedure on head 2's slice, giving its own pattern and output:

```
weights₂ = [[1.000, 0.000, 0.000, 0.000],       head₂_out = [[-0.717,  0.794,
-0.650],
          [0.518, 0.482, 0.000, 0.000],  →             [-0.751,  0.848, -0.550],
          [0.043, 0.077, 0.880, 0.000],                [-0.312,  0.381,  0.148],
          [0.064, 0.088, 0.360, 0.488]]                [ 0.085, -0.111,  0.331]]
```

The two heads produce *different* attention patterns from the same input. In head 1 the last row, the query for `sat`, spreads its weight over all four positions, with the largest share on `<BOS>`. In head 2 the same query keeps almost all of its weight on `sat` itself, and head 2's row for `cat` looks mostly at `the`. Each head is free to specialize. This is the structural fact the interpretability experiments hinge on: distinct heads can do distinct jobs, and you can study them one at a time.

Step 4: concatenate the heads

Glue the two 4×3 head outputs back together, side by side, into one $4 \times (h \cdot d_v) = 4 \times 6$ matrix:

```
concat = [[-0.551, -1.031, -0.818 | -0.717,  0.794, -0.650],
          [-0.611, -0.515, -0.300 | -0.751,  0.848, -0.550],
          [-0.466,  0.863,  0.963 | -0.312,  0.381,  0.148],
          [-0.395,  1.036,  1.118 |  0.085, -0.111,  0.331]]
           └──────── head 1 ──────┘   └──────── head 2 ──────┘
```

Step 5: W_O and the residual addition

W_O has shape $(h \cdot d_v) \times d_{\text{model}} = 6 \times 6$. It mixes the two heads' information together and maps it back to the residual width. The product $\text{concat} \cdot W_O + b_O$ is the attention block's **write-back**, shape 4×6:

```
attn = [[ 0.546, -1.092,  0.051,  0.332, -0.380,  0.189],
        [ 0.509, -0.993,  0.280,  0.047, -0.001, -0.089],
        [ 0.063, -0.209,  0.641, -0.590,  0.919, -0.586],
        [-0.174,  0.150,  0.555, -0.612,  0.981, -0.517]]
```

Add it to the residual stream: $h_{\text{mid}} = h^{(0)} + \text{attn}$, still 4×6. The addition is to $h^{(0)}$ itself, not to its normalised copy; the LayerNorm output feeds only the projections.

```
h_mid = [[ 0.661,  0.289,  0.426,  1.136, -0.216,  1.061],
         [ 0.860,  0.069,  0.088,  1.702, -1.009,  1.515],
```

```
         [ 1.720, -1.318,  0.523, -0.108,  0.331,  0.280],
         [ 0.015, -1.696,  1.761,  0.861,  1.970,  0.278]]
```

Step 6: the MLP block

This is the piece the first example skipped. The MLP acts on the residual stream one token at a time: the same W_1, W_2 applied to every row independently, with no interaction between positions. (Appendix A turns on that fact.)

First LN_2 normalises each row of h_{mid}, the same operation as LN_1 above (not printed). Then expand from width 6 to width $d_{\text{ff}} = 8$ via W_1 (shape 6×8) and b_1: $\text{pre} = \mathrm{LN}_2(h_{\text{mid}}) \cdot W_1 + b_1$, shape 4×8:

```
pre = [[ 1.050,  0.446,  0.578, -1.281,  0.431, -2.046,  0.553,  1.590],
       [ 0.976,  0.396,  0.539, -1.394,  0.452, -2.216,  0.728,  1.603],
       [-1.437, -1.079, -1.626, -1.364,  1.130, -0.463, -0.477, -2.145],
       [-0.088,  0.054, -0.814,  0.739,  0.554,  2.029, -2.270, -1.688]]
```

Apply GELU element-wise. Compare the two matrices entry by entry: the large positive values pass almost unchanged, the large negatives go to a small negative number, and the values near zero shrink. ReLU would have set every negative to exactly 0. This is the model's only nonlinearity, and it is what lets the MLP do more than a plain matrix multiply:

```
act = [[ 0.896,  0.300,  0.415, -0.128,  0.287, -0.042,  0.392,  1.501],
       [ 0.815,  0.259,  0.380, -0.114,  0.305, -0.030,  0.558,  1.515],
       [-0.108, -0.151, -0.085, -0.118,  0.984, -0.149, -0.151, -0.034],
       [-0.041,  0.028, -0.169,  0.569,  0.393,  1.986, -0.026, -0.077]]
```

Then contract back from width 8 to width 6 via W_2 (shape 8×6) and b_2: $\text{mlp} = \text{act} \cdot W_2 + b_2$, shape 4×6:

```
mlp = [[-0.200,  0.599,  0.047, -0.827,  0.831, -1.312],
       [-0.207,  0.524,  0.040, -0.738,  0.779, -1.182],
       [-0.347,  0.005,  0.385,  0.233,  0.370, -0.163],
       [ 0.449,  0.818, -0.661, -0.246, -0.869,  0.414]]
```

