Introduction to Transformers and LLMs

Nick Oh
London School of Economics
n.oh1@lse.ac.uk

Steven Feng
Stanford University
syfeng@stanford.edu

DRAFT v1.2 · 17 October 2026

Abstract

This note offers a self-contained mathematical account of attention and the decoder-only transformer, the architecture at the heart of modern large language models. It follows one forward pass of a trained model, from token embeddings to next-token prediction, guided throughout by a single observation about matrix multiplication. Read down its columns, a matrix product measures each input against a family of learned directions; read across its rows, it mixes a fixed set of learned components into each output. Every step of the transformer turns out to perform one of these two operations, and the whole architecture can be understood in that small vocabulary. One dividend of this approach is that the names the architecture carries, usually taken on faith, can instead be derived; the roles of queries and keys, in particular, emerge from the computation itself. The note presumes little beyond elementary linear algebra and a first acquaintance with neural networks; appendices collect video resources for review and further study.

Contents

1 Introduction **3**
2 A Primer on Linear Algebra and Neural Networks **5**
3 Language Model in a Nutshell **12**
4 Input Representation **15**
5 Layer Normalisation **17**
6 The Q, K, V Projections **19**
7 Attention Scores **21**
8 Causal (Autoregressive) Masking **24**
9 Value Retrieval **25**
10 Multi-Head Attention **27**
11 The Feedforward Network **30**
12 The LM Head **32**
A Prerequisites: Video Resources **36**
B Further Resources: Large Language Models **39**
C Further Resources: Stanford CS25 Transformers United **40**
D The Model as Code: Karpathy's microgpt **43**

A Note from the Authors

Hey readers! We're Nick and Steven. We first crossed paths at NeurIPS 2024 in Vancouver, which is one of the biggest AI conferences in the world, where thousands of researchers cram into convention halls and posters cover every surface. A year later, we ran into each other again at NeurIPS 2025 in San Diego, spending more time together than expected (mostly eating well and wandering the city at night). We tried to keep the conversation light, but that lasted about five minutes. Every conversation drifted back to research.

One thread was on finding good explanations of how transformers *actually* work. Most explanations say Q is the "query" that searches for relevant keys. But that's the what, not the why. Resources either lean on database analogies or punt to "the model learns it". But *why does the computation create these roles in the first place*? The explanation will sound trivial once stated, but we couldn't find it stated anywhere. This is our attempt to fill that gap.

A bit about us: Nick studied Politics and Economics at LSE, did a stint as an NLP engineer at a London-based hedge fund, and now works as a researcher advised by Dr. Fernand Gobet. Steven is a Computer Science PhD student at Stanford University, co-advised by Dr. Michael Frank and Dr. Noah Goodman. He has led Stanford's CS25 Transformers course for multiple years and is currently an Anthropic Research Fellow. Both of us are interested in psychologically and cognitively inspired methods, and advanced reasoning in small foundation models.

1 Introduction

Large language models (LLMs) have become central to modern artificial intelligence. Systems like Claude (by Anthropic) and GPT (by OpenAI) generate fluent text, answer questions, write code, and perform tasks that seemed out of reach a decade ago. At the heart of these systems lies the *transformer* architecture, and at the heart of the transformer lies *attention*.

This document is a mathematical introduction to attention and the transformer. Its goal is not just to show you the equations but to make them feel intuitive, so that you see how each operation shapes the flow of information, and why the architecture behaves as it does. It is common, for instance, to be told that Q holds "queries" and K holds "keys", and simply to accept the names. We will instead watch those roles *emerge* from the structure of the computation, and by the time we reach them, calling Q a query will feel like the only sensible name.

We make one simplifying assumption throughout, which is that the model is *already trained*. We study the forward pass, meaning what the network computes when it reads a sequence, and set aside how the weights were learned. Training is where the magic of specialisation happens, as the projection matrices start out as random noise and end up performing distinguishable jobs, even if we cannot always name what each one does. But the mechanics of learning are a story for another day.

Scope

We focus on the decoder-only transformer, the architecture underlying modern autoregressive[1] language models. At a high level the model predicts the next token in three steps. Each token first becomes a vector. Those vectors then pass through a stack of attention and feedforward layers, and what comes out the far end is turned into a probability distribution over the vocabulary.

We treat each step in detail, leaning throughout on a single idea, that a matrix multiplication can be read in two complementary ways, along its rows or along its columns. We do not cover encoder–decoder architectures (as in the original Transformer for translation), encoder-only models (like BERT), training infrastructure, or the many architectural variants. These matter, but they sit beside our goal rather than on the path to it.

[1] Autoregressive means the model generates one token at a time, with each prediction depending only on the tokens before it. The prefix "auto-" (self) and "regressive" (predicting from past values) together name this sequential, self-dependent kind of generation.

Prerequisites

This document assumes some familiarity with:

- **Linear algebra**, meaning vectors, matrices, dot products, and matrix multiplication
- **Neural networks**, meaning parameterised functions, gradient descent, loss functions, and backpropagation (at a conceptual level)

If some of this is rusty, do not worry. Section 2 rebuilds the one piece we rely on most, matrix multiplication, from the ground up, and Appendix A points to video reviews, including the *Essence of Linear Algebra* series by 3Blue1Brown (about 1 hour) and a short introduction to deep learning (about 1 hour, plus an optional 70-minute MIT lecture).

Time Estimate

The core material in Sections 1 through 12 is meant to be read slowly, with pauses for reflection, in about **1 hour**. With the prerequisite videos in Appendix A, the full journey takes roughly **3 hours**, or about **4 hours** with the optional MIT lecture. Readers who want to go further can find additional videos on large language models in Appendix B (about 1.5 hours). Appendix C goes a few steps beyond that, with a full term of Stanford's CS25 seminar, nine research talks on where the architecture has been taken since. Appendix D closes with the model from Andrej Karpathy's *microgpt*, a complete GPT in plain Python, for readers who want to see the forward pass of this note as code.

Roadmap

The document proceeds as follows:

- **Section 2** reviews linear algebra and neural networks, focusing on the two complementary ways to read a matrix multiplication: down its columns, as dot products that score each input against a set of learned directions (*measurement*); and across its rows, as linear combinations that build each output by blending stored components (*mixing*).
- **Section 3** fixes notation and gives a high-level tour of the transformer's computational chain.
- **Sections 4–9** build up single-head attention one step at a time, covering embedding, layer normalisation, the Q/K/V projections, attention scores, causal masking, and retrieval as a weighted sum of values.
- **Section 10** extends this to multi-head attention, explaining why several heads help and how their outputs are mixed back together.
- **Section 11** covers the feedforward network that processes each position on its own.
- **Section 12** covers the LM head, tracing the path from the final representations to next-token predictions, including the unembedding matrix and the training loss.

2 A Primer on Linear Algebra and Neural Networks

Transformers are built from a handful of mathematical pieces, such as dot products, matrix multiplications, and a few nonlinear functions. Of these, one piece appears far more often than the rest: *matrix multiplication*. This section builds that operation from its smallest piece, the dot product, and then makes one observation that pays off for the rest of the document, namely that a single product $Y = XW$ can be read in two complementary ways. We will call them *measurement* and *mixing*. They are the same arithmetic seen from two sides, and each returns later, measurement in how attention scores tokens and mixing in how it combines them.

The dot product

For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$ of the same length, their *dot product* (or inner product) is the scalar

$$\mathbf{a} \cdot \mathbf{b} \;=\; \sum_{k=1}^{d} a_k b_k.$$

Geometrically, $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta$, where $\|\cdot\|$ is the Euclidean norm (the "length" of the vector) and θ is the angle between the two. The dot product therefore measures *alignment*: it is large and positive when the vectors point the same way ($\cos 0^\circ = 1$), zero when they are perpendicular ($\cos 90^\circ = 0$), and negative when they point in opposing directions ($\cos 180^\circ = -1$).

A list of d numbers can be written two ways: across, as a *row vector* in $\mathbb{R}^{1\times d}$, or down, as a *column vector* in $\mathbb{R}^{d\times 1}$. In this note we will loosely treat a row and a column of the same length as vectors in one and the same space, and compare them freely there.

The matrix product

For two matrices $X \in \mathbb{R}^{n\times d}$ and $W \in \mathbb{R}^{d\times m}$, their product $Y = XW \in \mathbb{R}^{n\times m}$ is given entry by entry by

$$Y_{ij} \;=\; \sum_{k=1}^{d} X_{ik} \, W_{kj}.$$

Writing $\mathbf{x}_i \in \mathbb{R}^d$ for the i-th row of X and $\mathbf{w}_j \in \mathbb{R}^d$ for the j-th column of W, this is just $Y_{ij} = \mathbf{x}_i \cdot \mathbf{w}_j$: every entry of Y is a single dot product. For one input $\mathbf{x} = (x_1, \ldots, x_d)$ and one column of weights $\mathbf{w} = (w_1, \ldots, w_d)$, the entry reads $y = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$, the weighted sum at the heart of every linear model. A matrix product simply computes one such weighted sum for every pairing of an input with a column.

Throughout this document, X collects the *inputs*, one per row, so n simply counts how many inputs we send through at once. Each input, being a list of d numbers, is a point in $\mathbb{R}^d$, one coordinate per number. Each coordinate records one *feature*, a property the input can have; the axis says which property, the coordinate says how much, so an input $\mathbf{x}_i = (0.9,\ 0.2)$ carries 0.9 of feature 1 and 0.2 of feature 2. Because its axes are the features, we call this input space $\mathbb{R}^d$ the *feature space*. W holds the *weights*, and multiplying by it moves the whole of feature space into the *output space* $\mathbb{R}^m$, often called as a *linear map*. The product $Y = XW$ collects the outputs, one per row.

For simplicity we will take $n = d = m = 2$ throughout, with the running example

$$X = \begin{bmatrix} 0.9 & 0.2 \\ 0.1 & 0.9 \end{bmatrix}, \qquad W = \begin{bmatrix} 0.5 & -0.3 \\ -0.2 & 0.8 \end{bmatrix}, \qquad Y = XW = \begin{bmatrix} 0.41 & -0.11 \\ -0.13 & 0.69 \end{bmatrix}.$$

Here $Y_{11} = \mathbf{x}_1 \cdot \mathbf{w}_1 = 0.9 \cdot 0.5 + 0.2 \cdot (-0.2) = 0.41$, and the other entries follow the same way.

$$\begin{bmatrix} 0.9 & 0.2 \\ 0.1 & 0.9 \end{bmatrix} \times \begin{bmatrix} 0.5 & -0.3 \\ -0.2 & 0.8 \end{bmatrix} = \begin{bmatrix} \boxed{0.41} & -0.11 \\ -0.13 & 0.69 \end{bmatrix}$$

$$(0.9,\ 0.2) \cdot (0.5,\ -0.2) = 0.9 \cdot 0.5 + 0.2 \cdot (-0.2) = \boxed{0.41}$$

The defining rule. The entry Y_{ij} is the dot product of the i-th row of X with the j-th column of W.

That is the whole operation, and the matrix product Y is now settled. What remains is not to compute it but to read it. The one sum $Y_{ij} = \sum_k X_{ik} W_{kj}$ can be grouped by holding the column index fixed or by holding the row index fixed, and those two groupings become the two readings we develop next.

Reading 1: measurement (by columns of W)

Fix a column index j and write $\mathbf{w}_j$ for the j-th column of W. Now gather the entries of column j of Y: that column holds the dot product of every input (the rows of X) against the single vector $\mathbf{w}_j$. Reading the product this way, scoring each input against each column, we call measurement.

$$Y_{\cdot j} = X\mathbf{w}_j \in \mathbb{R}^n, \qquad (Y_{\cdot j})_i = Y_{ij} = \mathbf{x}_i \cdot \mathbf{w}_j$$

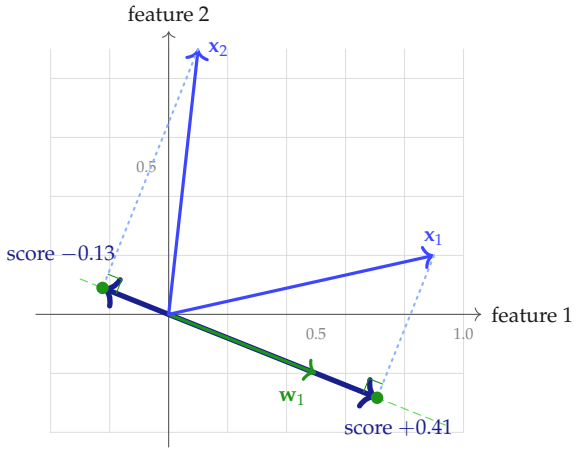
$$\underbrace{Y_{ij}}_{\text{entry of } Y} = \underbrace{\mathbf{x}_i}_{\text{row } i \text{ of } X} \cdot \underbrace{\mathbf{w}_j}_{\text{column } j \text{ of } W}$$

Measurement: an entry is a score against a column. Entry Y_{ij} is the dot product of input i — the i-th row of X — with the j-th column of W, measuring how far that input reaches along the direction $\mathbf{w}_j$. Each column poses its own question, and the matching column of Y collects every input's score.

