

1 Why Attention?

A word embedding assigns every token *one fixed vector*, no matter what surrounds it. But meaning depends on context:

- “The **bank** of the river” vs. “The **bank** approved my loan.”
- “The animal didn’t cross the street because **it** was too tired.” What does “it” refer to?
- “The angry dog **barked**”: on its own, *barked* says nothing about *who* barked or *how* they felt. That information sits in neighboring words.

A static embedding for “bank” must be a blurry average of all its senses. What we want is a way for each token to *look at the other tokens in the sentence and update its own vector accordingly*.

Key idea

Attention is a mechanism by which every token gathers information from the other tokens in the sequence, taking more from the tokens that are most relevant to it. The output is a new vector for each token that is now *context-dependent*.

Historical note. Attention was introduced for recurrent translation models (Bahdanau et al., 2014). The 2017 paper *Attention Is All You Need* (Vaswani et al.) showed that attention alone, with no recurrence, is enough, and this architecture is the *Transformer*. Nearly all modern large language models descend from it.

2 The Database Analogy

This section builds the whole idea of attention using nothing but a picture of a database. No vectors or dot products yet; those arrive in Part 3 as the machinery that makes the picture precise.

2.1 Every word is a record with attributes

Think of a sentence as a small database in which **every word is a record**. A record describes what kind of word it is and what it knows about the world, as a set of **attributes** (fields), each of which may hold a **value**. Possible attributes include:

- the *kind of word*: noun, verb, adjective, determiner, ...
- the *kind of thing*: animal, vehicle, musical instrument, person, ...

- *properties*: size, weight, color, sound, appearance, emotion, function, ...

and the values stored in those attributes are things like *big*, *small*, *blue*, *red*, *angry*, *loud*, *music*, *transportation*, *animal*, ...

Here is the sentence “**the angry dog barked**” stored as four records. A dash means the field is empty for that word, and a question mark marks a field the word *would like to know but cannot fill by itself*.

Attribute	the	angry	dog	barked
Word type	determiner	adjective	noun	verb
Kind of thing	—	—	animal	—
Emotion	—	angry	—	?
Sound	—	—	barking	loud, abrupt
Size / color	—	—	medium / —	—
Function	points at a specific thing	describes a noun	pet, guard	an action
Who does it?	—	—	—	?

Notice what is going on. The word *barked*, read in isolation, is an *incomplete record*: it knows it is a loud action, but its “who did it?” and “how did they feel?” fields are blank. Yet the answers are sitting in the other records: *dog* knows it is an animal that barks, and *angry* knows about the emotion.

Key idea

Each word is a record with some fields filled in and others missing. **Understanding a sentence means filling in a word’s missing fields using the other records around it.** Attention is the mechanism that does this filling in.

2.2 Step 1: the word throws out a query

To fill its gaps, *barked* sends out a **query**: a description of the kind of record it is looking for. In plain words:

“I am a verb about sound. I need a *doer*: ideally a noun that is an animate thing. If anyone nearby carries an *emotion*, I would like that too.”

A query is not a single exact request such as `id = 7`. It is a fuzzy description of the *kinds of attributes* the word is hunting for, and it may care about several of them at different strengths. (Here: “doer” matters a lot, “emotion” a little.)

2.3 Step 2: every record advertises its keys

Each record also carries **keys**: the part of the record that says *which kinds of questions it is able to answer*. Roughly, a record's keys are the attributes it is richly filled in for:

- *dog*: “I am an animate noun; I can be a doer; I am an animal.”
- *angry*: “I am an adjective; I carry an emotion.”
- *the*: “I am a determiner; I carry very little content.”
- *barked* (itself): “I am a verb; I carry a sound.”

The query is compared against the keys of *all* the records in the context, including the word's own record.

2.4 Step 3: rank (weigh) the records by how well their keys match

For each record we ask: how well do your keys match my query? The result is a ranking, and then we turn the ranking into **weights** that are positive and add up to 100%.

Record	Its keys (what it advertises)	Match with <i>barked</i> 's query	Weight	Its value (what it hands over)
<i>dog</i>	animate noun, can be a doer	strong	$\approx 50\%$	“animal; a dog; makes barking sounds”
<i>angry</i>	adjective, carries an emotion	medium	$\approx 25\%$	“emotion: angry”
<i>barked</i>	verb, carries a sound	weak	$\approx 12\%$	“loud, abrupt sound”
<i>the</i>	determiner, little content	weak	$\approx 12\%$	(nearly nothing)

(We will compute these percentages exactly in Part 4. For now, just read them as “*dog* matters most, *angry* next, the rest barely.”)

