

The Self-Attention Algorithm

1. Simpler Terms: "classroom" story

Imagine a classroom where every word in a sentence is a student.

Each student (word) has three cards: **Q**, **K**, and **V**.

Step 1: Three Cards (Asking and telling)

The sentence is: "Alice will eat pizza."

The goal is to figure out who is doing the eating (the subject) and what they are eating (the object) — but without any grammar rules, just by looking at how words relate to each other.

Query (Q)

what the word is *looking for*: It's the *question* the word asks others.

Example:

1. "eat" Q card might ask, "Who is doing the eating?"
2. "Alice" Q card might ask, "What action am I doing?"
3. "pizza" Q card might ask, "What action is happening to me?"

Key (K)

What the word *offers*: It's the *label* that tells others what kind of word it is.

Example:

1. "Alice" K card says, "I'm a subject/person who can do actions"
2. "eat" K card says, "I'm an action that needs a subject and object."
3. "pizza" K card says, "I'm an object that can be acted upon."

Value (V)

The *actual information* the word can share if chosen.

Example:

- "Alice" V card gives, "Alice = eater."
- "eat" V card gives, "eat = action."
- "pizza" V card gives, "pizza = food."

Step 2: Listening around the class

Every word looks at all the others and asks:

"How much do your Keys match what I'm looking for (my Query)?"

It scores everyone.

- If another word's Key fits the Query well → strong connection.
- If not → weak or zero connection.

Mathematically, this "match" is a **dot product** between Query and Key — a simple numeric similarity test.

Step 3: Combining answers

Each word then *listens* to everyone — but pays **more attention** to those with higher scores.

It takes a weighted mix of their **Values (V)**.

So, "eat" might listen mostly to "Alice's" Value because she matched the idea of "person."

Step 4: Updated knowledge

After gathering information, "eat" now knows *who* is doing the eating — because it listened more to the most relevant word's Value.

In short:

Card	Purpose	Example ("eat" and "Alice")
Query (Q)	What am I looking for?	"Who is the eater?"
Key (K)	What do I represent?	"I'm a person."
Value (V)	What info can I share?	"Alice = eater"
Attention	How well Q matches K	Strong between "eat" and "Alice"

It's basically a **conversation of meaning** where every word decides which other words to pay **attention** to — guided by Questions (Q), IDs (K), and shared Information (V).

Now, you understand the meaning of **attention!**

2. The Math: From Story to Formulas

Step 1: Every word becomes a **vector**

First, each word is converted into a **vector** — a list of numbers that captures the word's meaning.

For our sentence "**Alice will eat pizza**", each word gets an **embedding vector**.

Alice → \vec{x}_1 , will → \vec{x}_2 , eat → \vec{x}_3 , pizza → \vec{x}_4

We stack them into a matrix X :

$$X = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \end{bmatrix} \quad \leftarrow \text{each row is one word's vector}$$

Step 2: Make the three cards (Q , K , V)

Each word makes its Q , K , V cards by multiplying its vector by three learned weight matrices:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

Symbol	Meaning	Story equivalent
W_Q	Learned "what to ask"	How to form a Question
W_K	Learned "what to advertise"	How to form a Key label
W_V	Learned "what to share"	How to form a Value

Every word in X gets its own Q, K, V row — **all in one matrix multiply.**

Step 3: Score the match (dot product)

To check how well "**eat**"'s **Query** matches "**Alice**"'s **Key**, we compute a **dot product**:

$$\text{score}(\text{eat}, \text{Alice}) = \vec{q}_{\text{eat}} \cdot \vec{k}_{\text{Alice}}$$

For all word pairs at once:

$$\text{Scores} = QK^T$$

If two words are semantically related (subject and verb), the dot product is **large**. If unrelated, it is **small**.

Example scores for "eat" (row 3):

$$\text{Scores}_3 = [\underbrace{8.1}_{\text{Alice}}, \ 1.2, \ 3.0, \ \underbrace{6.5}_{\text{pizza}}]$$

Step 4: Scale and Normalize

Big dot products can grow very large and cause problems in training.