What is $\mathbf{w}_j$ doing here? It is a *vector* in feature space, the same $\mathbb{R}^d$ the inputs live in. Picture it as an arrow from the origin, carrying both a length and a way of pointing. What matters about this arrow is almost entirely the way it points, its *direction*. To score an input against $\mathbf{w}_j$ is to ask how far the input reaches *along* that direction. The input's perpendicular shadow on the line through $\mathbf{w}_j$ — its *projection* — makes that reach visible, and the dot product reports it as a single number, positive when the shadow runs forward, negative when it falls backward. The arrow's length, by contrast, plays no role in the comparison. It sets only the units the scores are reported in: stretching $\mathbf{w}_j$ multiplies every input's score by the same factor, so the ranking of the inputs, and the sign of each, are left exactly as they were. So a column of W

fixes a direction in feature space, and the matching column of Y records how far each input extends along that direction.

To make this concrete, let's look at the figure below. The two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ sit in feature space, and $\mathbf{w}_1 = (0.5, -0.2)$ fixes a direction, the dashed green line through it. Dropping each input perpendicularly onto that line gives its projection: $\mathbf{x}_1$ scores $\mathbf{x}_1 \cdot \mathbf{w}_1 = 0.9 \cdot 0.5 + 0.2 \cdot (-0.2) = 0.41$, while $\mathbf{x}_2$ scores -0.13. The length and sign of each shadow are just *alignment* made visible: $\mathbf{x}_1$ makes a small angle with $\mathbf{w}_1$ and so points much the same way, its shadow running forward; $\mathbf{x}_2$ makes an angle past a right angle and so points partly against $\mathbf{w}_1$, its shadow falling backward and its score turning negative. This is exactly $\mathbf{x} \cdot \mathbf{w}_1 = \|\mathbf{x}\| \, \|\mathbf{w}_1\| \cos\theta$ at work. The narrower the angle, the larger $\cos\theta$, and the longer the forward projection. Those two numbers are precisely the first column of Y.



Measurement as projection. Each input casts a perpendicular shadow onto the line through $\mathbf{w}_1$, and the thick arrow on that line is the projection itself. The labels give each input's *score* $\mathbf{x} \cdot \mathbf{w}_1$, proportional to the shadow's signed length.

Reading 2: mixing (by rows of W)

Fix instead a row index i, and write $\mathbf{r}_k \in \mathbb{R}^m$ for the k-th row of W. The i-th row of Y is then a *linear combination* of the rows of W whose coefficients are the entries of the i-th row of X. Reading the product this way, building each output row by combining the rows of W, we call mixing.

$$Y_{i\cdot} = \sum_{k=1}^{d} X_{ik}\, \mathbf{r}_k \in \mathbb{R}^m$$

$$\underbrace{\mathbf{y}_i}_{\text{row } i \text{ of } Y} = X_{i1}\,\mathbf{r}_1 + X_{i2}\,\mathbf{r}_2 + \cdots + X_{id}\,\mathbf{r}_d \qquad \text{(rows of } W\text{)}$$

Mixing: an output row is a blend of W's rows. Row i of Y is a weighted combination of the rows of W, $\mathbf{r}_1, \ldots, \mathbf{r}_d$, with weights $X_{i1}, \ldots, X_{id}$ read off row i of X. The rows are shared by every input; only the weights change from one input to the next, so each output row is its own blend of the same d rows.

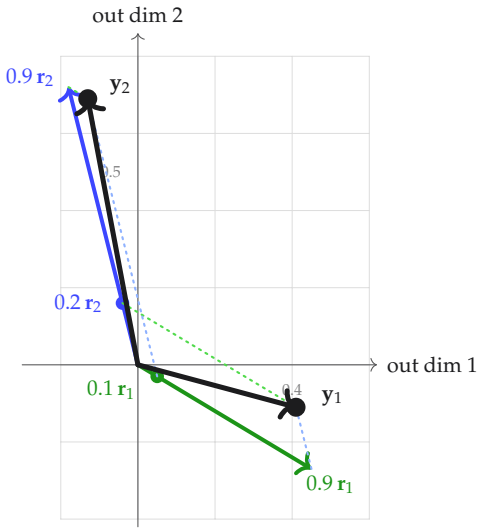
The rows of W are a fixed handful of vectors in the *output* space $\mathbb{R}^m$, shared by every input. To take a linear combination of them is to scale each by some amount and add the results, and

the output for input i is one such combination, $X_{i1}\,\mathbf{r}_1 + X_{i2}\,\mathbf{r}_2 + \cdots + X_{id}\,\mathbf{r}_d$. As the coefficients range over all values, *positive and negative*, such combinations sweep out the *span* of $\mathbf{r}_1, \ldots, \mathbf{r}_d$, the entire region of points they can reach. As long as the rows point in genuinely different directions, as our two do, that region is the whole of $\mathbb{R}^m$ whenever there are at least m of them; in general it is the subspace they span. Every output the map can ever produce is some blend of the same d rows, with the input's entries setting how much of each to take.

On the running example, $\mathbf{r}_1 = (0.5, -0.3)$ and $\mathbf{r}_2 = (-0.2, 0.8)$, and

$$Y_{1\cdot} = 0.9\,\mathbf{r}_1 + 0.2\,\mathbf{r}_2 = (0.41, -0.11), \qquad Y_{2\cdot} = 0.1\,\mathbf{r}_1 + 0.9\,\mathbf{r}_2 = (-0.13, 0.69).$$

The two rows $(\mathbf{r}_1, \mathbf{r}_2)$ are common to both outputs; only the amounts, read off each input's row, change.



Mixing: a blend of W's two rows. Each output is a linear combination of the two rows of W, each row scaled by the input's entries and added. Because the two rows span the output plane, every reachable output is some such blend.

The two readings coincide

Why two readings of one product, and why are they drawn in different spaces? Because the same matrix W meets the two readings through different slices. A *column* of W has d entries, one for each input feature, so it is a vector in the input space $\mathbb{R}^d$, sitting among the inputs; that is the space the measurement figure is drawn in, and a column is something an input can be measured *against*. A *row* of W has m entries, one for each output dimension, so it is a vector in the output space $\mathbb{R}^m$, sitting among the outputs; that is the space the mixing figure is drawn in, and a row is something an output can be built *from*.

Whichever reading we take, the arithmetic beneath it is the same. Pick any entry Y_{ij}. Measurement reaches it by pairing input i with column j; mixing reaches it as the j-th coordinate of the blend that forms output i. Both come out to the one sum $Y_{ij} = \sum_k X_{ik} W_{kj}$. There is really a single matrix product Y: read it column by column and it tells the measurement story, read it row by row and it tells the mixing story.

We carry both forward, under the names measurement and mixing, for the rest of the document.

Neural Networks in Essence

So far W has simply been given to us. In a neural network its entries are *learned*. This is what the name *machine learning* refers to, a machine that learns its own weights. The learning follows one loop. The network makes a prediction, compares it with the truth, and measures the gap as an error. An algorithm called *backpropagation* then traces the error backwards through the network, working out how much each weight contributed and in which direction a small correction would help. Over millions of examples the corrections accumulate, and the weights settle into values that serve the task. (The full mechanics are beyond our scope; the videos of Appendix A treat them properly.)

Machine learning (ML) papers frequently say a weight matrix "projects" inputs into a learned space. The original transformer paper by Vaswani et al. (2017), for instance, speaks of mapping the queries, keys, and values with "different, learned linear projections", and later writing often describes W_Q, W_K, W_V as projection matrices carrying embeddings into an attention space. In ML, "projection" usually means *some* linear transformation, that is, multiplying by a weight matrix to produce a new representation.

Consider $Y = XW$ where W has shape $d \times m$. We say W "projects" d-dimensional inputs into an m-dimensional learned space, learned because the entries of W are fixed by training. The columns of W become directions that detect meaningful patterns, and its rows become templates that blend together. Whether m is smaller than d, larger, or equal, the key idea is that W re-expresses inputs in terms of patterns this nudging has discovered to be useful for the task.

This *is* the building block of deep learning. A neural network chains these projections with nonlinearities, so that each layer applies $\mathbf{y} = f(XW + \mathbf{b})$, projecting into a new learned space via XW and then bending[2] it with the nonlinearity f. The nonlinearity is what makes depth meaningful. Without it, stacking layers gains nothing, since $(XW_1)W_2 = X(W_1W_2)$ collapses to a single linear transformation no matter how many layers we pile up. The nonlinearity is what preserves the hierarchy that depth creates.

The power of deep networks comes from iterating this loop, projecting into a learned space, bending through a nonlinearity, and projecting again. Layer 1 might project into a space where primitive features emerge, layer 2 into a space where combinations of those primitives become visible, layer 3 into a space where higher abstractions make sense. Each cycle refines what the network "sees" in the data, building from raw measurements towards abstract understanding.

A Note on Parallelisation To *parallelise* a computation is to perform many of its parts at the same time rather than one after another. Matrix multiplication is a natural candidate, because the entries of Y do not depend on one another. Computing any single entry y_{ij} needs only row i of X and column j of W, so all entries can be worked out simultaneously, in any order, on separate hardware. In the transformer setting, though, parallelisation carries a second and more consequential meaning, the removal of *sequential dependencies* across the sequence. Before

[2]Bending means reshaping the space in ways a matrix cannot. On its own, matrix multiplication can only rotate, stretch, and shear the space (i.e., straight lines stay straight). A *nonlinearity* is a simple, fixed rule applied to each number on its own — for instance ReLU, which replaces every negative value with zero, or the sigmoid, which squashes every value into the range from 0 to 1. Rules like these bend the space, adding the curves, folds, and corners a matrix never could. This is essential because real-world patterns are rarely linearly separable. Bending allows the network to curve decision boundaries around complex patterns, transforming messy, entangled data into spaces where simple linear separators work.

transformers, the dominant tool for sequences was the recurrent neural network (RNN), which read a sequence strictly in order, token 1, then token 2, then token 3, because each hidden state was built from the one before it. That was an unavoidable bottleneck, no matter how parallelisable the underlying matrix multiplications happened to be. Transformers remove it by processing all tokens in a sequence at once within each layer, with no position waiting on any other. This single architectural change was a major factor in making training on far larger datasets and models practical.

Row Convention Throughout this document, we represent samples (or token embeddings) as *row vectors*, not column vectors. Linear algebra textbooks typically use column vectors, but ML conventionally uses row vectors, with two practical reasons. The first is memory layout. In row-major storage (used by C, Python/NumPy, and PyTorch), the d entries of a row sit contiguously in memory, so a token stored as a row of an (n, d) matrix occupies one contiguous block. The second is readability. With row vectors the forward pass reads left to right as `x @ W1 @ W2 @ W3`, matching both reading order and the conceptual flow through the layers. Neither point makes the convention necessary. The identical computation can be written with column vectors, and the difference is one of convention and convenience rather than of what the network can express.

From Features to Word Vectors

Before we close this section, one matrix deserves a final look. Of the three in $Y = XW$, we know by now where two come from. Y is computed, and W is learned. Only X has been handed to us from outside, as numbers that someone measured and wrote down. This note, however, is about language models, whose inputs are words. Each word must somehow become a vector, and in an LLM that vector is not designed but *learned*, exactly as the weights are. The inputs, then, come out of training too, as numbers that were never measurements of anything. If nothing was measured, what do the entries record? Before facing that question for learned vectors, let's recall how meaning entered the numbers when we chose them ourselves.