2.5 Step 4: pull the values and take a weighted average

Having ranked the records, *barked* now **pulls the value** of each one, *in proportion to its weight*, and blends them:

$$\text{new } barked = 50\% \cdot (\text{dog's value}) + 25\% \cdot (\text{angry's value}) + 12\% \cdot (\text{own value}) + 12\% \cdot (\text{the's value}).$$

The result is a rewritten record for *barked*: its “who?” field is now filled with *a dog* and its “emotion” field with *angry*. The word has **enriched itself from its context**, and it now means something like “an angry dog's loud barking.”

	<i>barked before</i>	<i>barked after attention</i>
Doer (“who?”)	?	a dog (animal, barks)
Emotion	?	angry
Sound	loud, abrupt	loud, abrupt (blended in)

2.6 The same idea, written like a database query

In an ordinary database you would write something like this. It returns *the single row that matches exactly*:

```
SELECT value FROM words WHERE key = 'doer';      -- one exact answer
```

Attention behaves like a *soft* version of the same query: “score *every* row against my description, then give me a weighted blend of the values.” In loose pseudo-SQL:

```
SELECT SUM( match(query, key) * value ) FROM words;  -- weights sum to 1
```

and in a few lines of Python:

```
def soft_lookup(query, keys, values):
    scores = [match(query, k) for k in keys]    # how well does each key match?
    weights = normalize(scores)                 # positive, sums to 1
    return sum(w * v for w, v in zip(weights, values))
```

Instead of retrieving *one* record, we retrieve a *blend of all records*, weighted by how well they match. In Part 3 we replace `match` by the dot product and `normalize` by softmax.

2.7 Why keys and values are different things

A record’s **key** decides *whether it gets picked*; its **value** decides *what it contributes once picked*. These are separate because “what makes me findable” and “what I give you” are different jobs. The word *angry* is findable by the key “I carry an emotion,” but what it hands over is the specific emotion value *angry*. A different emotion word, such as *joyful*, would have nearly the same key and a very different value.

Intuition

Query = a question I ask. **Key** = a label saying which questions I can answer. **Value** = the answer I actually give. A good analogy is a library: the query is what you ask the librarian, the keys are the catalog tags on each book, and the values are the contents of the books you take home. You choose books by their tags but you read their contents.

2.8 Everybody does this at once, and with several questions

Two further observations will matter later.

- **Every word is both a questioner and a record.** While *barked* is looking up *dog*, the word *dog* is itself throwing out a query (“is anything describing me?”) and finding *angry*. All words update at the same time, each from its own query.
- **One word can ask several different questions.** *barked* may also want to know *where* or *at whom*. Each is a different query aimed at different keys. Running several such queries in parallel is exactly *multi-head attention* (Part 5).

2.9 Which records can a word look at?

In an *encoder* (e.g. for understanding a sentence) a word may query *all* records in the context. In a left-to-right text generator (GPT-style), a word may only query the *previous* words, since the future has not been written yet; this restriction is called *causal masking* (Part 8). In our example *barked* is the last word, so it can see everything either way.

2.10 From the picture to the math: a translation table

Database picture	Attention term	Made precise in
A word’s record	a row of $\mathbf{X}$ (the token’s vector)	Part 4, Step 1
The question the word asks	query $\mathbf{q}$	learned map $\mathbf{W}^Q$
The attributes a record advertises	key $\mathbf{k}$	learned map $\mathbf{W}^K$
The content a record hands over	value $\mathbf{v}$	learned map $\mathbf{W}^V$
How well a key matches the query	similarity score $\mathbf{q} \cdot \mathbf{k}$	Part 3 (dot product)
Turning scores into percentages	attention weights	Part 3 (softmax)
Blending the values	weighted average $\sum_j A_{ij} \mathbf{v}_j$	Part 4, Step 4
Several questions at once	multiple heads	Part 5
Rewriting the records for next round	stacked layers	Part 7

3 Two Small Tools

The picture needs two ingredients made precise: a number for “how well does a key match the query?” and a way of turning those numbers into weights that add up to 1.