We **scale** by $\sqrt{d_k}$ (where d_k = size of the Key vector):

$$\text{Scaled Scores} = \frac{QK^T}{\sqrt{d_k}}$$

Then apply **softmax** to convert scores into probabilities (weights that sum to 1):

$$\text{Attention Weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Example for "eat" after softmax:

$$\alpha_3 = [0.70, \underbrace{0.03, \text{ Alice}}_{0.07}, \underbrace{0.20}_{\text{pizza}}]$$

"eat" pays 70% **attention** to "Alice" and 20% **attention** to "pizza" — exactly what the story said!

Step 5: Collect the answers (weighted sum)

Each word gathers information by taking a **weighted sum of the Values**:

$$\text{output}_{\text{eat}} = \sum_j \alpha_{3,j} \cdot \vec{v}_j$$
$$= 0.70 \cdot \vec{v}_{\text{Alice}} + 0.03 \cdot \vec{v}_{\text{will}} + 0.07 \cdot \vec{v}_{\text{eat}} + 0.20 \cdot \vec{v}_{\text{pizza}}$$

"eat" has now **absorbed** information mostly from "Alice" (the subject) and a bit from "pizza" (the object).

For all words at once:

The Full Formula (One Line)

Combining all steps:

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

Step	Operation	Story equivalent
1	$X \rightarrow Q, K, V$	Every student makes 3 cards
2	QK^T	Everyone compares their Q to others' K
3	$\div \sqrt{d_k}$	Prevent scores from growing too large
4	softmax	Convert scores to attention weights (sum to 1)
5	$\times V$	Collect info, weighted by attention

Visualizing Attention Weights

The attention weight matrix for "**Alice will eat pizza**" might look like:

	Alice	will	eat	pizza
Alice	0.10	0.05	0.75	0.10
will	0.15	0.60	0.15	0.10
eat	0.70	0.03	0.07	0.20
pizza	0.10	0.05	0.65	0.20

- "**eat**" to "**Alice**" (0.70): verb finds its subject
- "**Alice**" to "**eat**" (0.75): subject finds its verb
- "**pizza**" to "**eat**" (0.65): object finds its action

Summary: Story to Math

Story	Math
Word = student	\vec{x}_i (embedding vector)
Making 3 cards	$Q = XW_Q, K = XW_K, V = XW_V$
Comparing Q to K	QK^T (dot product)
Scoring the match	$\frac{QK^T}{\sqrt{d_k}}$ (scaled)
Deciding how much to listen	$\text{softmax}(\dots)$
Collecting answers	$\times V$ (weighted sum)
Updated knowledge	Output vector per word

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

3. Training vs. Inference

A common confusion

"Does the model learn that 'Alice eats pizza' during training?

Or does it figure it out when we type the sentence?"

Both — but at **different times, doing completely different things.**

	Training	Inference
When	Before you use the model	When you type a sentence
What changes	W_Q, W_K, W_V are updated	Nothing — weights are frozen
What is learned	General patterns (verb and subject)	Nothing — just arithmetic
Input	Billions of sentences	Your one sentence

Story: Learning vs. Using

Think back to the classroom story.

Training is like the students going through school for years:

- They read millions of books
- A teacher corrects their mistakes every day
- Slowly, they build **instincts** — "verbs go with subjects", "objects follow verbs"
- The instincts are stored in their brain — i.e., in W_Q , W_K , W_V

Inference is like using those instincts on an exam:

- School is over — no more learning
- The student reads "Alice will eat pizza" for the first time
- The trained brain instantly connects "eat" and "Alice" without any grammar lookup

What is actually stored after training?

The model does **not** store sentences. It stores **weight matrices**:

$$W_Q, \quad W_K, \quad W_V$$

These matrices encode **general linguistic patterns** learned from billions of sentences:

- Verbs tend to attend to nearby subject nouns
- Pronouns refer back to earlier nouns
- Objects follow the verbs that act on them

When a new sentence arrives, the model uses these frozen matrices to **compute** relationships on the fly —

Training: making and correcting mistakes

During training, the model sees a sentence and tries to **predict** the next word.

Example: given "Alice will eat __", the correct answer is "**pizza**".

If it predicts wrong, calculate the **error (loss)**:

$$\mathcal{L} = -\log P(\text{pizza} \mid \text{Alice will eat})$$

Then **backpropagate** the error to nudge the weights:

$$W_Q \leftarrow W_Q - \eta \cdot \frac{\partial \mathcal{L}}{\partial W_Q}$$

η = learning rate (step size), \mathcal{L} = loss (how wrong the answer was)

Repeat for billions of sentences – W_Q, W_K, W_V gradually encode real language structure.