A *feature*, we said, is a property an input can have (Elhage et al., 2022), a fact about the world, with its coordinate recording how much. When a dataset records each day's weather as (temperature, humidity), *we* decided what the axes mean, and reading a feature off an input is just reading a coordinate. In settings like these the input space is transparent, because every axis was chosen by hand.

However, text breaks this arrangement. What are the features of the word *meaning*? Its abstractness, its life as noun and as verb, the company it keeps, the shade it takes on in *well-meaning* or in *the meaning of life*. The list is real but unbounded, and nobody can sit down and assign each entry an axis. Fortunately, the field has discovered that they can be *learned*. A method called word2vec (Mikolov et al., 2013a) trained small networks on enormous amounts of text with a simple objective, predicting each word from the words around it, and in doing so gave every word in the vocabulary its own vector. No axis was ever told what to mean. Yet when training ended, meaning had appeared in the geometry (Mikolov et al., 2013b):

$$\mathrm{vec}(\textit{king}) - \mathrm{vec}(\textit{man}) + \mathrm{vec}(\textit{woman}) \;\approx\; \mathrm{vec}(\textit{queen}).$$

Subtracting $\mathrm{vec}(\textit{man})$ removes the part of *king* that points along one particular direction, and adding $\mathrm{vec}(\textit{woman})$ writes in another. That the result lands near *queen* tells us that these

directions, in some sense, *represent* features. Gender and royalty are properties of words, facts about the world, and this space has encoded each one along a direction of its own.

The *linear representation hypothesis* holds that many meaningful features, concepts among them, are represented inside a neural network as approximately linear directions in its learned space. Take two words that are alike except in one feature, such as *man* and *woman*, and subtract their vectors. What is left is a shift, the change that turns one vector into the other. The hypothesis says that this shift is roughly the same for every such pair, from *king* to *queen*, from *uncle* to *aunt*, and that shared shift is the gender *direction*. The arithmetic above did nothing more than take *king* and apply the *man*-to-*woman* shift.

Now let's look at what a shift is made of. Vectors subtract entry by entry, so the gender shift is a list of d changes, one per entry. What matters is how those changes are spread across the entries, and the spreading runs both ways. One feature reaches many entries, because no axis was ever told what to mean. One entry is reached by many features, and it keeps only their total. So a feature cannot be read from one entry, and an entry cannot be traced back to a feature; what carries the feature is the whole list of d changes at once, which is the shift, the direction. Meaning, then, lies not in the entries but in the directions that run across them.

With measurement, mixing, and projection in hand, and with meaning now attached to the numbers, we have the whole vocabulary we need. Every operation in a transformer, however imposing its notation, turns out to be one of these moves repeated and recombined. Before examining those operations one at a time, the next section steps back to show the entire machine at once: how tokens become embeddings, how layers stack, how attention and feedforward networks interleave, and where the predictions finally emerge. With that map in view, each later section can zoom in on a single component.

3 Language Model in a Nutshell

In a nutshell, an LLM is a stack of transformer blocks. And each block is simply a chain of matrix operations. Before diving into the mathematics, let's see what this chain looks like.

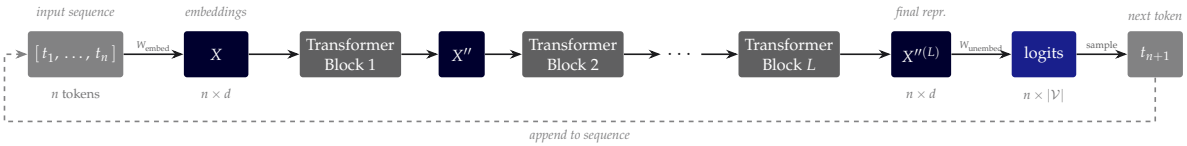


Figure 1: LLM Pipeline (High-Level)

LLMs take a sequence of tokens as input, convert each token into an embedding vector, and pass these through a stack of transformer blocks which build progressively richer, contextualised representations of meaning (Figure 1). The final representation of the last token then produces a score for every token in the vocabulary, the *logits*, which a final step turns into a probability distribution over possible next tokens. The model samples from this distribution, appends the chosen token to the input, and repeats; predicting the next token, one at a time.

Each transformer block is a multi-layer network consisting of attention, feedforward, normalisation layers, and residual connections. In Figure 2a, the residual pathway from X to X'' is drawn as a straight central line, the *residual stream*, with attention and the feedforward network branching off it, doing their work, and adding their results back in. Seen this way, each sublayer *reads* from the stream and *writes* its result back.

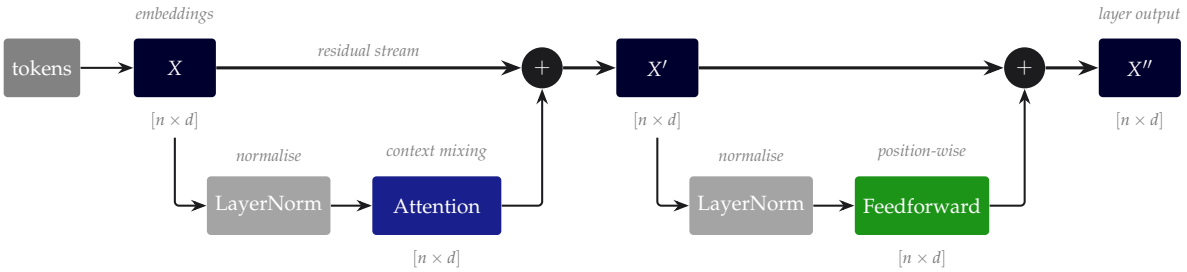
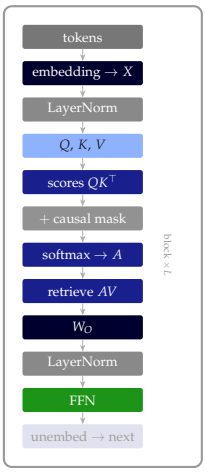


Figure 2a: Transformer Block (High-Level)

The **attention layer** builds contextual representations by allowing each token to "attend" to and integrate information from surrounding tokens. The **feedforward layer** is a simple two-layer fully connected neural network that processes the integrated information for each embedding independently, often called position-wise because it operates on each position separately. **Normalisation layers** support these sublayers, enabling the network to be trained stably at depth.

Figure 2b expands one transformer block into its individual operations, tracing how a token's representation flows through each step. It is the most detailed diagram in this document, and it is meant to be returned to. On a first read, do not try to parse every symbol. Instead, follow the flow downward and track one thing, the shape of the data. A representation enters at the top and travels down the residual stream, leaving at the bottom in the same shape ($n \times d$) it arrived, only richer; attention and the feedforward network branch off the stream, compute, and merge what they find back into the stream. The dimensions change only briefly inside the sublayers, both within attention (down to d_k and d_v) and within the feedforward network (up to d_{ff} and back down), before being restored to $[n \times d]$ each time. Every symbol in this figure is unpacked, one operation at a time, in the sections that follow.



This mini-map returns at each section with the active step lit. Sections 4 to 12 will cover these operations step by step.

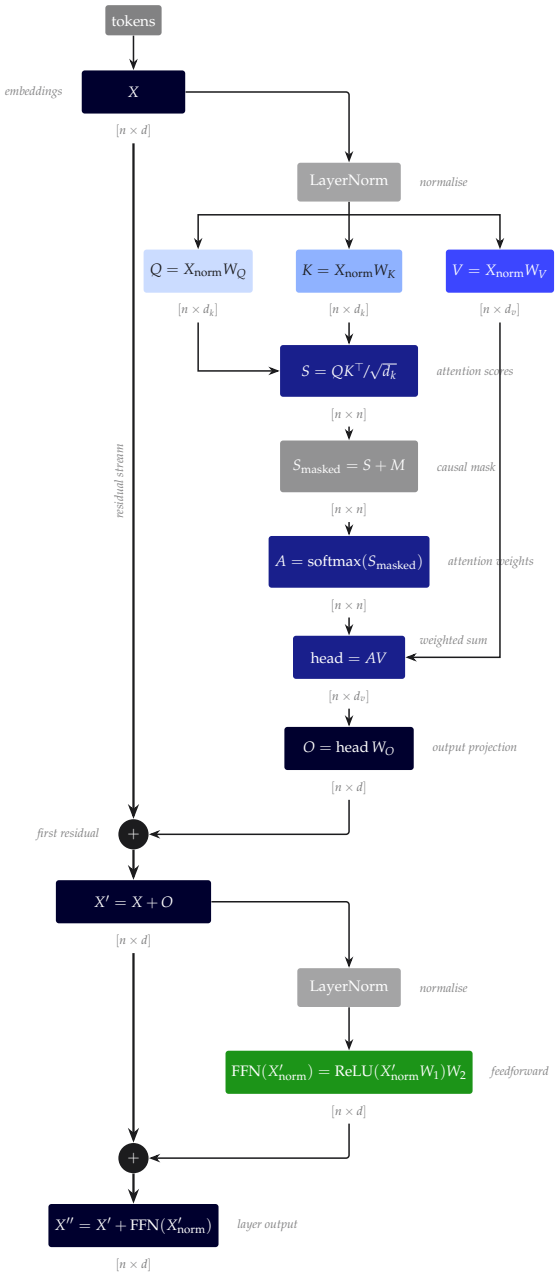


Figure 2b: Single-Head Attention Transformer

The notation in Table 1 is collected here for reference. There is no need to absorb it now. It is the page to flip back to whenever a symbol in a later section is unfamiliar, and the three short groupings beneath it explain how the symbols relate.

Symbol	Meaning
Dimensions	
n	Sequence length (number of tokens)
d	Model dimension (embedding size); sometimes written d_{model}
d_k	Query/key dimension per head
d_v	Value dimension per head (typically $d_v = d_k$)
d_{ff}	Feedforward hidden dimension (typically $4d$)
h	Number of attention heads
L	Number of transformer layers
$\lvert\mathcal{V}\rvert$	Vocabulary size (number of unique tokens)
Matrices	
W_{embed}	Embedding matrix $[\lvert\mathcal{V}\rvert \times d]$; maps token IDs to vectors
W_{unembed}	Unembedding matrix $[d \times \lvert\mathcal{V}\rvert]$; often $W_{\text{embed}}^\top$, though many large models untie the two
X	Input matrix $[n \times d]$; rows are token embeddings
Q, K, V	Query, key, value matrices $[n \times d_k]$ or $[n \times d_v]$
S	Attention scores $[n \times n]$; entry (i, j) is $q_i \cdot k_j / \sqrt{d_k}$
A	Attention weights $[n \times n]$; softmax of masked S, rows sum to 1
W_Q, W_K, W_V	Projection matrices $[d \times d_k]$ or $[d \times d_v]$
W_O	Output projection $[hd_v \times d]$ (with $hd_v = d$, this is $[d \times d]$)
W_1, W_2	Feedforward weights $[d \times d_{\text{ff}}]$ and $[d_{\text{ff}} \times d]$
Intermediate states	
X'	Intermediate representation after attention residual
X''	Output representation after feedforward residual

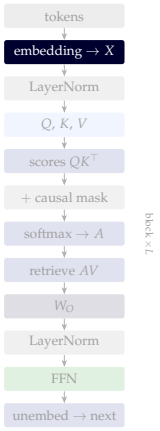
Table 1: Notation Reference

The first group describes the dimensions that define a transformer's size. A sequence of n tokens is embedded into vectors of dimension d. Each attention head operates on a smaller dimension d_k (for queries and keys) and d_v (for values), with h heads running in parallel (typically $d_k = d_v = d/h$). The model stacks L layers, and the final output spans a vocabulary of $|\mathcal{V}|$ tokens.

The second group describes the matrices that flow through the computation. The embedding matrix W_{embed} maps discrete token IDs to continuous vectors, producing the input X with one embedding per row. The projections Q, K, V are derived from X (after a layer norm) via learned weight matrices W_Q, W_K, W_V. The score matrix S and attention matrix A are both $n \times n$, capturing pairwise relationships between all token positions. The output projection W_O maps head outputs back to model dimension, and W_1, W_2 are the feedforward weights. After the final transformer block L, the unembedding matrix W_{unembed} (the transpose of W_{embed}, though the two are untied in many large models) maps the final representations back to vocabulary logits.