3.1 Dot product = match score

Once queries and keys are vectors, “how well do they match?” becomes the dot product. For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$,

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^d a_i b_i.$$

It is large and positive when the vectors point in a similar direction, near zero when they are unrelated (orthogonal), and negative when they point in opposite directions. Back in the database picture: if each coordinate stands for an attribute, the query’s coordinates say “how much I care about this attribute” and the key’s coordinates say “how richly I carry this attribute.” Multiplying and summing rewards records that carry exactly what the query cares about.

3.2 Softmax = scores to weights (the ranking)

Given scores $s_1, \dots, s_n$, softmax produces

$$w_j = \frac{e^{s_j}}{\sum_{m=1}^n e^{s_m}}.$$

All $w_j > 0$ and $\sum_j w_j = 1$. A large score takes most of the weight, but nothing is ever exactly zero, so the lookup is *soft*. Softmax is also differentiable, which is essential: it lets gradients flow through the lookup so that *the model can learn what to look up*.

4 One Attention Head, Step by Step

4.1 Setup

Suppose a sequence has n tokens. After embedding, each token is a vector of dimension d_{model} . Stack them as the rows of a matrix

$$\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}.$$

(Row i is “record i ” of our database.)

4.2 Step 1: create queries, keys, values

We do not use $\mathbf{X}$ directly as queries, keys and values. Each token's embedding is passed through three *learned linear maps*:

$$\mathbf{Q} = \mathbf{X} \mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X} \mathbf{W}^K, \quad \mathbf{V} = \mathbf{X} \mathbf{W}^V,$$

with $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $\mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$. Row i of $\mathbf{Q}$ is the query $\mathbf{q}_i$ of token i , and so on.

Intuition

The same embedding plays three roles. $\mathbf{W}^Q$ learns *what to ask*, $\mathbf{W}^K$ learns *what to advertise*, and $\mathbf{W}^V$ learns *what to hand over*. These are ordinary weight matrices, trained by backpropagation like everything else. Nobody ever tells the model that “doer” or “emotion” should be an attribute.

4.3 Step 2: score every query against every key

$$\mathbf{S} = \mathbf{Q} \mathbf{K}^\top \in \mathbb{R}^{n \times n}, \quad S_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j.$$

Entry S_{ij} says: “how relevant is token j to token i ?” Note that $S_{ij} \neq S_{ji}$ in general, since queries and keys are different. (*barked* looking for *dog* is not the same question as *dog* looking for *barked*.)

4.4 Step 3: scale and softmax each row

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{S}}{\sqrt{d_k}}\right),$$

where softmax is applied *row by row*. Row i of $\mathbf{A}$ is a probability distribution over tokens: “how much of my output should come from each token?” These are the **attention weights**.

4.5 Step 4: take the weighted average of values

$$\text{Output} = \mathbf{A} \mathbf{V}.$$

Row i of the output is $\sum_j A_{ij} \mathbf{v}_j$.

Key idea

Putting it all together, the formula you will see in every paper is

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

It is called **self-attention** when $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ all come from the same sequence $\mathbf{X}$.

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

If the components of $\mathbf{q}$ and $\mathbf{k}$ are independent with mean 0 and variance 1, then $\mathbf{q} \cdot \mathbf{k}$ has variance d_k , so typical scores grow like $\sqrt{d_k}$. Large scores push softmax toward a nearly one-hot output, where its gradients are almost zero and training stalls. Dividing by $\sqrt{d_k}$ restores unit variance and keeps softmax in a healthy range.

4.7 Worked example: “the angry dog barked”

We now make the ranking table of Part 2 exact. To keep the numbers readable, we skip Step 1 and directly give the vectors, and we follow just *one* query: the one thrown out by *barked*. We use $d_k = 2$ and $d_v = 3$.

- Key coordinates mean (*descriptor / emotion-ness, animate-noun-ness*).
- Value coordinates mean (*emotion content, animal content, sound content*).

Record	key $\mathbf{k}$	value $\mathbf{v}$	meaning
<i>the</i>	(0, 0)	(0, 0, 0)	little content
<i>angry</i>	(1, 0)	(3, 0, 0)	strong emotion
<i>dog</i>	(0, 1)	(0, 2, 1)	an animal that barks
<i>barked</i>	(0, 0)	(0, 0, 2)	a loud sound

The query of *barked* is $\mathbf{q} = (1, 2)$: “I care a little about an emotion (1) and a lot about an animate noun (2).”

