DAY 2

How a Transformer Computes the Next Token

From tokenization to decoding, the full forward pass

ROADMAP

Workshop Roadmap

DAY 1 — COMPLETED

Why LLMs Behave as They Do

- → Inference vs. training
- → Next-token prediction
- → Loss → gradients → updates
- → Why this objective works
- → Data, scale, and alignment

¡ Day 1: the model as a **system shaped by training.**

DAY 2 — TODAY

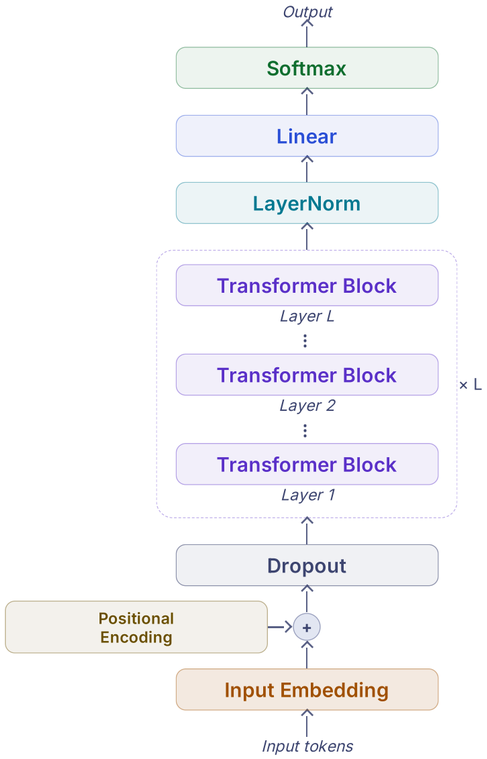
How a Transformer Processes Text and Predicts the Next Token

- → Tokenization & embeddings
- → Self-attention
- → Feed-forward layers
- → Full forward pass end-to-end

¡ Day 2: the model as a **machine executing computation.**

ARCHITECTURE

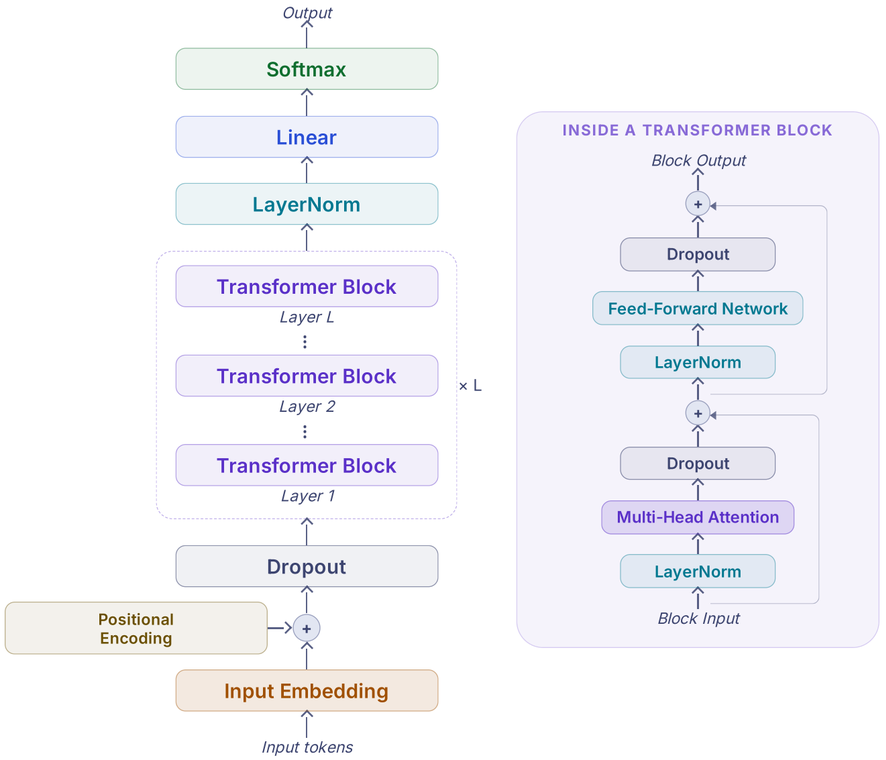
Decoder-Only Transformer: High-Level View



¡ Each block writes into the **residual stream**, which accumulates across all layers.