The third group tracks intermediate states through a layer: X' after the attention sub-layer, X'' after the feedforward sub-layer. With the architecture and notations mapped out, we now trace the computation step by step, starting with how tokens become vectors.

4 Input Representation



The input to a transformer is a sequence of text. A tokeniser splits that text into tokens[3] and assigns each one an integer ID drawn from a fixed vocabulary of $|\mathcal{V}|$ entries. For example,

"meaning emerges from context" $\longrightarrow$ [1847] [9301] [482] [2174].

On their own these IDs carry no meaning. They are addresses, not descriptions, and the number 1847 tells us nothing about the word it stands for. The mapping from token strings to these integers is stored in a vocabulary file alongside the model, commonly `vocab.json` or `tokenizer.json`.

Turning those addresses into meaning is the job of the embedding matrix $W_{\text{embed}} \in \mathbb{R}^{|\mathcal{V}| \times d}$, which holds a learned d-dimensional vector for every token in the vocabulary, one per row. The integer ID now acts as a row index, so fetching token i's embedding is just reading off row i, turning a meaningless address into a rich vector.

$$e_i = W_{\text{embed}}[\text{token}_i] \in \mathbb{R}^{1 \times d}$$

$$\begin{bmatrix} e_0 \\ e_i \\ \vdots \\ e_{|\mathcal{V}|-1} \end{bmatrix}_{|\mathcal{V}| \times d}$$

W_{embed}. One row per vocabulary token; the lookup for token i returns row i. Width d = model dimension.

That vector e_i is everything the model knows about the token before it has seen any context, and that qualifier matters in two ways. First, the embedding is the same every time the token appears, no matter what surrounds it. The word *bank* receives one fixed vector whether the sentence is about a river or a savings account, so e_i can only capture a kind of blurred blend of its senses a token ever has. Sharpening that blur into a single intended sense is the job of attention, later, and is the reason the model needs context at all. Second, the embedding is blind to position. A token maps to the same vector whether it is the first word or the hundredth, because the lookup depends only on which token it is, never on where it sits. Before the representation can be useful, that missing piece, position, has to be supplied.

Positional Encoding

Word order carries meaning, and a model that ignored it would miss much of what a sentence says. Reordering the words does change the input, since the rows end up in a different order,

[3] A token is typically a word, subword, or character. Most modern LLMs use Byte Pair Encoding (BPE) or a close relative, often via tokeniser libraries such as SentencePiece. BPE iteratively merges frequent character sequences into single tokens, balancing vocabulary size against the ability to represent rare words as combinations of common subwords.

but it changes nothing else, as each word fetches the same vector wherever it sits. Two phrases with the same tokens in different orders therefore arrive as the same set of embeddings, merely rearranged, and the sole trace of order is the order of the rows. That trace goes unused, because attention inherently treats the rows as an unordered set, reading each token from its content and ignoring where it sits. Position must therefore be written into the vectors themselves, so the order-blind layers ahead can see it.

Positional encoding repairs this by giving each position its own signature. Position i is assigned a vector $p_i \in \mathbb{R}^{1\times d}$ that depends only on the slot, not on the word filling it, and this signature is combined with the token's embedding. How the signature is produced varies by model. It may be learned, as a separate table of vectors indexed by position, or computed from a fixed formula, as in the sinusoidal[4] pattern of the original Transformer (Vaswani et al., 2017). In these additive schemes the two pieces of information are simply summed,

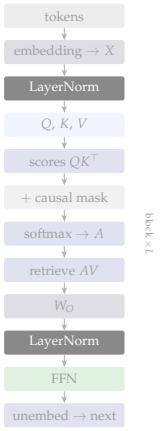
$$x_i = e_i + p_i \in \mathbb{R}^{1\times d},$$

so that a single vector now records both *what* the token is, through e_i, and *where* it sits, through p_i.

Stacking the n resulting vectors as rows gives the input matrix $X \in \mathbb{R}^{n\times d}$, with X now carrying both meaning and order. This is the matrix that enters the first transformer block.

[4]As of 2026, many modern models take a different route. Rotary position embeddings (RoPE), for instance, inject position not by adding a vector here but by *rotating* each token's query and key inside the attention computation, turning them by an angle that grows with the token's position (e.g., the tenth token's query is rotated twice as far as the fifth's, so tokens near the end of a long sequence arrive rotated furthest). The effect is that attention ends up encoding *relative* distance rather than absolute slots. The principle is the same as in the additive scheme used here, that position must enter somewhere, even when the mechanism differs.

5 Layer Normalisation



Every transformer block applies layer normalisation, usually shortened to *layer norm*, at two points: once before attention and once before the feedforward network[5]. Its job is to keep the numbers flowing through the network at a steady, predictable scale. A token's vector is just a list of numbers, and every layer changes those numbers a little as the vector passes through. Across many layers these changes accumulate, and there is nothing in the design that keeps the numbers within any fixed range. They can grow much larger or shrink much smaller from one layer to the next, and that instability is a problem above all for *training*.

During training, a weight matrix is learned gradually, through many small corrections that nudge it towards detecting its feature a little better each time. The size of each correction tracks the size of the inputs it was computed from. If that scale lurches from layer to layer, the corrections lurch with it, too large at one moment and too small at the next, and learning becomes slow, unsteady, and can stall altogether. Layer norm removes the problem at its source by resetting each vector to a *standard scale* before it reaches the next computation, so that the matrices that follow (W_Q, W_K, W_V, and the feedforward weights W_1, W_2) always meet inputs in a familiar range.

Normalisation or Standardisation?

A quick word on terminology, because the two terms are easy to confuse and layer norm sits right on the seam between them. In statistics, *standardisation* has a precise meaning: take a set of numbers, shift them so their average is zero, then divide by their standard deviation so that the spread becomes exactly one (e.g., z-score). *Normalisation* is the looser, more general word for putting numbers onto a common, comparable scale by whatever means; min–max scaling, for instance, maps the smallest value to 0 and the largest to 1 via $(x - \min)/(\max - \min)$. The operation layer norm actually performs is, strictly speaking, standardisation. It computes a z-score for each token, across that token's features. But the technique is named for the broader goal it serves, which is to keep every token's vector on a predictable, common scale.

The Computation

Layer norm works on one token at a time, using only that token's own numbers. Given token i's embedding $\mathbf{x}_i \in \mathbb{R}^{1\times d}$, it first finds the mean μ_i and standard deviation σ_i of the d values inside that single vector, then standardises them,

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}.$$

The small constant ϵ, around 10^{-5}, sits inside the square root and guards against dividing by zero in the rare case that a token's values are all identical, which would otherwise make

[5]Normalising *before* each sub-layer is called Pre-Norm. The original Transformer (Vaswani et al., 2017) normalised *after* each sub-layer instead (Post-Norm), but most modern models use Pre-Norm because it trains more stably at depth.

σ_i^2 vanish. Note that this is done *independently* for each token, so one token's rescaling never depends on another's.

Standardising a vector might sound destructive. If we slide and squeeze its numbers, do we not also damage the meaning they carry? Section 2 gives us the vocabulary for a precise answer. Meaning, on the picture built there, lives in the *direction* a vector points, not in its overall size, and standardisation barely touches direction. Dividing by the spread changes the vector's length alone. Subtracting the mean erases exactly one direction, the uniform one, along which every entry rises or falls by the same amount.

The Learned Adjustment

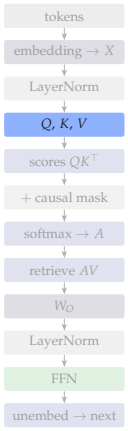
Standardisation alone is a blunt reset. It forces every token to average exactly zero and spread exactly one, which is stable but rigid, since some axes may deserve more weight, or a different centre, than this one-size-fits-all standard permits. So layer norm follows the reset with a small learned adjustment,

$$\mathbf{x}_{i,\text{norm}} = \gamma \odot \hat{\mathbf{x}}_i + \beta,$$

where γ rescales the standardised vector and β shifts it. The parameters $\gamma, \beta \in \mathbb{R}^d$ act one axis at a time, with γ_j setting the scale of axis j and β_j its centre, and they are shared across all tokens and learned alongside the rest of the network's weights.

What layer norm allows is thus a common footing. Standing in front of every major computation in the block, it ensures the attention and the feedforward network always read a vector on the same steady, learned scale. With the input now normalised, we can turn to the heart of the block, where attention projects each token into queries, keys, and values.

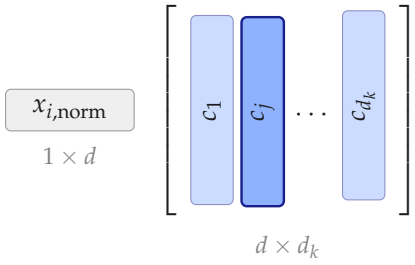
6 The Q, K, V Projections



The heart of attention is three linear transformations applied to the same input. Each takes a token's representation and re-expresses it for a different purpose, and all three are exactly the kind of operation we studied in Section 2, a matrix multiplication, read here through the measurement[6] lens. Recall that each token's representation $x_i \in \mathbb{R}^{1\times d}$ already encodes both what the token is and where it appears. The projection step asks three different sets of questions of that representation.

Let's take Q, the first of the three. The product $Q = X_{\text{norm}}W_Q$ multiplies each token's normalised embedding $x_{i,\text{norm}}$ by the matrix $W_Q \in \mathbb{R}^{d\times d_k}$, giving that token its query $q_i = x_{i,\text{norm}}W_Q$. Writing W_Q as a collection of d_k columns, each a d-dimensional vector, this projection is nothing more than d_k dot products computed at once, one against each column.

$$q_i = \begin{bmatrix} x_{i,\text{norm}}\cdot c_1 & x_{i,\text{norm}}\cdot c_2 & \cdots & x_{i,\text{norm}}\cdot c_{d_k}\end{bmatrix}_{1\times d_k}$$



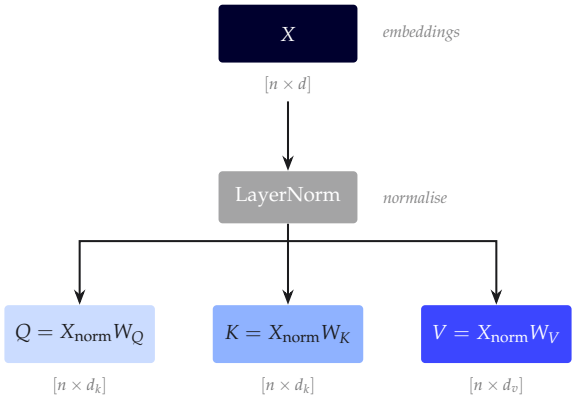
W_Q (and W_K, W_V alike). Each column is one direction the token $x_{i,\text{norm}}$ (left) is scored against; $X_{\text{norm}}W_Q$ projects every token onto all d_k of them at once. Recall that the machine learning community calls W_Q a *projection*, in the loose sense we met in Section 2, meaning any learned linear map that carries vectors from one space into another.

Dotting x_i against column c_j asks how strongly the token exhibits the particular blend of features that c_j weighs, and returns a single score. The d_k columns ask d_k such questions in parallel, so the output q_i is a compact profile of the token, reading off how it scores on question 1, question 2, and so on up to question d_k.

These "questions" themselves are also learned. At the start of training the columns are random, asking arbitrary questions whose answers mean nothing, and training gradually reshapes them into questions worth asking, even though we usually cannot put a human name to what any single one detects. One might come to track a token's grammatical role, another its broad topic, but these are guesses[7] at an organisation the network arrives at on its own.

[6] Whenever attention or the feedforward network *reads* from the residual stream, multiplying it by a weight matrix, the natural reading is measurement, the stream's content scored against learned directions. Whenever a sublayer *writes* its result back to the stream, the natural reading is mixing, a blend of learned vectors added into the stream. Reading measures; writing mixes.

[7] A research field, *mechanistic interpretability*, is devoted to this problem, working to replace guesses about what a network's parts detect. A foundational work is the mathematical framework of transformer circuits (Elhage et al., 2021).



The three projections. The normalised representation is sent through three separate weight matrices, producing the query, key, and value profiles of every token.

The same reading applies to the other two matrices. Each of W_K and W_V is likewise a stack of columns, and each column asks its own learned question of the same embedding $x_{i,\text{norm}}$. The essential picture is that one token representation is put through three different sets of questions at once, producing three different profiles of the same token, q_i, k_i, and v_i. All three descend from the very same row of X, the one vector carrying the token's identity and position. They differ only in which questions are asked.