Training: what the weights learn

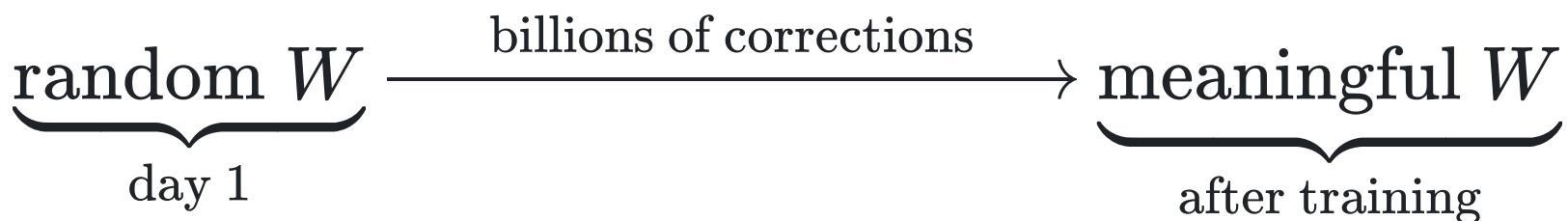
After seeing millions of sentences:

- "**Alice** will **eat** pizza" — eat should attend to Alice
- "**Bob** will **drink** coffee" — drink should attend to Bob
- "**The dog** will **chase** the ball" — chase should attend to dog

The matrices learn a general rule — **without anyone writing it**:

Verbs should ask: "Who is the subject?"

Subject nouns should answer: "I can perform actions."



Warning!

W_Q is NOT per word — it is shared across all words.

There is **one** W_Q matrix for the whole model (per head, per layer). Every word — "Alice", "eat", "pizza" — passes through the **same** W_Q .

The reason each word gets a **different** query vector is simply that their **input embeddings are different**:

```
Alice [1.0, 0.5, 0.2, ...] × W_Q → q_alice (one direction)
eat   [0.2, 0.8, 0.9, ...] × W_Q → q_eat   (different direction)
pizza [0.9, 0.1, 0.3, ...] × W_Q → q_pizza (another direction)
```

W_Q is a **lens** — every word passes through it, but each word's unique embedding comes out pointing in a different direction.

How big is Alice, eat, pizza?

This connects directly to d_{model} .

- The "Alice" vector = the word embedding
- When "Alice" enters the model, it is looked up in an **embedding table** and converted to a vector of size d_{model} .

Embedding table

That vector **is** the Alice vector — the same d_{model} we saw in W_Q .

Model	Alice vector size
GPT-2 small	768 numbers
GPT-3	12,288 numbers
LLaMA 2 7B	4,096 numbers

So in GPT-2 small, "Alice" is represented as a list of **768 floating point numbers**.

There is one big **embedding table** – a matrix of shape:

Embedding table = vocabulary size $\times d_{\text{model}}$

For GPT-2 small:

50,257 words \times 768 dimensions = ~38 million numbers

"Alice" has a row in this table. That row is its vector. The values are **learned during training** – just like W_Q and W_K .

How big is W_Q ?

$$W_Q \text{ shape} = d_{\text{model}} \times d_k$$

Model	d_{model}	d_k (per head)	One W_Q size
GPT-2 small	768	64	$768 \times 64 = 49,152$ numbers
GPT-3	12,288	128	$12,288 \times 128 = 1,572,864$ numbers
LLaMA 2 7B	4,096	128	$4,096 \times 128 = 524,288$ numbers