ARCHITECTURE

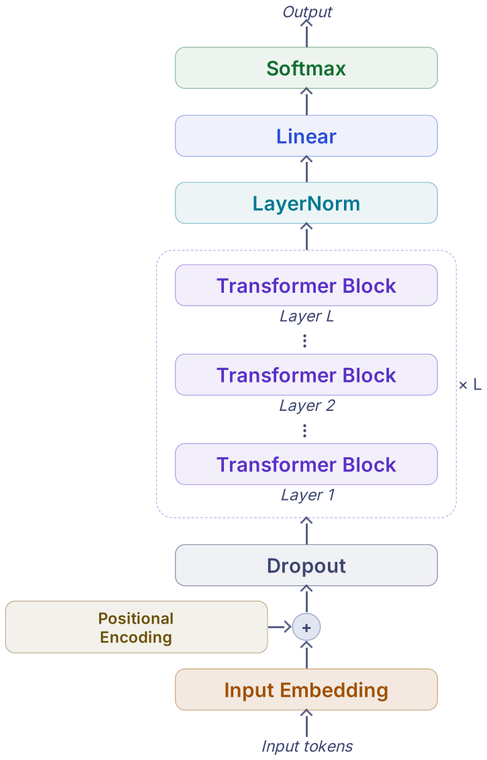
Decoder-Only Transformer: High-Level View

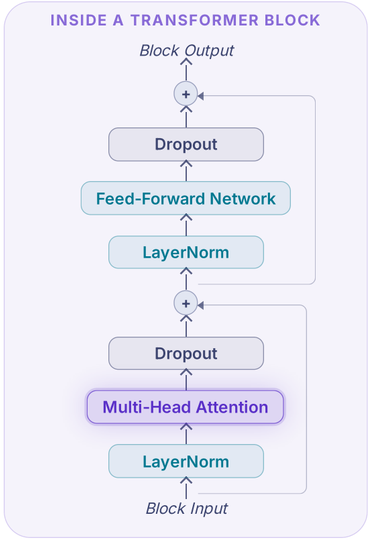


¡ Each block writes into the **residual stream**, which accumulates across all layers.

ARCHITECTURE

Decoder-Only Transformer: High-Level View





SELF-ATTENTION IN ONE LINE

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

The core operation behind language modeling, reasoning, and generation.

¡ Each block writes into the **residual stream**, which accumulates across all layers.

How Many r's Are in "Strawberry"?

TOKENIZATION

strawberry

You see: **3 r's**

Model answer: **2 r's**

What the Model Sees

YOU TYPE

s t r a w b e r r y

MODEL SEES

[straw] [berry]

The model gets **2 token vectors**, not 10 letter vectors. Letter counting requires reconstructing characters from subword pieces.

¡ This is **structural**, not a bug. The next slides show where the limitation comes from.

TOKENIZATION

Before Reading, the Model Tokenizes

Raw text → Tokenizer → Token IDs → Embeddings

Three Steps

+ Define a fixed vocabulary of discrete symbols.
+ Split input text into pieces from that vocabulary.
+ Map each piece to an integer ID.

¡ The vocabulary is learned once, then **frozen**. GPT-2 uses 50,257 tokens; GPT-4-class tokenizers are roughly 100k.

TOKENIZATION

Before Reading, the Model Tokenizes

Raw text → Tokenizer → Token IDs → Embeddings

Three Steps

+ Define a fixed vocabulary of discrete symbols.
+ Split input text into pieces from that vocabulary.
+ Map each piece to an integer ID.

¡ The vocabulary is learned once, then **frozen**. GPT-2 uses 50,257 tokens; GPT-4-class tokenizers are roughly 100k.

Running Example

Used in the next three slides.

SENTENCE

```
"researchers fine-tune large-scale models"
```

SUBWORD TOKENIZATION

`[research]` `[ers]` `[fine]` `[-]` `[tune]` `[large]` `[-]` `[scale]` `[model]` `[s]`

```
[3245, 891, 4680, 12, 10812, 3267, 12, 5649, 219, 82]
```

IDs shown are illustrative; exact values depend on the tokenizer.