Match scores $\mathbf{q} \cdot \mathbf{k}$, then divided by $\sqrt{d_k} = \sqrt{2}$:

	<i>the</i>	<i>angry</i>	<i>dog</i>	<i>barked</i>
$\mathbf{q} \cdot \mathbf{k}$	0	1	2	0
divided by $\sqrt{2}$	0	0.71	1.41	0
$e^{(\cdot)}$	1.00	2.03	4.11	1.00
weight ($\div 8.14$)	0.12	0.25	0.51	0.12

(The four $e^{(\cdot)}$ values add up to 8.14; dividing each by it gives weights that sum to 1.) These are the $\approx 12\%$, 25% , 50% , 12% of the ranking table in Part 2.

Weighted average of the values:

$$0.12 (0, 0, 0) + 0.25 (3, 0, 0) + 0.51 (0, 2, 1) + 0.12 (0, 0, 2) \approx (0.75, 1.01, 0.75).$$

Reading the result. Before attention, *barked* was (0,0,2): pure sound. Afterwards it is (0.75, 1.01, 0.75): it now carries *emotion* (from *angry*), *animal-ness* (from *dog*), and still some *sound*. It has become “an angry dog’s barking.”

Intuition

Notice that the *sound* component dropped from 2 to 0.75: an average dilutes what the word originally had. This is one reason the Transformer adds the output back onto the original vector (the residual connection, Part 6) instead of replacing it.

5 Multi-Head Attention: Several Questions at Once

A single head produces a single set of weights per token, one “question” per token. But a word like *barked* wants to know many things at once: Who did it? How did they feel? At whom was it directed? When? A single softmax would have to compromise between these.

Solution: run h attention heads in parallel, each with its *own* $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$:

$$\text{head}_i = \text{Attention}(\mathbf{X}\mathbf{W}_i^Q, \mathbf{X}\mathbf{W}_i^K, \mathbf{X}\mathbf{W}_i^V),$$

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \mathbf{W}^O.$$

Typically $d_k = d_v = d_{\text{model}}/h$, so the total cost is similar to one big head. $\mathbf{W}^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ mixes the heads’ outputs back together.

Intuition

Think of h different database queries issued simultaneously: one head might look for the doer of the action, another for emotions nearby, another for the previous token, another for a matching pronoun referent. Each head works in its own smaller subspace, and the model learns which questions are worth asking. Different heads do tend to specialize (e.g. syntax, coreference, copying), though not always in a tidy human-readable way.

Parameter count (ignoring biases): the four matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{W}^O$ (each head’s slices together form a $d_{\text{model}} \times d_{\text{model}}$ matrix) give $4d_{\text{model}}^2$ parameters per attention layer.

6 The One-Layer Transformer

Attention alone is not quite enough. A full Transformer *block* adds four more ingredients.

6.1 Positional information

Attention treats its input as a *set*: shuffle the tokens and each token’s output just shuffles with them. “Dog bites man” and “man bites dog” would look the same! In database terms, the records have no “row number” field. We therefore *add a position signal* to each embedding before the first layer. The original paper used fixed sinusoids:

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right).$$

Many models instead learn the position vectors, or encode relative position inside the attention score (e.g. rotary embeddings). The input to the first layer is $\mathbf{X} = \text{embedding} + \text{position}$.

6.2 Feed-forward network (FFN)

After attention, each token passes *independently* through the same small two-layer network:

$$\text{FFN}(\mathbf{z}) = \max(0, \mathbf{z}\mathbf{W}_1 + \mathbf{b}_1) \mathbf{W}_2 + \mathbf{b}_2,$$

with an inner dimension typically $d_{\text{ff}} = 4d_{\text{model}}$ (modern models often use GELU or SwiGLU instead of ReLU).

Intuition

Attention is communication; the FFN is computation. Attention moves information *between* tokens. The FFN then processes what each token has gathered, *without* looking at other tokens. Much of a model’s stored factual knowledge is believed to live in the FFN weights.

If you know MapReduce, the parallel is direct. The FFN is the *map* step: the same function is applied independently to every record, with no communication. Attention is the *shuffle and reduce*: records send their values to wherever the keys say they are relevant, and each destination aggregates what arrives (a weighted sum). One Transformer block is one round of “shuffle, reduce, then map,” and stacking layers chains such rounds together. The difference is that attention is a *soft* shuffle: every value is sent everywhere, scaled by its weight, rather than being routed to a single exact key.