A Note on the Names

Nothing so far has explained the names. We have three sets of questions and three profiles of the same token; calling one a query, another a key, and the third a value has not yet been earned.

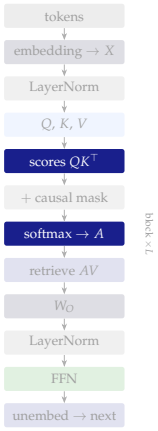
A common and useful picture compares attention to a lookup in a dictionary. A query is matched against a set of keys, and the value filed under the best-matching key is returned, except that here the match is "soft", meaning that every key's value contributes a little rather than one winning outright. It is a good first intuition, and we will recover it.

But we will not start from it, because starting there invites us to simply accept that one matrix *is* the queries and another *is* the keys, as though the names were assigned by decree. They are not. The roles are going to *emerge* from the computation, and the names will turn out to fit for a precise reason, one that has nothing to do with the dictionary metaphor and everything to do with the arithmetic of the next two sections. So for now, treat the following as provisional labels rather than established facts,

- $q_i = x_{i,\text{norm}} W_Q$, loosely, what token i is looking for,
- $k_i = x_{i,\text{norm}} W_K$, loosely, what token i offers to others,
- $v_i = x_{i,\text{norm}} W_V$, loosely, the content token i contributes when attended to.

At this stage these are just three different linear maps of the same input, and nothing we have written so far forces them into these roles. Why a token's query should seek, and why its key should be the thing sought, all of this comes from where each matrix sits in the computation that follows.

7 Attention Scores



The previous section left us with three profiles of each token (q_i, k_i, and v_i) and an open question. We attached provisional names to them, but nothing yet justified those names. The three matrices are just three different linear maps. This section is where that changes. By the end of it the labels *query* and *key* will have earned themselves, and we will see what forces them apart.

The full attention computation is

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

and we will take it in two stages: the scores $QK^\top$ in this section, and the retrieval by V later. Notice first that the comparison itself does not care which token plays which role. Measuring how well a query matches a key is a dot product, and a dot product treats its two arguments alike. So nothing in the act of measuring one token against another favours the query side or the key side; whatever ends up distinguishing the two roles must come from elsewhere in the computation.

Attention Score Matrix

Consider the product $QK^\top$. With $Q \in \mathbb{R}^{n \times d_k}$ and $K \in \mathbb{R}^{n \times d_k}$, transposing K gives a $d_k \times n$ matrix, so the product is $n \times n$.

$$\begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1d_k} \\ \vdots & \vdots & \ddots & \vdots \\ q_{i1} & q_{i2} & \cdots & q_{id_k} \\ q_{n1} & q_{n2} & \cdots & q_{nd_k} \end{bmatrix} \times \begin{bmatrix} k_{11} & \cdots & k_{j1} & \cdots & k_{n1} \\ k_{12} & \cdots & k_{j2} & \cdots & k_{n2} \\ \vdots & & \vdots & & \vdots \\ k_{1d_k} & \cdots & k_{jd_k} & \cdots & k_{nd_k} \end{bmatrix} = \begin{bmatrix} s_{11} & \cdots & s_{1j} & \cdots \\ \vdots & & \vdots & \\ s_{i1} & \cdots & \boxed{s_{ij}} & \cdots \\ \vdots & & \vdots & \end{bmatrix}$$

row i = query q_i — col j = key k_j

The raw scores $QK^\top$. The boxed entry $s_{ij} = q_i \cdot k_j$ is query i met with key j. ($K^\top$ shown directly, so its columns are the keys; dividing by $\sqrt{d_k}$ gives the scaled score matrix S, keeping scores at a steady size so the softmax ahead stays soft.)

Entry (i, j) is the dot product $q_i \cdot k_j$. Read in the measurement language of Section 2, it scores how strongly position i's query aligns with position j's key — a single number for how well what token i is looking for matches what token j offers. A large positive score means the match is strong; a score near zero means they are unrelated; a negative score means they actively pull apart.

Softmax Turns Scores into Attention

Nothing so far has distinguished the query role from the key role. The rule filling each entry (i.e., dot a query against a key) draws no line between the two. What finally separates the roles is the next step, the softmax that turns these raw scores into attention.

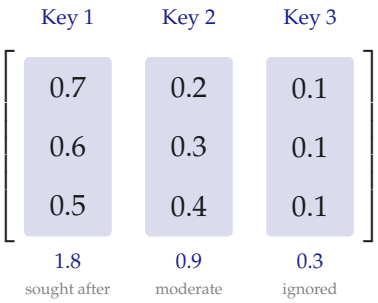
Softmax is applied *one row at a time* to the scaled scores $S = QK^\top/\sqrt{d_k}$, turning each row into non-negative weights that sum to one,

$$A_{ij} = \frac{\exp(S_{ij})}{\sum_{j'=1}^{n} \exp(S_{ij'})}.$$

The decisive point is that softmax acts along rows, not columns, and this treats the two directions of the matrix completely differently. Each row is forced into a probability distribution: a query must spend exactly one unit of attention, no more and no less, divided among the keys according to how well each matched. Whereas columns are bound by nothing. A key's column can sum to almost anything, depending on how many queries happened to want it.



Rows are probability distributions. Each query spends exactly one unit of attention, divided among the keys — so every row of A sums to one.



Columns are under no constraint. Column sums (how much total attention each key drew) are free: Key 1 pulls 1.8, Key 3 only 0.3, though every query divided its own unit fairly.

This is the asymmetry the section was driving towards, and it gives queries and keys genuinely different natures. Read across a row, a query holds a fixed budget of one unit of attention and is guaranteed to spend all of it, so however the scores fall, every query comes away having gathered something. Read down a column, a key has no such guarantee. It is merely available to be matched, and whether it is attended to heavily, lightly, or not at all is decided entirely by how well it answers the needs of the queries around it.

Thus, *a query behaves like a seeker that will always retrieve; a key behaves like a candidate competing to be retrieved*.

What This Means for Learning

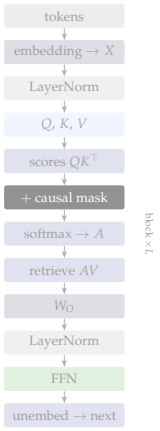
We can now say, at least in spirit, why W_Q and W_K end up doing different jobs even though they have the same shape and start from the same random initialisation. The two matrices sit in different positions in the computation, so they are shaped by different feedback. A query is always judged on what it manages to gather, so W_Q is pushed to produce queries that align

with the keys actually worth gathering. A key earns influence only when queries choose it, so W_K is pushed to make tokens findable by the queries that should want them. Same arithmetic, opposite sides of it, and the pressure on each reflects the side it sits on. Nothing here was designed by hand. The structure creates the pressure, and training[8] does the rest.

The third matrix, W_V, sits outside all of this. It takes no part in the competition over who attends to whom — it lives outside the softmax entirely — and it learns something different: what a token should contribute once it has been attended to. We will see it work in Section 9. First, though, the mechanism as described lets every query reach every key, and for a model that generates text left to right that is too much freedom. The next section adds the one constraint that makes autoregressive generation possible, restricting each position to attend only to itself and the positions before it.

[8]During training, the model improves by nudging W_Q and W_K a little after each mistake (these nudges are what training calls *gradients*). Picture the attention weights as a grid, one row per query and one column per key. The correction to a query is built from its own *row*, that is, from how that single query spread its attention across the tokens. The correction to a key is built from its *column*, gathering every query that did or did not attend to it. One matrix is therefore corrected a row at a time and the other a column at a time, so the two receive different feedback, and that is part of why they come to specialise. Working the corrections out in full is beyond our forward-pass scope.

8 Causal (Autoregressive) Masking



The attention we have built earlier lets every query reach every key. A model that writes text one token at a time cannot be allowed that, so this section adds a single constraint, each position may attend only to itself and the positions before it, by "hiding" the future from every position even when the future happens to be present.

That "hiding" takes a single addition. Before the softmax, we add a mask M to the scores, holding 0 wherever a position is allowed to attend and $-\infty$ wherever it is not, so the attention computation reads

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V.$$

$$\begin{array}{c} \\ q_1 \\ q_2 \\ \vdots \\ q_n \end{array} \begin{array}{c} \begin{array}{cccc} k_1 & k_2 & \dots & k_n \end{array} \\ \begin{bmatrix} 0 & -\infty & \cdots & -\infty \\ 0 & 0 & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \end{array}$$

The mask M. Added to the scores just before the softmax: 0 (lower triangle, $j \le i$) leaves a score untouched, while $-\infty$ (upper triangle, $j > i$) drives it to zero weight, since $e^{-\infty} = 0$. The divide is the diagonal — query i may attend to key j only when $j \le i$.

Softmax exponentiates every score before normalising, and $\exp(-\infty) = 0$, so a masked position is driven to precisely zero weight. The allowed scores, with 0 added, pass through unchanged.

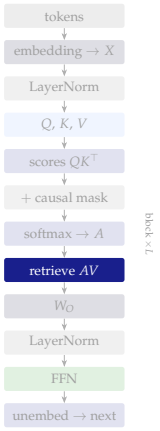
The attention weights that come out are lower-triangular, empty above the diagonal.

$$\begin{array}{c} \\ q_1 \\ q_2 \\ \vdots \\ q_n \end{array} \begin{array}{c} \begin{array}{cccc} k_1 & k_2 & \dots & k_n \end{array} \\ \begin{bmatrix} A_{11} & 0 & \cdots & 0 \\ A_{21} & A_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \end{array}$$

Lower-triangular weights. After the masked softmax every weight above the diagonal is zero (grey); each query's surviving weights (green) still sum to one. Row i spreads its attention over positions 1 through i.

The first position has no past, so it can only attend to itself. The second draws on positions one and two, the third on one through three, and so on. Every position now sees its own token and the ones before it, and nothing it could not yet know.

9 Value Retrieval



The attention weights tell each position *where* to look, but not *what* it finds there. The value vectors carry the what, and putting the two together is the step where information finally moves between positions. The head's output is a single multiplication,

$$\text{head} = A(Q,K)\,V,$$

which we will write simply as AV from here on, the dependence on Q and K now understood.

Retrieval is Mixing

This multiplication is the *mixing* operation from Section 2, where the output row of a matrix product was a *linear combination* of the rows of the right-hand matrix. Here the right-hand rows are the value vectors $v_1, \ldots, v_n$, each a learned projection of one token's representation.

$$\underbrace{\text{head}_i}_{\text{output for } q_i} = a_{i1}\,\underbrace{\mathbf{v}_1}_{\text{value vectors = rows of } V} + a_{i2}\,\mathbf{v}_2 + \cdots + a_{in}\,\mathbf{v}_n$$

head $= AV$, read as mixing. The output for query i is a weighted blend of the *value vectors* $\mathbf{v}_1, \ldots, \mathbf{v}_n$ — the rows of V — with weights $a_{i1}, \ldots, a_{in}$ taken from row i of A. The value vectors are a shared store; only the weights change from one query to the next, so every output is its own recipe over the same ingredients.

Position i's output is a blend of all the value vectors, with each v_j poured in according to how much attention i paid to position j. A position attended to heavily contributes most of its value, one barely attended contributes a trace, and one not attended at all contributes nothing.

What a Value Carries

It remains to say what is actually being mixed. Like the query and key, the value is a learned view of the same token, computed by its own projection,

$$V = X_{\text{norm}} W_V, \qquad v_i = x_{i,\text{norm}} W_V,$$

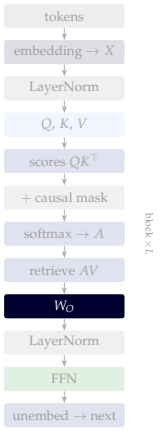
where $W_V \in \mathbb{R}^{d \times d_v}$ is, like W_Q and W_K, a stack of learned columns, each asking its own question of the embedding. The three projections differ only in what their questions are for. The columns of W_Q shape what a token *seeks*, those of W_K shape what makes it *findable*, and those of W_V shape what it *delivers* once found. As before, nothing fixes these questions in advance. The columns of W_V begin random and are shaped by training to capture whatever a token can usefully pass on to the positions that attend to it. How a token is *found* and what it *transmits* can therefore be two different things; a noun located by a pronoun's search sends back its meaning, not its grammar, and nothing requires key and value to agree.