Add it back to the residual stream: $h^{(1)} = h_{\text{mid}} + \text{mlp}$, shape 4×6. Layer 1 is now complete:

```
h¹ = [[ 0.460,  0.888,  0.473,  0.309,  0.615, -0.251],
      [ 0.653,  0.593,  0.127,  0.964, -0.230,  0.332],
      [ 1.373, -1.313,  0.909,  0.125,  0.701,  0.118],
      [ 0.464, -0.879,  1.100,  0.615,  1.101,  0.692]]
```

Note the shape went $6 \to 8 \to 6$ inside the MLP: the residual stream stays width 6 everywhere; the width-8 expansion happens *only inside* the block and is contracted away before the write-back. The residual stream's width d_{model} is invariant, and that constancy is what lets every block read and write the same $T \times d_{\text{model}}$ tape.

Step 7: layer 2 (same shapes, new weights)

Layer 2 has its own $W_Q, W_K, W_V, W_O, W_1, W_2$, biases and LayerNorms, but the shapes and the procedure are identical to layer 1. It reads $h^{(1)}$, runs two-head attention, writes back, runs its MLP, writes back, and produces $h^{(2)}$. Only the write-backs and the running result are shown:

```
attn² = [[-0.193, -0.321, -0.141,  0.137,  0.252,  0.461],
         [ 0.396, -0.107,  0.105, -0.304, -0.044, -0.196],
         [-0.346,  0.546,  0.043,  0.061,  0.114, -0.366],
         [-0.165,  0.341, -0.055,  0.038,  0.137, -0.253]]

h_mid² = [[ 0.267,  0.567,  0.331,  0.447,  0.867,  0.211],
          [ 1.049,  0.486,  0.232,  0.660, -0.274,  0.136],
          [ 1.027, -0.767,  0.952,  0.186,  0.815, -0.248],
          [ 0.299, -0.538,  1.045,  0.652,  1.237,  0.439]]

mlp² = [[-0.130,  0.288, -0.101,  0.487, -1.111,  0.423],
        [ 0.771,  0.198, -0.414, -1.116, -0.000,  0.343],
        [-0.376,  0.338, -0.123, -0.095,  0.405, -0.127],
        [-0.244,  0.731, -0.180,  0.027, -0.762,  0.424]]

h² = [[ 0.138,  0.854,  0.230,  0.933, -0.245,  0.634],   # <BOS>
      [ 1.821,  0.684, -0.182, -0.456, -0.275,  0.480],   # the
      [ 0.651, -0.429,  0.830,  0.091,  1.220, -0.376],   # cat
      [ 0.054,  0.193,  0.865,  0.679,  0.475,  0.864]]   # sat
```

Each of these is 4×6. After $L = 2$ layers the residual stream is still 4×6, the same shape it started as. Every layer reshaped the *contents*, never the *shape*.

Step 8: read out the last token

Exactly as in Section 5, with the one operation the toy skipped: the final LayerNorm. Take the last row of $h^{(2)}$ (the representation for `sat`, now informed by two full layers of attention + MLP), subtract its mean, divide by its standard deviation, then multiply coordinate by coordinate by the learned scale γ and add the learned shift β:

```
h²_last     = [ 0.054,  0.193,  0.865,  0.679,  0.475,  0.864]
γ           = [ 2.253,  2.390,  2.512,  2.328,  2.723,  2.476]
β           = [-0.339, -0.496, -0.744,  0.154, -0.418,  0.055]
LN(h²)_last = [-3.702, -3.005,  2.011,  1.323, -0.821,  2.758]
```

The six entries had a mean of 0.522 and a standard deviation of 0.313 before the norm. After it they span a range of about nine, because γ is close to 3 in every coordinate. This is the step that sets the scale of the logits (Section 5's final-step notes), and the four lines above are enough to redo it by hand.

Multiply by the unembedding $U = E^\top$ (shape $d_{\text{model}} \times V = 6 \times 20$) to get logits, then softmax. The vocabulary has twenty tokens, so this is the whole distribution:

```
logits = [-1.061, -1.561,  1.816, -1.123,  3.128, -1.668, -2.273, -7.735,
-7.690,  3.806,
            <PAD>   <BOS>   <EOS>   <SEP>     the     cat     dog     mat
floor     sat
              2.051, -5.294, -4.888, 10.255, -3.425, -3.823, -1.211, -0.718,
1.318,  1.405]
               ran      is     was      on     big   small     red    blue
fast    slow

probs  = [ 0.000,  0.000,  0.000,  0.000,  0.001,  0.000,  0.000,  0.000,
0.000,  0.002,
             0.000,  0.000,  0.000,  0.997,  0.000,  0.000,  0.000,  0.000,
0.000,  0.000]
```

The five largest, at three decimals:

rank	token	logit	probability
1	`on`	10.26	0.997
2	`sat`	3.81	0.002
3	`the`	3.13	0.001
4	`ran`	2.05	0.000
5	`<EOS>`	1.82	0.000

After `<BOS> the cat sat` the model puts 0.997 on `on`. In its training world every sentence that begins `the cat sat` continues with `on`, so this is the right answer; the runners-up, `sat` and `the`, share 0.002. The pipeline produced a clean $[V]$-shaped distribution, which sums to 1 before the two-decimal rounding, and this time the winner means something.