6.3 Residual connections

Each sub-layer’s output is *added* to its input: $\mathbf{x} + \text{Sublayer}(\mathbf{x})$. This means a layer only needs to learn an *update* to the token’s vector rather than rebuild it from scratch, and gradients flow easily through deep stacks. In our example, *barked*’s vector after the layer is “original *barked*” + “what I gathered about the angry dog,” so the original sound information is not lost to averaging.

6.4 Layer normalization

Layer normalization rescales each token’s vector to have zero mean and unit variance across its features (then applies a learned scale and shift). It stabilizes training.

6.5 One complete block

In the original (“post-norm”) arrangement:

$$\mathbf{Z} = \text{LayerNorm}(\mathbf{X} + \text{MultiHead}(\mathbf{X})), \quad \mathbf{Y} = \text{LayerNorm}(\mathbf{Z} + \text{FFN}(\mathbf{Z})).$$

Most modern models use the “pre-norm” variant ($\mathbf{X} + \text{MultiHead}(\text{LayerNorm}(\mathbf{X}))$), which trains more stably when deep.

6.6 A one-layer model, end to end

tokens $\rightarrow$ embeddings + positions $\rightarrow$ one block (MHA $\rightarrow$ FFN) $\rightarrow$ linear layer $\rightarrow$ softmax over vocabulary

For language modeling, the final linear layer maps each token’s vector to one score per vocabulary word, and softmax turns these into next-token probabilities. Training minimizes the cross-entropy between this distribution and the true next token. Gradients flow back through everything, including $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$, which is how the model *learns* what to query, advertise and return.

What a one-layer model can do: it can already do a single round of “who is relevant to me?”: attach adjectives to nouns, find the doer of a verb, find the previous token, copy a recent word.

What it cannot do well: all queries and keys are computed from *raw, context-free embeddings*. A token can only be matched on what its isolated embedding says. Anything that requires matching on *already-contextualized* information is out of reach. This motivates depth.

7 Why Stack Layers?

7.1 The core idea: depth = sequential rounds of lookups

In a stacked Transformer, the output of block ℓ becomes the input of block $\ell + 1$. In layer 2, the queries and keys are computed from vectors that *have already been updated by layer 1*.

Key idea

Width (heads) gives parallel lookups in one round; depth (layers) gives sequential rounds, where each round can build on the results of the previous one. This is the same reason deep neural networks beat shallow ones: composition of simple steps yields complex functions.

7.2 Database view: multi-hop queries

Imagine our database where *after each round of queries, the records are rewritten with what was retrieved*. We saw this already: after one round, the record for *barked* has its “who?” and

“emotion” fields filled in. In the next round, other words can query these *updated* records. This is like a database join: first find the related records, then query using the information you just found.

7.3 Example: building up meaning

Take “The old red car broke down.”

- **Layer 1:** *car* gathers from *old* and *red*. Its vector now means roughly “old red car” (a noun phrase), and *The* attaches to it as well.
- **Layer 2:** *broke* looks for its subject. It can now match against *car*’s *enriched* key (“a complete noun phrase that can be a subject”), which did not exist in layer 0.
- **Layer 3 and beyond:** *down* pairs with *broke* to form a phrasal verb; the clause as a whole becomes available for even higher-level questions (sentiment, topic, what will come next).

A similar story applies to pronouns, as in “The trophy didn’t fit in the suitcase because **it** was too big.” Resolving *it* requires first representing what *big* means for the sentence, and then using that to select the right referent: at least two dependent steps.

7.4 The residual stream as a shared workspace

A popular mental model: all tokens carry a vector (the *residual stream*). Each attention layer *reads* from the stream, moves information between tokens, and *writes* the result back by addition. Each FFN reads, computes, and writes back. After N layers, the stream contains a rich, context-dependent representation of each token.

7.5 Typical sizes

Model	N layers	d_{model}	h heads	d_{ff}	d_k
Original Transformer (base)	6 + 6	512	8	2048	64
Modern large language models	dozens	thousands	dozens	$\approx 4 d_{\text{model}}$	64–128

8 The Full Transformer Architecture

8.1 Encoder block

Exactly the block of Part 6: self-attention, then FFN, each with residual connection and layer norm. In the encoder, every token may attend to *every* token (bidirectional). An encoder stack of N blocks turns an input sequence into contextual representations.