This is the essence of attention. A token arrives at the layer describing mostly itself, and leaves describing itself in context, its representation updated by a blend of the values it

attended to, each in proportion to the attention it drew. A token no longer merely carries its own meaning forward through the network; it absorbs context from the tokens around it, and what it absorbs reshapes what it represents.

That completes a single attention head. Everything it gathers, though, comes back as one blend under one set of proportions, and the next section asks what a layer gains by running several heads at once.

10 Multi-Head Attention



Everything so far has described *single-head* attention, one set of projections W_Q, W_K, W_V, producing one attention pattern and one output. Often, in practice, transformers run several such mechanisms side by side, each with its own projections, and this section explains why that helps and how the parallel outputs are recombined.

Splitting Attention into Heads

A single head can already look in several places at once. Its weights are a distribution, free to spread over a subject, a referent, and a topic alike. What it cannot do is keep those looks apart. The weights sum to one, so every need competes for the same budget, and everything retrieved comes back as one blend, gathered for different reasons yet averaged under the same proportions (Vaswani et al., 2017).

Multi-head attention pulls them apart. It runs h smaller attentions, the *heads*, each with its own budget, its own blend, and its own reason for looking. The model is given h separate sets of projections, one per head, with head i holding

$$W_Q^{(i)}, W_K^{(i)} \in \mathbb{R}^{d \times d_k}, \qquad W_V^{(i)} \in \mathbb{R}^{d \times d_v}, \qquad \text{typically } d_k = d_v = d/h,$$

and each head runs the entire attention mechanism of the previous sections — scores, masking, softmax, and retrieval alike — inside its own slice of the dimensions,

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}} + M\right) V_i \in \mathbb{R}^{n \times d_v}$$

$$Q_i = X_{\text{norm}} W_Q^{(i)}, \quad K_i = X_{\text{norm}} W_K^{(i)}, \quad V_i = X_{\text{norm}} W_V^{(i)}$$

What the split buys is h separate budgets. Each head normalises its own scores, so a token now spends one unit of attention *per head* rather than one in total, and the same pair of tokens draws h weights instead of one. In practice, the per-head matrices are stacked side by side into one wide array, $W_Q = [\, W_Q^{(1)} \cdots W_Q^{(h)} \,] \in \mathbb{R}^{d \times d}$, and a single multiplication $X_{\text{norm}} W_Q$ produces every head's queries at once, with K and V handled likewise. The first d_k columns of $X_{\text{norm}} W_Q$ are exactly Q_1, the next d_k exactly Q_2, and the wide array is h independent projections stored together.

What each head comes to specialise in is left to training. Every head's projections begin random, exactly as in the single-head case, and are shaped by the data, so one head may drift towards grammatical agreement, another towards meaning, another towards position. Figure 3 shows the full computation for $h = 4$: the input X fans out into four parallel streams, each computing its own queries, keys, values, attention pattern, and retrieval, after which the four outputs are concatenated and passed through a final projection, the step we turn to next.

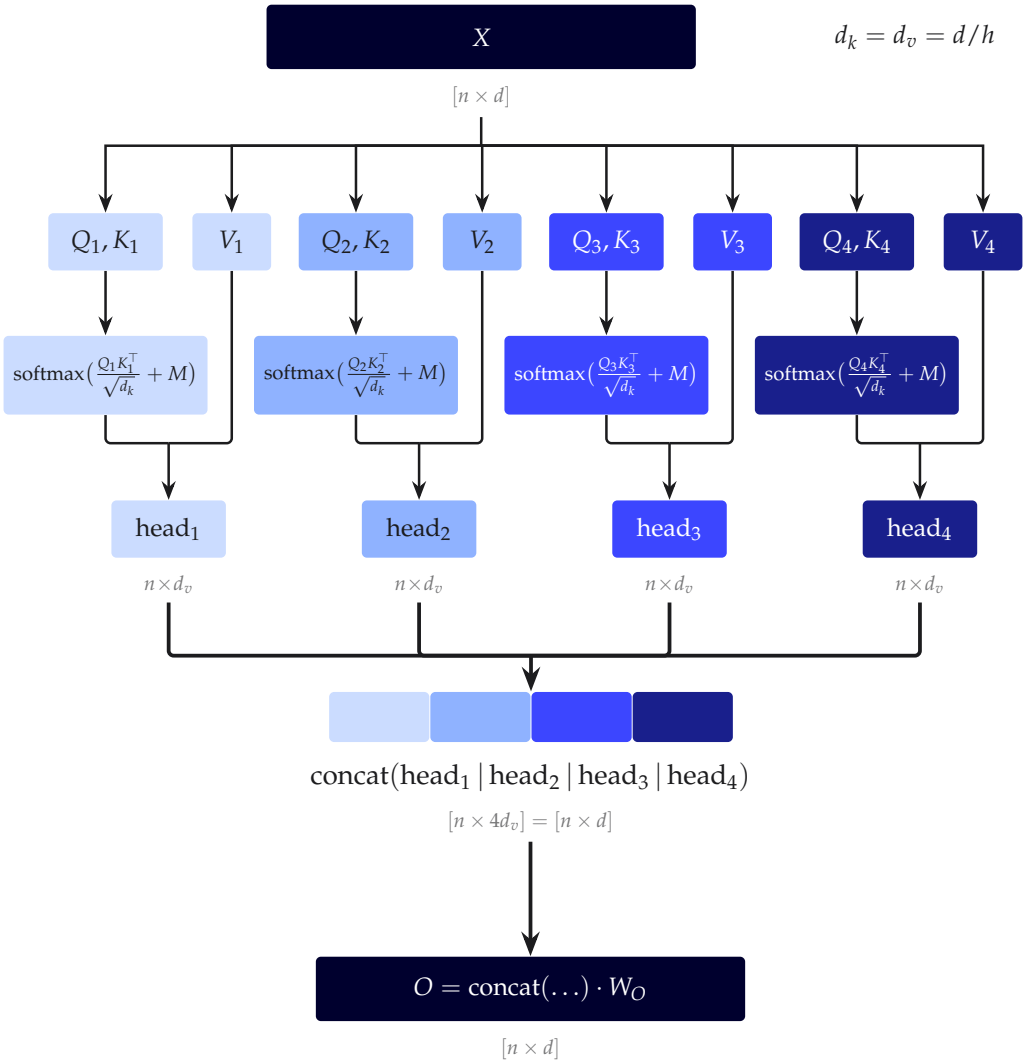


Figure 3: Multi-Head Attention

Writing the Heads Back

Each head returns an $n \times d_v$ matrix. Setting the h of them side by side gives

$$\text{concat}(\text{head}_1, \ldots, \text{head}_h) \in \mathbb{R}^{n \times h d_v} = \mathbb{R}^{n \times d},$$

which lands back at width d because $hd_v = d$. A final matrix $W_O \in \mathbb{R}^{d \times d}$ turns that into the layer's output, and we will use the *mixing* reading. Read W_O by its rows, where row r is a vector $\mathbf{w}_r$ of d numbers, and a vector of d numbers is a direction in the residual stream itself, so every row of W_O already lives in the space the layer must write to. Write $\mathbf{h}_t$ for row t of the concatenation above, one vector of d numbers.

$$\mathbf{h}_t = \big(\underbrace{h_{t,1}, \ldots, h_{t,d_v}}_{\text{head}_1} , \underbrace{h_{t,d_v+1}, \ldots, h_{t,2d_v}}_{\text{head}_2} , \cdots , \underbrace{h_{t,d-d_v+1}, \ldots, h_{t,d}}_{\text{head}_h} \big)$$

Its entry $h_{t,r}$ says how much of direction $\mathbf{w}_r$ to write. Position t's output is their blend,

$$\mathbf{o}_t = \sum_{r=1}^{d} h_{t,r}\, \mathbf{w}_r \in \mathbb{R}^{1 \times d},$$

a single vector in the stream's own coordinates, and O, the n such rows stacked, is the full output of multi-head attention, ready to be added back.

$$\mathbf{o}_t = h_{t,1}\,\mathbf{w}_1 \cdots h_{t,d_v}\,\mathbf{w}_{d_v} + h_{t,d_v+1}\,\mathbf{w}_{d_v+1} \cdots h_{t,2d_v}\,\mathbf{w}_{2d_v} + \cdots + h_{t,d-d_v+1}\,\mathbf{w}_{d-d_v+1} \cdots h_{t,d}\,\mathbf{w}_d$$

$\mathbf{o}_t = \mathbf{h}_t W_O$**, read as mixing.** Every $\mathbf{w}_r$ is a row of W_O, a direction in the d-dimensional stream, and the amount $h_{t,r}$ beside it says how much of that direction position t writes. The d rows come in h groups of d_v, one group per head, and a group spans the subspace that head writes into. Within a group the three dots stand for the $d_v - 2$ terms not shown, all of them added.

The Residual Connection

Attention's output is added back to the same representation it was computed from, through a residual[9] connection,

$$X' = X + O,$$

so that X' now holds two things at once:

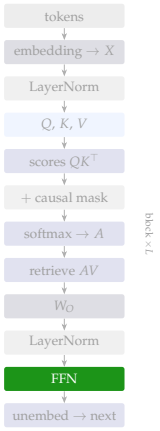
- the original token embedding, preserved untouched through the addition, and
- the context gathered from other positions, added on top by attention.

Nothing has been overwritten. Attention only *adds* to what was already there, which is also what lets these blocks be stacked to great depth. Because each sublayer writes its result onto a running representation rather than replacing it, it only has to learn a small correction to what the previous layers built, and the addition gives gradients a clean path back through the network during training.

This is also the limit of what attention can do. Although the attention weights are themselves produced nonlinearly (by the softmax), the step that uses them is linear. Given the weights, each position's output is a plain weighted average of value vectors already present, so attention can only ever return a linear mixture of its inputs, never a nonlinear transformation of their content. It can route information between positions and average it together, but the result is always built from the value vectors it started with, never a genuinely new quantity computed from them. This given-the-weights linearity is a workhorse of the interpretability literature (Elhage et al., 2021). That is the job of the next component, the feedforward network, which takes the context attention has assembled at each position and actually processes it.

[9]The name is inherited from residual networks (He et al., 2016), where the input is carried forward unchanged and a sublayer needs to learn only the *residual*, the difference between the input and the desired output. This running representation is often called the *residual stream*. Each sublayer reads from it and adds its contribution back, so the stream accumulates the embedding, then attention's context, then the feedforward network's processing, and so on through the layers.

11 The Feedforward Network



Multi-head attention left each position holding the context it had gathered from the rest of the sequence, recombined across heads and added back into its representation. But attention only ever *moved* information between positions; it never transformed the content it carried. That is the feedforward network's job. Attention *moves* information between positions, the feedforward network *processes* it.

Once attention has gathered context from other tokens into each position, the feedforward network — a small two-layer network — works on each position on its own, putting every position through the very same transformation. This is its defining contrast with attention, which existed to let positions communicate. Attention mixes across tokens, the feedforward network thinks about each token in isolation.

Loosely speaking, the network reads then writes, and it does so through a shape of expand, process, contract. Each position's d-dimensional vector is first widened to a much larger d_{ff} dimensions, typically four times d, then passed through a nonlinearity, then squeezed back down to d. The first matrix (W_1) scores the incoming vector against d_{ff} learned directions at once (measurement), so the widening is really a bank of detectors, each asking whether some pattern is present and reporting how strongly. The nonlinearity then keeps the patterns that fired and suppresses the rest, and the second matrix (W_2) *mixes* those surviving signals back down into a single d-dimensional vector, assembled as a recipe over learned components.

The Architecture

As in the attention sublayer, the input is first passed through LayerNorm (Pre-Norm), stabilising its scale so the patterns W_1 looks for stay reliable from layer to layer (Section 5). Following the notation of Figure 2b, we write X' for the post-attention representation entering this sublayer and $X'_{\text{norm}} = \text{LayerNorm}(X')$ for its normalised form.