The same model on `<BOS> the cat is`

The prompt above tests a bigram: `on` follows `sat` in every sentence of the world, so a model that looked only at the current token would get it right. The copula is not a bigram. After `is` or `was` the predicate adjective belongs to the class of the subject noun: an animal is `big` or `small`, a surface is `red` or `blue`. A copula subject carries no adjective in this world, so both adjectives of the class are always possible after `is` and after `was`, and no sentence contradicts itself. The deciding token

is the subject noun, one place before the copula. It is never the copula itself, which is all a per-position model has at that position. On the copula prompt the model answers by the class rule, and swapping the subject noun swaps the answer:

prompt	1st	2nd	3rd
`<BOS> the cat is`	`big` 0.532	`small` 0.390	`floor` 0.022
`<BOS> the mat is`	`red` 0.540	`blue` 0.443	`mat` 0.005

Probing all 16 of the world's copula sentence types, the five held out of training included, the model answers inside the right class on 100% of them, with a combined probability on the two class adjectives between 0.92 and 0.99.

The world briefly had a second rule, and how it failed is worth reporting. An earlier version let the subject carry an adjective at the copula and required `is` to repeat it: `the big cat is big` was a sentence, `the big cat is small` was not. The model never learned it. Across four training configurations and an early checkpoint it answered the class's first adjective whatever the subject adjective was, getting the repeat right on between a half and three quarters of those sentence types, no better on the ones it had trained on than on the held-out ones. The behavioural metric I was watching scores the class, and `big` and `small` are both animal adjectives, so it read 1.000 throughout and reported nothing. I removed the sentences rather than publish a world the model contradicts, which is why a copula subject here carries no adjective. A metric is evidence about the thing it scores and about nothing else. (Those figures come from a corpus variant I did not keep, so they are the one set of numbers in this section that the script cannot re-render from the published model; they are recorded in the model's notes.)

Some attention head must be carrying the noun's identity forward to the copula's position, since the MLP cannot (Appendix A). Here are the attention weights of the query `is`, in each of the four heads, on `<BOS> the cat is`:

query `is` attends to →	`<BOS>`	`the`	`cat`	`is`
layer 1, head 1	0.059	0.197	0.630	0.114
layer 1, head 2	0.015	0.025	0.222	0.738
layer 2, head 1	0.035	0.240	0.633	0.092
layer 2, head 2	0.344	0.336	0.286	0.034

Two of the four heads put most of the copula's attention on `cat`. The largest share is 0.633, in layer 2, head 1, and layer 1, head 1 is close behind at 0.630. Layer 1, head 2 attends mostly to `is` itself, and layer 2, head 2 spreads its weight over `<BOS>`, `the` and `cat`. Appendix A measures which of these heads the rule depends on, and the answer does not follow the brightness of the `cat` column.