For GPT-2 small, one single W_Q matrix holds about **50,000 numbers.**

It multiplies — heads × layers

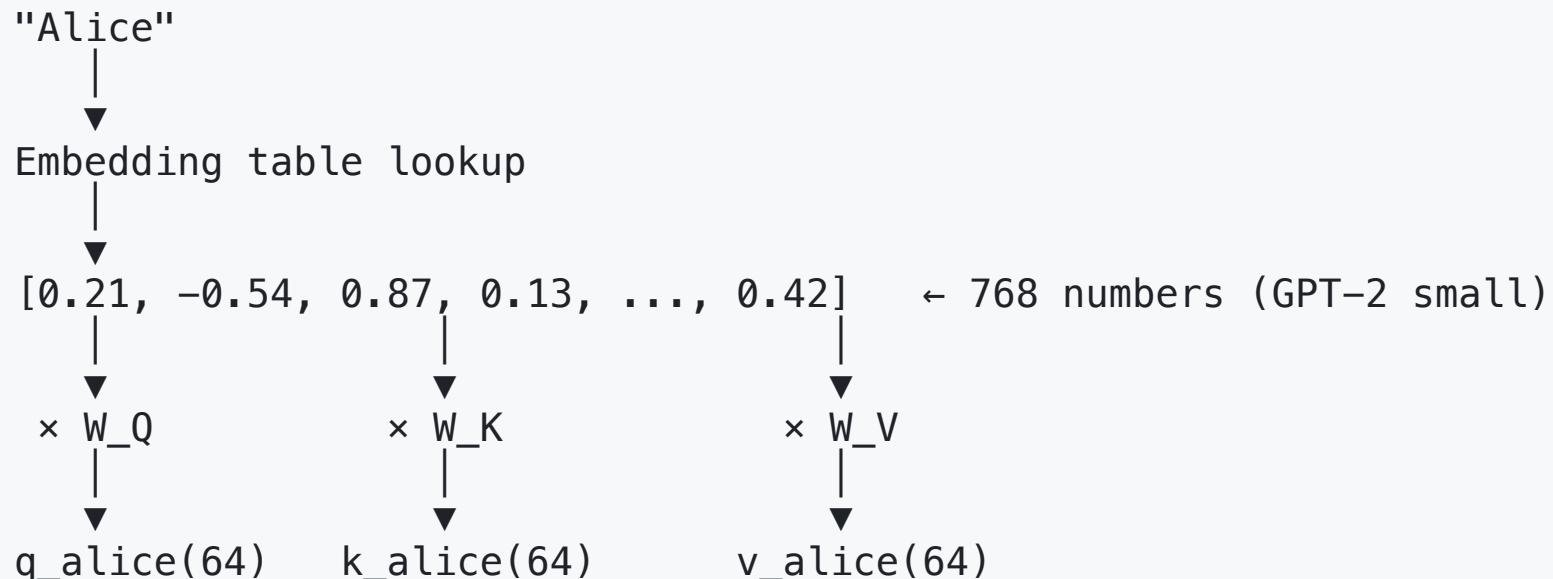
In practice W_Q exists once per **attention head, per layer.**

GPT-2 small has 12 heads and 12 layers:

12 layers × 12 heads × 49,152 = ~7 million numbers (just for W_Q)
 W_K and W_V are the same size → ~21 million numbers total

GPT-3 (96 layers, 96 heads) scales this up dramatically — which is why it has 175 billion parameters total.

The full picture for "Alice"



The 768-dimensional Alice vector gets **projected down** to 64 dimensions for each of Q, K, V — that is what W_Q (768×64) does.

Why so many dimensions?

Each dimension captures some aspect of "Alice-ness" — but not in a human-readable way. After training, dimensions might loosely encode things like:

- "person vs. object"
- "animate vs. inanimate"
- "proper noun vs. common noun"
- ...hundreds of other subtle features

No single dimension has a clean label. The meaning is **distributed** across all 768 numbers together — this is why they are called **distributed representations**.

Summary

Thing	Size (GPT-2 small)
"Alice" embedding vector	768 numbers
After $\times W_Q \rightarrow$ query vector	64 numbers
After $\times W_K \rightarrow$ key vector	64 numbers
After $\times W_V \rightarrow$ value vector	64 numbers

"Alice" starts as 768 numbers.

W_Q compresses it to 64 numbers that answer:

"what is Alice looking for?"

W_K compresses it to 64 numbers that answer:

"what does Alice offer?"

Training Example

What W_Q and W_K physically are

They are just **matrices of numbers** — like a spreadsheet of weights.

- They don't store "Alice" or "pizza" directly.
- What they store is a **transformation recipe**.

Remember that each word is represented by a vector \vec{x}_i .

- When you multiply a word's embedding by W_Q , you get that word's **query vector** — a direction in space that says "what kind of thing am I looking for?"
- When you multiply by W_K , you get the **key vector** — a direction that says "what kind of thing am I?"

They are random at first, but training rotates them so that related words end up pointing in similar directions.