The network applies two linear maps with a nonlinearity, ReLU[10], between them,

$$\text{FFN}(X'_{\text{norm}}) = \text{ReLU}(X'_{\text{norm}} W_1)\, W_2,$$

where $X'_{\text{norm}} \in \mathbb{R}^{n \times d}$ is the normalised post-attention representation, $W_1 \in \mathbb{R}^{d \times d_{\text{ff}}}$ does the widening, and $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$ the squeezing back.

The First Layer Measures

This is the measurement reading of Section 2. Each column of W_1 is a learned question, a d-dimensional pattern, and dotting a position's vector against it scores how strongly that

[10] We use ReLU throughout for simplicity, following the original Transformer (Vaswani et al., 2017). Many modern LLMs use alternatives, such as the smoother GELU or gated units like SwiGLU and GeGLU. These can improve performance, but the expand, process, contract role of the sublayer remains the same. We also drop the bias terms throughout this section for simplicity, since they add nothing to the measurement-and-mixing reading that follows. A reader who prefers the fuller form can restore them, writing each linear map as $XW + b$ in place of XW.

pattern is present,

$$(X'_{\text{norm}}W_1)_{ij} = x'_{i,\text{norm}} \cdot c_j,$$

with c_j the j-th column. Across all d_{ff} columns, every position receives d_{ff} scores, one per pattern. ReLU then zeroes the negatives, keeping a score only where the position's vector aligned positively with the column,

$$H = \text{ReLU}(X'_{\text{norm}}W_1) \in \mathbb{R}^{n \times d_{\text{ff}}}.$$

The result H is best pictured as an on/off pattern. For each token, every one of the d_{ff} entries is either zero, where the pattern was absent, or positive, where it fired, marking which patterns that token triggered (and, in the size of each nonzero entry, how strongly). Which entries are active varies from token to token, and that input-dependent pattern of who-fired is the only thing the second layer sees.

The Second Layer Mixes

The second layer turns that firing pattern into an output. Each *row* of W_2 is a learned building block, one paired with each of the d_{ff} patterns, and a token's output is the sum of just the rows whose patterns fired, each scaled by how strongly,

$$\text{FFN}(x'_{i,\text{norm}}) = \sum_{j=1}^{d_{\text{ff}}} H_{ij}\,(\text{row } j \text{ of } W_2).$$

Wherever an entry of H_i is zero, its row is multiplied by zero and drops out; the patterns the token actually triggered select their rows from W_2 and write them into the output in proportion to how strongly they fired. The first layer detects which features are present, and H is the switchboard that lets exactly those features choose what gets written.

Together with attention, which decides what each position gets to see, the feedforward network decides what to make of it, and the two alternate, layer after layer, building richer representations as they go.

The Residual Connection

As after attention, the feedforward output is added to the representation rather than replacing it,

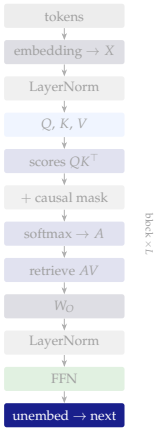
$$X'' = X' + \text{FFN}(X'_{\text{norm}}),$$

so X'' now carries three things:

- the original token embedding, from the initial embedding step,
- the context gathered from other positions, added by attention, and
- the features computed from that context, added now by the feedforward network.

This completes one transformer block. The output X'' becomes the input to the next, which runs its own attention, now over these richer representations, and its own feedforward network, potentially detecting higher-order patterns built from the features the earlier layers laid down.

12 The LM Head



This is the last step, and the one where everything built so far pays off. The stack of L layers has done its work, and we hold a final representation $X''^{(L)} \in \mathbb{R}^{n \times d}$, one richly contextualised vector per position. A final LayerNorm settles these onto a stable scale,

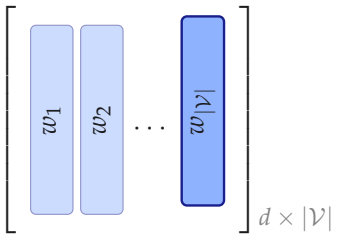
$$X_{\text{final}} = \text{LayerNorm}(X''^{(L)}) \in \mathbb{R}^{n \times d},$$

and all that remains is to turn each of these vectors into an actual prediction, a probability for every word that could come next. Everything in this section — the final layer norm, the unembedding projection, and the softmax — together forms what practitioners call the *language-modelling head*, or *LM head*, a small prediction head sitting on top of the transformer body. This last move rests on two things already in hand.

The first is causal masking. Because every attention layer was masked, row i of X_{final} was built from tokens 1 through i alone. Row 3 has never seen token 4 or anything beyond it. Each position's representation is therefore a summary of the past alone, which is the only thing a genuine next-token prediction is ever allowed to depend on. The second is the measurement view of Section 2. Turning a representation into vocabulary scores is, once again, nothing but dot products against a set of learned directions, the same move that opened the attention mechanism, now aimed at the vocabulary instead of at other tokens. We take the final step in three short stages, unembedding, then softmax, and finally the loss that ties the whole thing to training.

The Unembedding Projection

To score the vocabulary, we measure each representation against one learned direction per word. These directions are the columns of the unembedding matrix $W_{\text{unembed}} \in \mathbb{R}^{d \times |\mathcal{V}|}$, where $|\mathcal{V}|$ is the vocabulary size,



W_{unembed}. One column per vocabulary token. A representation is scored against all of them; the scores are the logits. (Often $W_{\text{embed}}^{\top}$.)

There are $|\mathcal{V}|$ columns, one per vocabulary token, and for $j = 1, \ldots, |\mathcal{V}|$ the column $w_j \in \mathbb{R}^d$ is token j's *signature*: a pattern describing what a representation should look like for token j to be the likely next word. Scoring a representation against all of them at once — one dot product per column — yields one number per token, the alignment of the representation with each signature, and these are the logits passed to the softmax.

A common and elegant choice ties these signatures to the input embeddings of Section 4. There, the embedding matrix W_{embed} stored each token's vector as a *row*; transposing it,

$W_{\text{embed}}^{\top}$, stands those rows up as columns, and setting $W_{\text{unembed}} = W_{\text{embed}}^{\top}$ makes each token's output signature identical to its input embedding. This *weight tying* saves a large number of parameters, since one matrix serves both ends, but it also says something meaningful: a token is predicted to the extent that the running representation has come to *resemble that token's own embedding*. The model is, in effect, asking at each position which word the context now looks most like.

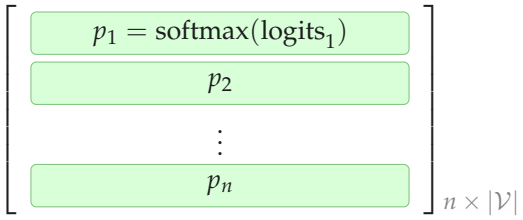
The scoring itself is one matrix multiplication.

$$\text{logits} = X_{\text{final}}\, W_{\text{unembed}} \in \mathbb{R}^{n \times |\mathcal{V}|}, \qquad \text{logits}_{ij} = x_i^{(\text{final})} \cdot w_j.$$

Read through the measurement lens, entry (i, j) is the alignment between position i's representation and token j's signature. Strong alignment yields a high score, meaning the model finds token j a good fit for what position i has come to mean. These raw scores are called *logits*, and row i holds one for every word in the vocabulary.

From Scores to Probabilities

Logits are unbounded scores, not yet probabilities. Softmax, applied across each row, turns each row of logits into a probability distribution,



Output probabilities. Each contextual representation is multiplied by W_{unembed} to give $|\mathcal{V}|$ logits, then softmax turns row i into a probability distribution over the next token.

where p_{ij} is the probability the model assigns to token j following position i, and each row sums to one.

Training: Learning to Predict

To train the model we need the right answer at each position, and the text already provides it. For every position but the last, the target y_i is simply the token that follows it, read straight from the data with no labelling. Each such position is its own small prediction problem with its answer sitting inside the sequence, so one input of length n supplies $n - 1$ training signals.

Every position is therefore scored with the probability the model assigned to the token that actually followed. The cross-entropy loss records this,

$$L_i = -\log P_{i,y_i},$$

which is near zero when the model was confident and correct, since P_{i,y_i} near one makes $-\log P_{i,y_i}$ near zero, and grows large when the model gave the true word little probability. Averaging over the $n - 1$ predicting positions gives the loss for the whole sequence,

$$L = -\frac{1}{n-1} \sum_{i=1}^{n-1} \log P_{i,y_i},$$

and minimising it, across vast amounts of text, is the entire training objective. Every weight we have met, the embeddings, the projection matrices W_Q, W_K, W_V, the head mixer W_O, the feedforward weights, and the unembedding, is adjusted to make this number smaller.

Inference: Generating Text

Given a prompt of n tokens, we run a single forward pass, then look only at row n, the one position that has seen the entire prompt. Its distribution is the model's prediction for token $n+1$. We choose a token from it, append that token to the sequence, and run again, now with $n+1$ tokens, and repeat.

The one freedom left is how to pick a token from each distribution, and several techniques exist. *Greedy* decoding simply takes the single most likely token at every step, a fully deterministic choice. Sampling instead draws a token at random according to the distribution, and *temperature* scaling tunes how that draw behaves by dividing the logits by a constant T before the softmax. A low temperature, $T < 1$, sharpens the distribution and concentrates the draw on the model's favourites; a high temperature, $T > 1$, flattens it, raising the chance that less likely tokens are chosen and making generation more varied. Generation continues until the model emits the end-of-sequence token `<eos>`, reaches a length limit, or meets whatever other stopping rule the application sets.

And there the journey closes. A sequence of token IDs entered as meaningless integers, became vectors carrying meaning and position, and passed through layers that alternately let each token gather context from the others and then process what it had gathered. Throughout, two readings of a single operation did the work, measurement asking how strongly one vector aligns with a learned direction, and mixing assembling each output as a recipe over learned components. At the very end, the same measurement reappears one last time, scoring each contextualised representation against the vocabulary to predict what comes next. Stack these blocks deep, train them on enough text, and from these few repeated parts what we are willing to call "intelligence" emerges.

References

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. `https://transformer-circuits.pub/2021/framework/index.html`.

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. *Transformer Circuits Thread*, 2022. `https://transformer-circuits.pub/2022/toy_model/index.html`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013b.

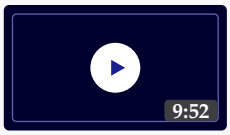
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

A Prerequisites: Video Resources

This appendix collects video resources for readers who want a thorough review of the mathematical foundations. The videos are organised into two sections.

A.1 Linear Algebra

The *Essence of Linear Algebra* series by 3Blue1Brown provides geometric intuition for the concepts used throughout this document. The following chapters are particularly relevant:



Vectors

3Blue1Brown — Essence of Linear Algebra, Ch. 1

Introduces vectors as arrows in space and as lists of numbers. Establishes coordinates, vector addition, and scalar multiplication.

▶ WATCH youtube.com/watch?v=fNk_zzaMoSs

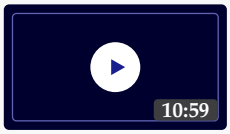


Linear combinations, span, and basis vectors

3Blue1Brown — Essence of Linear Algebra, Ch. 2

Defines linear combinations as weighted sums of vectors. Introduces basis vectors and the idea that any vector can be built by scaling and adding basis vectors.

▶ WATCH youtube.com/watch?v=k7RM-ot2NWY

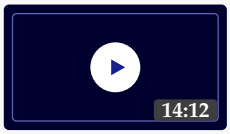


Linear transformations and matrices

3Blue1Brown — Essence of Linear Algebra, Ch. 3

Shows that matrices encode linear transformations. The columns of a matrix tell you where the basis vectors land. Matrix–vector multiplication computes where a vector ends up after the transformation.

▶ WATCH youtube.com/watch?v=kYB8IZa5AuE

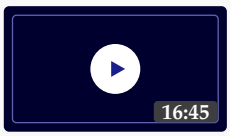


Dot products and duality

3Blue1Brown — Essence of Linear Algebra, Ch. 9

Reveals the geometric meaning of the dot product: projecting one vector onto another and measuring the result. Connects dot products to linear transformations via duality.

▶ WATCH youtube.com/watch?v=LyGKycYT2v0



Abstract vector spaces

3Blue1Brown — Essence of Linear Algebra, Ch. 16

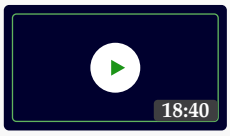
Generalises vectors beyond arrows and lists of numbers. Shows that functions can be vectors too, and introduces the axioms that define a vector space. Explains why linear algebra applies to any setting with sensible notions of addition and scaling.