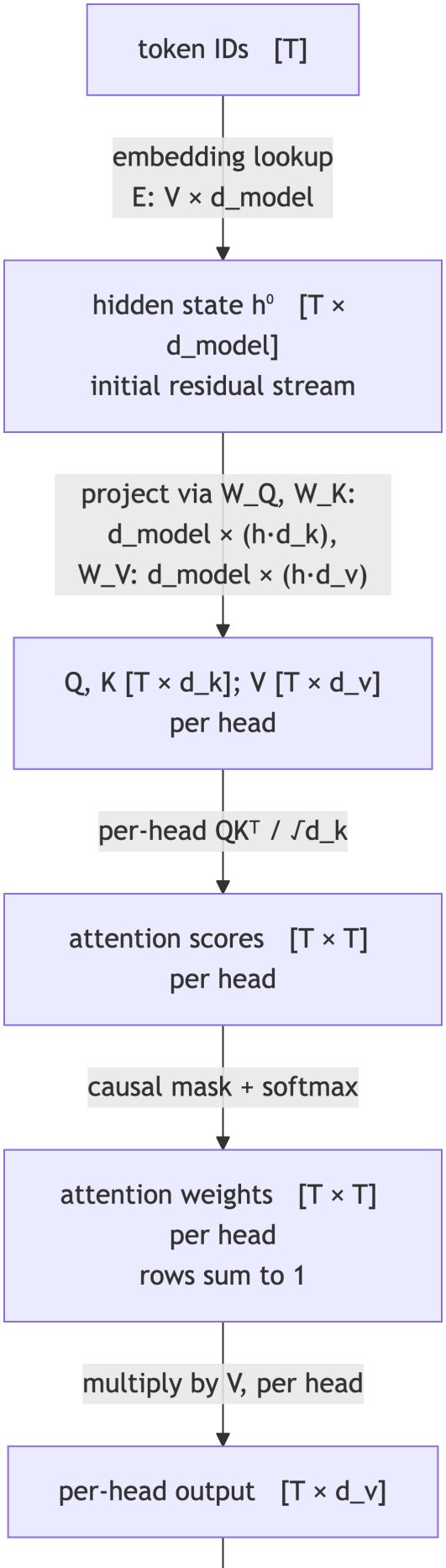
What this example added

Reading the shape table top to bottom, four things are now visible that the single-head example could not show:

1. **Heads are lanes.** One W_Q multiply produces all heads at once; the result is *sliced* into h independent $T \times d_k$ blocks, each running attention alone, then concatenated and remixed by W_O. The width bookkeeping is $d_{\text{model}} \to (h \cdot d_k) \to d_{\text{model}}$, and here $h \cdot d_k = 2 \cdot 3 = 6 = d_{\text{model}}$ exactly.
2. **The MLP is a per-token refinery.** It expands each token's vector to d_{ff}, applies GELU, contracts back, and adds the result to the residual, touching each position in isolation. The width bookkeeping is $d_{\text{model}} \to d_{\text{ff}} \to d_{\text{model}}$.
3. **Layers stack without changing shape.** The residual stream is a $T \times d_{\text{model}}$ tape that every block reads from and writes to additively. Stacking L layers just repeats the read–transform–write cycle; the shape is invariant from $h^{(0)}$ to $h^{(L)}$.
4. **Normalisation is everywhere and changes nothing about shape.** Five LayerNorms run in one forward pass of this model, two per layer and one at the end, each a per-row rescaling of a $T \times d_{\text{model}}$ block. None mixes positions.

7. Recap of the data flow

Shapes shown per head where the per-head structure matters; $h \cdot d_k = h \cdot d_v = d_{\text{model}}$ in most architectures, but they are conceptually distinct.


token IDs [T]
embedding lookup
E: V × d_model
hidden state h⁰ [T × d_model]
initial residual stream
project via W_Q, W_K: d_model × (h·d_k), W_V: d_model × (h·d_v)
Q, K [T × d_k]; V [T × d_v] per head
per-head QKᵀ / √d_k
attention scores [T × T] per head
causal mask + softmax
attention weights [T × T] per head
rows sum to 1
multiply by V, per head
per-head output [T × d_v]

Things to keep clear:

1. **Q is one-shot, K and V persist.** That's why the cache is "KV": Q for past tokens is never needed again.
2. **The residual stream is the spine.** Every attention and MLP block reads from it and writes back to it (additively). All the "thinking" passes through this $T \times d_{\text{model}}$ tensor.
3. **Per-head V has shape $T \times d_v$, not $T \times d_{\text{model}}$.** The full-width $T \times d_{\text{model}}$ shape only appears *after* concatenating the h heads (and only if $h \cdot d_v = d_{\text{model}}$, which is the usual but not mandatory choice). W_O is the operator that maps that concatenated block back into the residual.
4. **Only the last token's logits matter for next-token prediction at inference time.** During *training* you compute logits for every position (to predict each next token in parallel), but at generation time you only need the last.
5. **Logits are unnormalized; softmax produces the actual distribution.** Temperature, top-k, top-p sampling all operate on logits or the resulting distribution to control output diversity; the third note works through each of them.
6. **Attention is the only cross-token channel; the MLP is per-position.** Attention blocks are the *only* place where one token's representation can be influenced by another's. MLP blocks refine each token in isolation. So every *relational* thing a model does (agreement, coreference, copying, "don't repeat the previous speaker") must be carried by attention. Appendix A makes this precise by switching attention off; Appendix B works a small CNN by hand to sharpen the filter-vs-head contrast from Section 1.

Appendix A, what if attention heads were disabled?

A Transformer, stripped to its skeleton, is a stack of MLP blocks with attention blocks inserted between them, all communicating through one residual stream. A classic feed-forward neural network is essentially just the MLP part. So a natural question, and a useful one for understanding what attention *buys* you, is what happens if the attention is switched off. There are two clean ways to do it, and they arrive at the same destination.

The baseline, a network with only MLP blocks

Recall from Section 6 that an MLP block processes the residual stream one token at a time: the same W_1, GELU, W_2 applied to each row independently, with no reference to any other position. So a network built *only* from MLP blocks processes every token in complete isolation. It can learn a fixed mapping "this token (at this position) $\longrightarrow$ that output," i.e. per-token and position-conditioned statistics, but it has no mechanism for one token to influence another's representation. Whatever "sat" becomes, it becomes without ever consulting "the" or "cat."