Concrete example — before training

Say "eat" has embedding $[0.2, 0.8, 0.9]$ and W_Q is random garbage (this is just an example, the real numbers are way bigger):

```
w_Q (random) = [[0.3, -0.1],  
                 [-0.2, 0.7],  
                 [0.4, 0.2]]
```

```
q_eat = [0.2, 0.8, 0.9] × w_Q = [0.30, 0.61] ← random direction  
k_alice = [1.0, 0.5, 0.2] × w_K = [0.21, 0.44] ← random direction
```

```
score = q_eat · k_alice = 0.30×0.21 + 0.61×0.44 = 0.33  
score(eat, pizza) = 0.32 ← almost the same!
```

Softmax gives nearly **equal attention** to Alice and pizza. The model doesn't know which matters.

What training actually does to W_Q and W_K

The model sees "**Alice will eat __**" and predicts "pizza" (wrong at first – maybe it says "Alice").

Backpropagation traces back through the attention formula:

"The reason I predicted wrong is that 'eat' wasn't **attending** strongly enough to 'Alice' (the subject). I need to adjust W_Q and W_K so that verbs and subject-nouns produce a **higher dot product**."

After this one correction, W_Q and W_K shift slightly.

After **millions of sentences** with the same pattern:

- "Bob will **drink coffee**"
- "She is **running fast**"
- "The dog will **chase the ball**"

...the matrices gradually rotate so that **verb query vectors** and **subject noun key vectors** point in similar directions in space.

After training – what W_Q encodes

Think of W_Q as encoding the question "**what role do I play?**"

After training, when "eat" is multiplied by W_Q , the result is a vector pointing in the "**verb looking for subject**" direction.

When "Alice" is multiplied by W_K , the result points in the "**I am a subject/person**" direction.

These two directions are now **aligned** — so their dot product is large.

```
Before training: q_eat points randomly, k_alice points randomly → low score  
After training: q_eat points "verb→", k_alice points " $\leftarrow$ subject" → HIGH score
```

Why "Alice" beats "pizza"

After training:

```
q_eat → direction meaning "I'm a verb, looking for who does this action"  
k_alice → direction meaning "I'm a person who can perform actions"  
k_pizza → direction meaning "I'm an object that gets acted upon"
```

The dot product $q_{\text{eat}} \cdot k_{\text{alice}}$ is large because these two directions **align** — verb-query meets subject-key.

The dot product $q_{\text{eat}} \cdot k_{\text{pizza}}$ is smaller because verb-query and object-key **don't align as well** for this particular question.

The key insight

W_Q and W_K don't store facts like "Alice is a subject."

They store **geometric transformations** that, when applied to word embeddings, produce vectors whose **similarity reflects linguistic relationships**.

Training teaches the matrices to **rotate and stretch** the embedding space so that words that should attend to each other end up **pointing in similar directions**.

Inference: using the frozen weights

Now you type: "**Alice will eat pizza**" — possibly a sentence never seen before.

The weights are **frozen (the same)**. The model just computes:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

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

Because W_Q and W_K were trained to connect verbs to subjects,

$\vec{q}_{\text{eat}} \cdot \vec{k}_{\text{Alice}}$ comes out **high** — attention weight becomes **0.70**.

The 0.70 is **computed** from trained weights, not looked up from memory.

This is why we need a powerful GPU to run inference — it has to do all these matrix multiplications and softmax calculations on the fly.

Why this matters: brand-new sentences

Because the model learned **general patterns**, not specific sentences,
it handles sentences it has **never seen before**:

- "**Zara will devour a croissant**"
- "**The robot will assemble the car**"
- "**My cat will ignore me again**"

	Search engine	LLM
Knows "Alice eats pizza"	Only if indexed	Not stored
Handles a new sentence	No	Yes
What it uses	Index lookup	Trained weight matrices

Summary

Question	Answer
Does the model memorize "Alice eats pizza"?	No
What does training produce?	W_Q, W_K, W_V encoding general patterns
When you type a sentence, does it learn?	No — weights are frozen
Where does the 0.70 come from?	Computed on the fly using trained weights
Can it understand brand-new sentences?	Yes — it learned patterns, not facts

Self-attention is a **skill**, not a **memory**.

1. Now, you understand the meaning of the LLM **model**, what is your interpretation of the meaning of it from the LLM **weights**?
2. Why it is so expensive and time-consuming to train an LLM? What is the meaning of the training process?
3. Why do we need expensive GPUs to run inference on LLMs? What is the meaning of the inference process?