▶ WATCH youtube.com/watch?v=TgKwz5Ikpc8

Total runtime ≈ 1 hour

A.2 Neural Networks

The following videos provide foundational knowledge on neural networks, including the concepts of parameterised functions, gradient descent, and backpropagation.

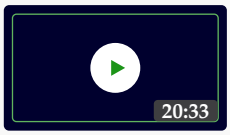


But what is a neural network?

3Blue1Brown — Deep Learning, Ch. 1

Introduces neural networks as layered structures of neurons. Explains how weighted sums and activation functions combine to recognise patterns, using handwritten digit recognition as an example.

▶ WATCH youtube.com/watch?v=aircAruvnKk

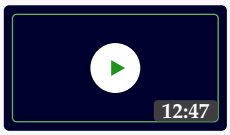


Gradient descent, how neural networks learn

3Blue1Brown — Deep Learning, Ch. 2

Explains gradient descent as the process of iteratively adjusting weights to minimise a loss function. Visualises the loss landscape and the path towards better solutions.

▶ WATCH youtube.com/watch?v=IHZwWFHWa-w

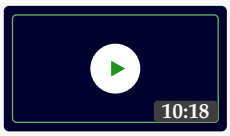


Backpropagation, intuitively

3Blue1Brown — Deep Learning, Ch. 3

Builds intuition for backpropagation: how errors at the output propagate backward through the network to inform weight updates. Focuses on the conceptual picture rather than formulas.

▶ WATCH youtube.com/watch?v=Ilg3gGewQ5U

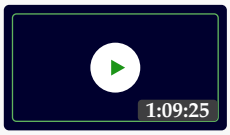


Backpropagation calculus

3Blue1Brown — Deep Learning, Ch. 4

Walks through the calculus of backpropagation: the chain rule applied layer by layer. Shows how gradients are computed in practice.

▶ WATCH youtube.com/watch?v=tIeHLnjs5U8



MIT Introduction to Deep Learning | 6.S191

Alexander Amini

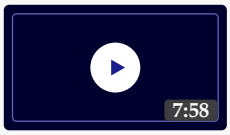
Covers perceptrons, activation functions, loss functions, gradient descent, backpropagation, and regularisation. Provides a comprehensive introduction to neural network foundations.

▶ WATCH youtube.com/watch?v=alfdI7S6wCY

Total runtime ≈ 2 hours 10 minutes (≈ 1 hour excluding the MIT lecture)

B Further Resources: Large Language Models

This appendix provides additional video resources for readers who want deeper context on LLMs and transformers. These videos by 3Blue1Brown offer visual explanations that complement the mathematical treatment in this document.

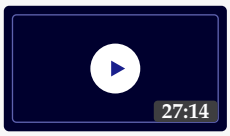


Large Language Models explained briefly

3Blue1Brown

A concise overview of what LLMs are and how they work. Provides context for the detailed mechanics covered in this document.

▶ WATCH youtube.com/watch?v=LPZh9BOjkQs

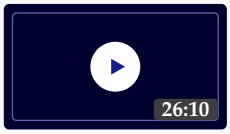


Transformers, the tech behind LLMs

3Blue1Brown — Deep Learning, Ch. 5

Introduces the transformer architecture with visual animations. Covers embeddings, attention, and how information flows through the network.

▶ WATCH youtube.com/watch?v=wjZofJX0v4M

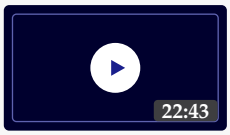


Attention in transformers, step-by-step

3Blue1Brown — Deep Learning, Ch. 6

A detailed walkthrough of the attention mechanism. Visualises how queries, keys, and values interact to route information between positions.

▶ WATCH youtube.com/watch?v=eMlx5fFNoYc



How might LLMs store facts

3Blue1Brown — Deep Learning, Ch. 7

Explores how factual knowledge might be encoded in transformer weights. Discusses the MLP layers as potential sites of fact storage and retrieval.

▶ WATCH youtube.com/watch?v=9-Jl0dxWQs8

Total runtime ≈ 1.5 hours

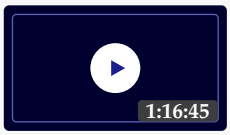
Further reading. *A Mathematical Framework for Transformer Circuits* (Elhage et al., 2021) extends this document's central ideas, developing the residual-stream view drawn in Figures 2a and 2b into a full framework for analysing what trained transformers compute.

C Further Resources: Stanford CS25 Transformers United

CS25: Transformers United is a Stanford seminar series on transformer research, led in part by one of this document's authors and open to anyone, in person or by livestream. Each week a researcher working on the architecture gives a talk on their own work, and the recordings are made public. The sixth edition ran in the spring of 2026; its nine lectures are collected below, and the course pages are at `web.stanford.edu/class/cs25`.

Where this document stays inside a single trained layer and derives it, these talks show what has been built around it: alternatives to the architecture, the engineering of training at scale, and the applications it now carries. They assume roughly what the preceding sections have supplied.

C.1 The Architecture and Its Alternatives

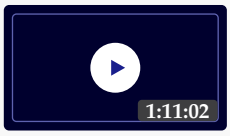


Overview of Transformers

Steven Feng & Karan Singh — Stanford University

Opens the series with the history of machine learning and natural language processing, the transformer itself, and the impact it has had. Closes on recent trends, breakthroughs, applications, and the challenges that remain.

▶ WATCH `youtube.com/watch?v=bHSDPgZYie0`

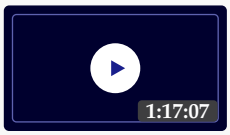


From Representation Learning to World Modeling through Joint Embedding Predictive Architectures

Hazel Nam & Lucas Maes — Brown University; Mila, Université de Montréal

World models are moving away from reconstructing their input and towards predicting in a latent space instead. Presents two recent JEPA-based approaches that illustrate the shift from complementary angles.

▶ WATCH `youtube.com/watch?v=GBd7iuJkW08`



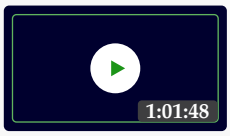
On the Tradeoffs of State Space Models and Transformers

Albert Gu — CMU and Cartesia AI

A high-level overview of the state space model, a subquadratic alternative to the transformer. Covers the core characteristics and the design choices that separate SSMs from the architecture built up here.

▶ WATCH `youtube.com/watch?v=OyimE74UMF8`

C.2 Training and Scale

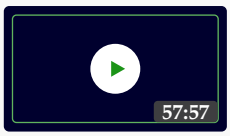


The Ultra-Scale Talk: Scaling Training to Thousands of GPUs

Nouamane Tazi — Hugging Face

Training a large language model takes more than raw compute; it takes a working understanding of parallelism and of the architectural choices that interact with it. Goes into the practicalities of training at ultra-scale.

▶ WATCH youtube.com/watch?v=I5BKi32IEa8

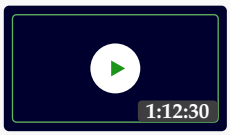


From Next-Token Prediction to Next-Generation Intelligence: The Future of Pretraining

Shrimai Prabhumoye — Mistral AI, previously NVIDIA

Recent progress in the design of pretraining algorithms. Emphasises the ordering of data, the integration of reasoning-centred data, and reinforcement-based objectives in shaping what a model can do.

▶ WATCH youtube.com/watch?v=e_H_tkpCAK4



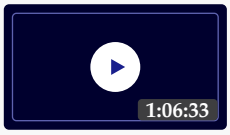
Distinct Modes of Generalization from Parameters and Context, and Paths to Bridge the Gap

Andrew Lampinen — Anthropic

A model can be taught something by two routes: training on it, which updates the parameters, or presenting it in the context. Describes striking differences between the ways a model generalises from each, and how the gap might be closed.

▶ WATCH youtube.com/watch?v=dJtHauhRasc

C.3 Applications and Systems

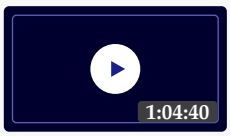


Advancing Science and Medicine with Collaborative AI Agents

Vivek Natarajan — Google DeepMind

Introduces general-purpose systems built to accelerate scientific discovery and widen access to medical expertise. Among them the AI co-scientist, a multi-agent Gemini-based system that generates and refines research hypotheses.

▶ WATCH youtube.com/watch?v=jFdH7n6BAl0

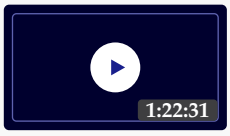


From Language Models to Native Multimodal Intelligence

Victoria Lin — Thinking Machines Lab

Next-token prediction at scale induces remarkably general capabilities: acquiring knowledge, following instructions, reasoning and planning. Traces how the ideas behind language models have shaped multimodal ones, and where the next difficulties lie.

▶ WATCH youtube.com/watch?v=NDdc39KYqDU



Serving Transformers: Lessons from the Trenches of Production Inference

Charles Frye — Modal

Training is only the first step in building an application. Turns to inference, forward passes without backward passes, and to what serving a model to real clients actually demands.

▶ WATCH youtube.com/watch?v=ZUdIsRZhWXI

Five earlier editions of the series sit on the same channel, among them talks by Geoffrey Hinton, Ashish Vaswani and Andrej Karpathy.

D The Model as Code: Karpathy's microgpt

This note took the transformer apart one matrix at a time. Andrej Karpathy's *microgpt* puts it back together as code: a single file of about two hundred lines of plain Python, with no libraries, that trains a small GPT on a list of names and then samples new ones. Reproduced below is the function that defines the model, his code as written, with a few comments of ours at the top to tie it to this document. The complete runnable file, with the tokeniser, the tiny autograd engine, the training loop and the sampling loop, is in the post that accompanies it: karpathy.github.io/2026/02/12/microgpt.

```
# gpt() from Karpathy's microgpt, as written. How it maps onto this document:
# - One token per call; keys/values cache earlier positions' K and V, per layer, so
#   attention only sees the past: the causal mask of Section 7 with no mask matrix.
# - state_dict holds the weights: wte/wpe are the embeddings (Section 4), attn_wq/
#   wk/wv/wo are W_Q, W_K, W_V, W_O; mlp_fc1/fc2 are W_1, W_2; lm_head: unembedding.
# - linear is the matrix-vector product of Section 2, softmax that of Section 7, and
#   rmsnorm a LayerNorm (Section 5) without the centring step.
def gpt(token_id, pos_id, keys, values):
    tok_emb = state_dict['wte'][token_id] # token embedding
    pos_emb = state_dict['wpe'][pos_id] # position embedding
    x = [t + p for t, p in zip(tok_emb, pos_emb)] # joint token and position embedding
    x = rmsnorm(x)

    for li in range(n_layer):
        # 1) Multi-head attention block
        x_residual = x
        x = rmsnorm(x)
        q = linear(x, state_dict[f'layer{li}.attn_wq'])
        k = linear(x, state_dict[f'layer{li}.attn_wk'])
        v = linear(x, state_dict[f'layer{li}.attn_wv'])
        keys[li].append(k)
        values[li].append(v)
        x_attn = []
        for h in range(n_head):
            hs = h * head_dim
            q_h = q[hs:hs+head_dim]
            k_h = [ki[hs:hs+head_dim] for ki in keys[li]]
            v_h = [vi[hs:hs+head_dim] for vi in values[li]]
            attn_logits = [sum(q_h[j] * k_h[t][j] for j in range(head_dim)) /
    ↪ head_dim**0.5 for t in range(len(k_h))]
            attn_weights = softmax(attn_logits)
            head_out = [sum(attn_weights[t] * v_h[t][j] for t in range(len(v_h))) for j in
    ↪ range(head_dim)]
            x_attn.extend(head_out)
        x = linear(x_attn, state_dict[f'layer{li}.attn_wo'])
        x = [a + b for a, b in zip(x, x_residual)]
        # 2) MLP block
        x_residual = x
        x = rmsnorm(x)
        x = linear(x, state_dict[f'layer{li}.mlp_fc1'])
        x = [xi.relu() for xi in x]
        x = linear(x, state_dict[f'layer{li}.mlp_fc2'])
        x = [a + b for a, b in zip(x, x_residual)]

    logits = linear(x, state_dict['lm_head'])
    return logits
```