Method 1: remove the attention layers entirely

Delete the attention sub-blocks and wire the embeddings straight into the MLP stack. The update rule becomes simply

$$h^{(\ell+1)} = h^{(\ell)} + \text{MLP}\ (h^{(\ell)}), \quad \text{per token, no mixing.}$$

This *is* the pure-MLP baseline. In the running example, the representation of "sat" can now never absorb anything from "the" or "cat": each column of the residual stream is processed down its own private pipe. A **relational** rule is therefore impossible: anything that requires *comparing two positions* (for instance "the next item must differ from the current one") cannot be expressed, because the two positions never meet.

Method 2: keep the architecture, make attention transparent

The second way disables attention *without deleting anything*. Keep the full Transformer architecture, all the W_Q, W_K, W_V, W_O machinery, but fabricate the weights so that the attention block writes nothing into the residual stream.

Recall the write-back from Section 6: the attention block's contribution is $\text{attn} = \text{concat}\ (\text{head outputs}) \cdot W_O$, and it is *added* to the residual. So set

$$W_O = 0 \quad (\text{equivalently} \quad W_V = 0).$$

Now, no matter what attention pattern $QK^\top$ computes, the value added to the residual is the zero vector:

$$h_{\text{mid}} = h^{(\ell)} + 0 = h^{(\ell)}.$$

The residual stream passes through the attention block untouched. The Q/K/V machinery still runs and still computes attention patterns, but those patterns are *transparent*: they have no effect on anything downstream. Functionally, the model is once again the pure-MLP network of Method 1.

(A subtler "identity" fabrication is available: make each token attend only to itself, so attention copies each value through. That doesn't give transparency: copying each token's value and adding it back *doubles* the contribution rather than leaving the stream unchanged. True transparency means the block adds zero, which is what zeroing the write-back achieves.)

Both roads lead to the same place

Whether you disable attention by deletion (Method 1) or by making it transparent (Method 2), the Transformer collapses to a **per-position MLP network**, a stack that refines each token in isolation and can never move information between positions. This is the precise sense in which:

> **Attention is the only channel through which tokens communicate.** Everything *relational* a language model does must be carried by attention, because it is the only operation that moves information across positions.

That is also why the tiny-model story these notes accompany is about attention *heads* specifically: if a rule relates one token to another, the circuit that enforces it has to live in the attention machinery, because there is nowhere else for it to be.

On the trained model of Section 6 this is a measurement. Both constructions were run on `<BOS> the cat is`, and the subject-noun swap of Section 6 was run with attention off and with attention on. The table gives the largest absolute difference over the twenty logits:

check	max abs Δ over the 20 logits
Method 2 ($W_O = 0$, $b_O = 0$ in both layers) against Method 1 (attention blocks deleted)	0 (identical to the last bit)
one head's rows of W_O zeroed against the engine's own head-ablation routine	0 (identical to the last bit)
attention off: `<BOS> the cat is` against `<BOS> the mat is`	0 (identical to the last bit)
attention on: the same swap	10.995

With attention off, the model gives `<BOS> the cat is` and `<BOS> the mat is` the same twenty logits, to the last bit: nothing at the copula's position can depend on any token before it. With attention on, the two prompts differ by up to 10.995 in a single logit, and the answers fall on opposite sides of the rule:

prompt	attention	top token	its probability	p(`big`) + p(`small`)
`<BOS> the cat is`	on	`big`	0.532	0.922
`<BOS> the mat is`	on	`red`	0.540	0.000
`<BOS> the cat is`	off ($W_O = 0$)	`sat`	0.629	0.000
`<BOS> the mat is`	off ($W_O = 0$)	`sat`	0.629	0.000
`<BOS> the cat is`	on, layer 1 head 1 removed	`the`	0.989	0.000

The pure-MLP network answers `sat` on both prompts, at 0.629, which is not an adjective at all. That number says nothing about what a per-position model could learn: these MLP weights were trained with attention present, so removing attention leaves a damaged network, not a trained per-position one. What the measurement shows is the identity, and the identity is exact.

Ablation, and why Method 2 is the important one

Method 2 has a feature deletion lacks: it is selective. The write-back matrix W_O is organized in blocks, one slice of rows per head (Section 6, Step 5). Zero out just one head's slice, and you make *exactly that head* transparent while leaving every other head working. Re-run the model, measure what changes, and you have a causal probe: *what does this one head actually do?*

This per-head version of Method 2 is called **ablation**, and it is one of the main tools of mechanistic interpretability, alongside the sparse autoencoders that try to read a layer's features off a residual stream directly (Cunningham et al., 2023). On the Section 6 model it was run for each of the four

heads in turn. The measure is **rule accuracy**: the fraction of copula positions, over the eight sentence types held out of training, where the top prediction is an adjective of the subject noun's class. Five of those eight types carry a copula, so the measure moves in fifths.

head removed (its rows of W_O set to 0)	rule accuracy on the 8 held-out types
none (the trained model)	1.00
layer 1, head 1	0.00
layer 1, head 2	0.00
layer 2, head 1	0.60
layer 2, head 2	0.60

Both layer-1 heads are load-bearing: zeroing layer 1, head 1 or layer 1, head 2 takes rule accuracy to 0.00. With either head's three rows of W_O set to zero, accuracy on the held-out types falls from 1.00 to 0.00, and on `<BOS> the cat is` the top prediction becomes `the` (the last row of the table above). Removing a layer-2 head costs two of the five decisions. Layer 2, head 1 carries the largest attention weight on `cat` of any head, and the rule survives its removal at 0.60.