8.2 Causal (masked) self-attention

When generating text left to right, token i must not peek at future tokens (that would be cheating during training). In database terms, a word may only query the *previous* records. We add a mask $\mathbf{M}$ to the scores before softmax:

$$\text{Attention} = \text{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}} + \mathbf{M}\right) \mathbf{V}, \quad M_{ij} = \begin{cases} 0 & j \leq i \\ -\infty & j > i. \end{cases}$$

After softmax, $e^{-\infty} = 0$, so future positions get zero weight. (In “the angry dog barked”, a causal *angry* could not see *dog*, but *barked* can see all three earlier words.)

8.3 Cross-attention

In an encoder-decoder model (e.g. for translation), the decoder has an extra attention sub-layer where the **queries come from the decoder** but the **keys and values come from the encoder’s output**. In database terms: the decoder asks questions of the stored “database” of the source sentence.

8.4 Decoder block

Each decoder block has three sub-layers, each with residual connection and layer norm:

1. masked self-attention (look at previously generated tokens),
2. cross-attention to the encoder output,
3. feed-forward network.

8.5 Overall picture

Encoder-decoder (original Transformer, translation)

```
-----  
source tokens -> embed + position -> [Encoder block] x N --+  
                                         | (keys, values)  
target so far -> embed + position -> [Decoder block] x N <-+  
                                         (masked self-attn,  
                                         cross-attn, FFN)  
                                         -> linear -> softmax -> next token
```

Decoder-only (GPT-style, most modern LLMs)

```
-----  
tokens -> embed + position -> [masked self-attn + FFN] x N  
                                         -> linear -> softmax -> next token
```

Most modern large language models are *decoder-only*: the same stack of masked self-attention blocks, trained to predict the next token.

8.6 Training in one paragraph

Training minimizes cross-entropy loss between the predicted next-token distribution and the actual next token, summed over all positions (thanks to the causal mask, all positions of a sequence can be trained in parallel in a single forward pass). All parameters (embeddings, every $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, $\mathbf{W}^O$, FFN weights, normalization parameters, output layer) are learned jointly by backpropagation. The roles we call query, key and value emerge purely because that division of labor reduces the loss.

8.7 A cost to know about

The score matrix $\mathbf{S}$ is $n \times n$, so attention costs $O(n^2)$ time and memory in sequence length n . Doubling the context length roughly quadruples the attention cost, which is why long-context models require special engineering. On the other hand, all positions are processed in parallel, which suits GPUs far better than step-by-step recurrent networks.

9 Where the Analogy Breaks Down, and Summary

9.1 Honest limits of the database analogy

- **Soft vs. hard.** A database returns the one matching record. Attention returns a *weighted blend of all records*.
- **Learned vs. designed.** Database attributes are human-defined (“emotion”, “size”, “doer”). Attention’s queries, keys and values are *learned vectors* whose meaning is not guaranteed to be human-readable. Our attribute lists are a story about what directions in vector space might encode.
- **One vector, not a field list.** A token has a single key vector and a single value vector, not a named list of fields with some left empty.
- **Every token is both reader and record.** In self-attention, the same token issues a query *and* is stored as a key–value record.
- **The database rewrites itself.** After each layer, the records have changed. A real database does not do that, but this is exactly what makes depth powerful.
- **Many queries at once.** Multi-head attention issues several different queries per token in parallel.

9.2 One-page summary

Concept	One-line meaning
Record	A word's vector: what it is and knows, with gaps to be filled from context
Query $\mathbf{q}$	What this token is looking for
Key $\mathbf{k}$	What this token advertises about itself (used for matching)
Value $\mathbf{v}$	What this token contributes if selected (used for content)
Attention weights	Softmax of query-key similarity: how much to read from each token
Output	Weighted average of values: a context-aware new vector
Multi-head	Several independent lookups in parallel, then combined
Positional encoding	Tells the model token order (attention alone ignores it)
FFN	Per-token computation on what attention gathered
Residual + LayerNorm	Layers learn updates; training stays stable
Stacking layers	Sequential rounds of lookups; later rounds build on earlier ones
Causal mask	Prevents looking at future tokens during generation
Cross-attention	Queries from decoder; keys and values from encoder

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