What the ablation is actually measuring here

A single-head ablation is meant to be selective: one head transparent, every other head working, and any change attributable to that head. On a model this small it is only half that. Scoring every next-token decision of the corpus under each configuration, split into the copula slots and the slots a model reading only the current token could answer, gives the scope of the damage:

configuration	top-1 on the 432 slots a bigram could answer	right class on the 16 copula slots
none (the trained model)	0.699	1.000
layer 1 head 1 removed	0.417	0.000
layer 1 head 2 removed	0.431	0.000
layer 2 head 1 removed	0.662	0.500
layer 2 head 2 removed	0.690	0.500
attention off	0.250	0.000

Removing either layer-1 head takes the bigram slots from 0.699 to 0.417 and 0.431. Those predictions need no attention at all, so that fall is damage the ablation was not aiming at, and the rule accuracy of 0.00 in the previous table is partly collateral rather than the excision of a rule-specific circuit. The layer-2 ablations are the clean case: they leave it at 0.662 and 0.690 while halving the class accuracy. Four heads and six dimensions leave no spare capacity, so a probe that discriminates on a larger model is blunt on this one. The trained model reads 0.699 rather than 1.000 on those slots because many of them are genuinely ambiguous: after `the`, five different tokens can follow.

That is a limit of the demonstration, not of the method, and it is why the findings below are stated on a larger model. The three are measured in full in The clockwork and the dice, the opening note of the ABC of language models series, on a model small enough that the ground-truth rules are known. This note points at those experiments rather than surveying the literature:

- **A rule can rest on a single head.** Ablate that one head and a behavior the model performed perfectly collapses; ablate any other head and nothing changes. In that post one layer-0 head copies the token the chain rule calls for: zero its W_O block and the top-1 prediction flips, with the probability of the right token falling from 0.999 to 0.0005, while zeroing an inert head instead leaves it at 0.999. The behavior was carried, causally, by one specific lane in one specific layer.
- **And a rule can rest on no single head.** The same model's other rule, "call anyone except yourself", has no such part. Ablating each of the eight heads in turn, the largest single effect still leaves rule accuracy at 0.860, all four layer-0 heads carry a piece, and no head's removal switches the rule off. That is a negative result with a precise scope: no single head, under single-head zero-ablation. Head pairs, feed-forward neurons and directions in the residual stream were not searched.
- **Attention patterns can mislead.** A head whose attention *looks* like it implements a rule (say almost all of its weight sits on the relevant earlier token) may, when ablated, turn out to change nothing: it was not load-bearing. Conversely a head with a messy, unremarkable-looking pattern may be the one holding the rule. The attention pattern tells you what a head looks at; only ablation tells you what it does.

The point about misleading patterns is where intuition goes wrong: you cannot read a head's *function* off its attention picture. You have to switch the head off, Method 2 one head at a time, and watch what breaks. Disabling attention, far from being a destructive curiosity, is therefore the single most useful tool for finding out where in a network a behavior lives, which is the question the accompanying tiny-language-model experiments are built to answer.

Appendix B, a convolutional OCR pass by hand

Section 1 sketched the OCR convolutional network in words. Here is one run through, with numbers small enough to check by hand, so the filter is as concrete as the head. The two are then laid side by side.

The image

Take a tiny 5×5 grayscale image. Each pixel is 0 (blank) or 1 (ink). This one shows a **vertical stroke** down the middle column, the kind of mark that distinguishes, say, a `1` or the spine of a `T`:

```
       col: 0 1 2 3 4
row 0:      0 0 1 0 0
row 1:      0 0 1 0 0
row 2:      0 0 1 0 0
row 3:      0 0 1 0 0
row 4:      0 0 1 0 0
```

One filter, sliding

A filter is a small fixed grid of weights. Here is a 3×3 **vertical-stroke detector**: it rewards ink in its center column and punishes ink on either side, so it responds most strongly to a vertical line.

```
F_vert = [[-1,  2, -1],
          [-1,  2, -1],
          [-1,  2, -1]]
```

To convolve, slide this filter over every 3×3 window of the image; at each stop, multiply overlapping cells and sum to a single number. A 5×5 image with a 3×3 filter has $3 \times 3 = 9$ valid stops, so the output (the **feature map**) is 3×3.

Look at two stops to see the mechanism:

Top-left window (rows 0–2, cols 0–2). The stroke is off to the right, so the filter's center column sits on blanks:

```
window      = [[0,0,1],     elementwise·F_vert, summed:
               [0,0,1],     each row: 0·(-1) + 0·(2) + 1·(-1) = -1
               [0,0,1]]     three rows → -3
```

Top-center window (rows 0–2, cols 1–3). Now the stroke lines up under the filter's center column:

```
window      = [[0,1,0],     each row: 0·(-1) + 1·(2) + 0·(-1) = +2
               [0,1,0],     three rows → +6
               [0,1,0]]
```

Do this at all nine stops and you get the feature map. The center column lights up (+6); the flanks are suppressed (–3):

```
conv = [[-3,  6, -3],
        [-3,  6, -3],
        [-3,  6, -3]]
```

ReLU, then pooling

Apply **ReLU** (zero out negatives): the map keeps only the positive evidence "a vertical stroke is here":

```
relu(conv) = [[0, 6, 0],
              [0, 6, 0],
              [0, 6, 0]]
```

Then **max-pool** with a 2×2 window to shrink the grid and add a little position tolerance (each output cell is the max of a 2×2 patch). The 3×3 map becomes 2×2:

```
pool = [[6, 6],
        [6, 6]]
```

The pooled map is uniformly high: the filter reports "vertical stroke, present." Pooling means it would keep reporting that even if the stroke shifted a pixel, so the detector is now slightly *position-invariant*, exactly the property Section 1 said convolution buys you.

A layer has many filters

A real layer applies *many* filters in parallel, each producing its own feature map. Add a second filter, a **horizontal-stroke detector** (ink rewarded in the center *row*):

```
F_horiz = [[-1, -1, -1],
           [ 2,  2,  2],
           [-1, -1, -1]]
```

Run it on the *same* vertical-stroke image and every window cancels to zero: there is no horizontal ink to reward:

```
conv = relu = pool = all zeros
```

So the two filters disagree, informatively: the vertical detector fires, the horizontal detector is silent. That contrast is the feature the classifier uses.

Flatten and classify

Flatten the two pooled maps into one feature vector (four numbers per filter, eight in all):

```
features = [6, 6, 6, 6,   0, 0, 0, 0]
            └ vertical ┘   └ horizontal ┘
```

A final linear layer reads these features into one score per class (let the "vertical" class sum the vertical-filter features and the "horizontal" class sum the horizontal ones), and softmax turns the scores into probabilities:

```
logits = [24,  0]            # vertical, horizontal
softmax ≈ [1.00, 0.00]       # "this is a vertical stroke"
```

The network has converted a grid of pixels, step by step, into a point in a tiny two-class solution space, and read out a crisp answer, the same arc as the language model, just over characters instead of next tokens. (The probability is emphatic because this is a noise-free toy; on real handwriting the distribution would be softer.)

Filter vs head, side by side

A filter and a head are both *feature detectors that get reused across positions*, but they differ in the one respect that matters for language:

	Convolutional filter (CNN)	**Attention head** (Transformer)
What it stores	a fixed 3×3 pattern of weights	three projections W_Q, W_K, W_V (and shares W_O)
Receptive field	**fixed and local**, always the same small window	**dynamic and global**, chosen at runtime, can reach any earlier token
How it matches	slides the *same* pattern over every position; fires where the pixels match it	computes, from the content, a query–key similarity to decide *which* positions to read
What "reuse across positions" means	weight sharing: identical weights at every location	the same W_Q, W_K, W_V at every position, but the *attention pattern* is recomputed per input
Output at a position	one activation = how well the local patch matches the pattern	a weighted blend of other positions' **V** payloads
Set by the data…	at **training** time (the filter weights are learned, then frozen)	at training time *and* **at run time** (the pattern depends on the actual tokens)

The crucial row is *receptive field*. The vertical filter above can only ever see a 3×3 patch; to relate ink in the top-left corner to ink in the bottom-right, a CNN must stack many layers until their windows overlap. That is fine for images, where the clues are local. A head pays no such toll: the query for one token can match the key of a token hundreds of positions away in a *single* step, and *which* token it matches is decided by the content, not fixed by the architecture. That is the whole reason attention displaced convolution for language. Looping back to Appendix A, it is also why a relational rule in a language model lives in a *head*: the head is the only component whose reach is wide enough, and content-driven enough, to relate one token to another.

Appendix C, Glossary

For readers who would like the basics or a refresher. Terms are grouped roughly by where they first appear; Notes 2 and 3 carry their own glossaries for the terms specific to them.

Token, vocabulary

A **token** is the unit of text the model reads, a whole word in the toy examples here, usually a subword piece in production models. The **vocabulary** (V) is the fixed set of all possible tokens (20 in Section 5′s toy, 50,257 in GPT-2).

Embedding, embedding matrix E

The vector of real numbers that represents a token, its coordinates in the model's high-dimensional "language space." The **embedding matrix** E has one row per vocabulary token; "looking up" a token means reading its row.

Residual stream

The running vector the Transformer maintains for each position and updates layer by layer; every attention and MLP block *adds* its output to it. It is the only place the model's "thinking" lives, and the prediction is read from its final state.

Attention head, Q / K / V

A sub-mechanism that, for each token, decides how much to read from every earlier token. It does so with three learned projections of the residual: the **query** ("what am I looking for?"), the **key** ("what do I offer to match against?"), and the **value** ("what payload do I carry if matched?"). Query–key similarity sets the attention pattern; the values are what flow along it.

Multi-head attention

Running h attention heads in parallel on independent slices of Q/K/V, then concatenating their outputs and mixing them with W_O. Each head can specialise in a different relation.

W_Q, W_K, W_V, W_O

The four learned weight matrices of an attention block: three that project the residual into queries, keys and values, and W_O that maps the concatenated head outputs back into the residual stream.

MLP block (W_1, W_2, GELU or ReLU)

The fully-connected "feed-forward" sub-layer that refines each token's vector *in place*. It expands the vector to a wider hidden size via W_1, applies a non-linearity (**GELU** in the trained model of Section 6; **ReLU**, which zeroes negatives, in Appendix B's convolutional net), and contracts back via W_2. It never mixes information across positions.

Logits, softmax, temperature

The model's raw, unnormalised scores over the vocabulary are **logits**. **Softmax** turns them into a probability distribution (exponentiate, then normalise to sum to one). **Temperature** τ rescales the logits before softmax: low τ sharpens the distribution (greedy at $\tau \to 0$), high τ flattens it.

Unembedding U, weight tying

The matrix U that maps the final residual vector to one logit per vocabulary token; each column is a token's representation. **Weight tying** sets $U = E^\top$, the same dictionary used for input lookup and output scoring (standard in GPT-2, Gemma, and the toys here).

LayerNorm / RMSNorm

A normalisation applied to the residual vector, row by row, that rescales its components to a controlled magnitude and applies a learned per-dimension scale and shift: for a row x with mean μ and standard deviation σ, $\mathrm{LN}\ (x) = \gamma \odot (x - \mu)/\sigma + \beta$, with γ and β learned per coordinate.

In the pre-LN arrangement of Section 6 it runs before every attention block, before every MLP block, and once more just before the unembedding (Section 6, Step 8, prints one).

Positional encoding

The vector added to each token's embedding to mark its position, since attention by itself is order-blind. Fixed sinusoids in Vaswani et al. (2017) and in Section 6, learned rows in GPT-2; rotary variants apply the position inside the attention scores instead. Section 6, Step 1, explains how one summed vector still lets the model read the word and the place separately.

Causal mask

The rule that each token may attend only to itself and earlier tokens. Implemented by setting the upper triangle of the attention scores to $-\infty$ before softmax, so future positions get weight zero.

KV cache

The stored keys and values of all past tokens, reused at each generation step so the new token's query can attend back to the whole history. Queries are not cached, hence "KV", not "QKV".

Prefill vs. generation

Prefill processes the whole prompt at once, writing every token's K and V into the cache. **Generation** then adds one token at a time, growing the cache by one row of K and V per step.

Ablation

Switching off one component (e.g. zeroing one head's slice of W_O) and re-measuring behaviour, to test what that component *causally* does (Appendix A).

Convolution, filter, feature map (CNN terms)

A **convolution** slides a small fixed **filter** of weights over an image, computing a local weighted sum at each position; the resulting grid is a **feature map**. The contrast with an attention head, fixed local window against dynamic content-driven reach, motivates Section 1 and Appendix B.

References

Cunningham, H., Ewart, A., Riggs, L., Huben, R., & Sharkey, L. (2023). *Sparse autoencoders find highly interpretable features in language models.* arXiv. https://arxiv.org/abs/2309.08600

Elhage, N., Hume, T., Olsson, C., Schiefer, N., Henighan, T., Kravec, S., … Olah, C. (2022). *Toy models of superposition.* Transformer Circuits Thread. https://transformer-circuits.pub/2022/toy_model/index.html

Inan, H., Khosravi, K., & Socher, R. (2017). *Tying word vectors and word classifiers: A loss framework for language modeling.* International Conference on Learning Representations. https://arxiv.org/abs/1611.01462

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., … Amodei, D. (2020). *Scaling laws for neural language models.* arXiv. https://arxiv.org/abs/2001.08361

nostalgebraist. (2020, August 31). *Interpreting GPT: The logit lens.* LessWrong. https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens

Press, O., & Wolf, L. (2017). *Using the output embedding to improve language models.* Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, 157–163. https://arxiv.org/abs/1608.05859

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems, 30.* https://arxiv.org/abs/1706.03762