WORD-LEVEL

Approach 1: Word-Level Tokenization

Running Sentence: Word-Level Tokenization

If every word is in the vocabulary:

`[researchers]` `[fine-tune]` `[large-scale]` `[models]`

But compounds like **fine-tune** and **large-scale** are often missing, so they collapse to unknown tokens:

`[researchers]` `[UNK]` `[UNK]` `[models]`

WORD-LEVEL

Approach 1: Word-Level Tokenization

Running Sentence: Word-Level Tokenization

If every word is in the vocabulary:

`[researchers]` `[fine-tune]` `[large-scale]` `[models]`

But compounds like **fine-tune** and **large-scale** are often missing, so they collapse to unknown tokens:

`[researchers]` `[UNK]` `[UNK]` `[models]`

Strengths

+ Short sequences for familiar text.
+ Common words stay intact as single units.

Limits

- Language is open-vocabulary. English Wikipedia alone has ~1.5M unique word forms.
- New terms like "GPT-4", "LoRA", and "RLHF" create unknowns.

WORD-LEVEL

Approach 1: Word-Level Tokenization

Running Sentence: Word-Level Tokenization

If every word is in the vocabulary:

`[researchers]` `[fine-tune]` `[large-scale]` `[models]`

But compounds like **fine-tune** and **large-scale** are often missing, so they collapse to unknown tokens:

`[researchers]` `[UNK]` `[UNK]` `[models]`

Strengths

+ Short sequences for familiar text.
+ Common words stay intact as single units.

Limits

- Language is open-vocabulary. English Wikipedia alone has ~1.5M unique word forms.
- New terms like "GPT-4", "LoRA", and "RLHF" create unknowns.

! **Efficient, but brittle.** Anything outside the fixed vocabulary becomes unknown.

CHARACTER-LEVEL

Approach 2: Character-Level Tokenization

Running Sentence: Character-Level Tokenization

[r] [e] [s] [e] [a] [r] [c] [h] [e] [r] [s] [ ] [f] [i] [n] [e] [-] [t] [u] [n]
[e] [ ] [l] [a] [r] [g] [e] [-] [s] [c] [a] [l] [e] [ ] [m] [o] [d] [e] [l] [s]

WORD-LEVEL (BEST CASE)

4 tokens

CHARACTER-LEVEL (SAME SENTENCE)

~40 tokens

SCALING: 500-WORD PASSAGE

~2,500 character tokens

CHARACTER-LEVEL

Approach 2: Character-Level Tokenization

Running Sentence: Character-Level Tokenization

[r] [e] [s] [e] [a] [r] [c] [h] [e] [r] [s] [ ] [f] [i] [n] [e] [-] [t] [u] [n]
[e] [ ] [l] [a] [r] [g] [e] [-] [s] [c] [a] [l] [e] [ ] [m] [o] [d] [e] [l] [s]

WORD-LEVEL (BEST CASE)

4 tokens

CHARACTER-LEVEL (SAME SENTENCE)

~40 tokens

SCALING: 500-WORD PASSAGE

~2,500 character tokens

Strengths

+ No unknown tokens — ever.
+ Handles new words, typos, code, any language.

Limits

- Sequences become $5–10\times$ longer.
- Self-attention is $O(n^2)$, so $5\times$ longer means roughly $25\times$ more compute.
- The model spends capacity reassembling words from letters on every forward pass.

CHARACTER-LEVEL

Approach 2: Character-Level Tokenization

Running Sentence: Character-Level Tokenization

[r] [e] [s] [e] [a] [r] [c] [h] [e] [r] [s] [ ] [f] [i] [n] [e] [-] [t] [u] [n]
[e] [ ] [l] [a] [r] [g] [e] [-] [s] [c] [a] [l] [e] [ ] [m] [o] [d] [e] [l] [s]

WORD-LEVEL (BEST CASE)	CHARACTER-LEVEL (SAME SENTENCE)	SCALING: 500-WORD PASSAGE
4 tokens	~40 tokens	~2,500 character tokens

Strengths

+ No unknown tokens — ever.
+ Handles new words, typos, code, any language.

Limits

- Sequences become $5–10\times$ longer.
- Self-attention is $O(n^2)$, so $5\times$ longer means roughly $25\times$ more compute.
- The model spends capacity reassembling words from letters on every forward pass.

! **Perfect coverage, impractical cost.** Quadratic attention scaling makes it too expensive at realistic sequence lengths.

SUBWORD STRATEGY

Approach 3: Subword Tokenization, the Practical Default

Running Sentence: Subword Tokenization

`[research]` `[ers]` `[fine]` `[-]` `[tune]` `[large]` `[-]` `[scale]` `[model]` `[s]`

Frequent stems become single tokens. Hyphens, prefixes, and suffixes become reusable pieces, so words stay decomposable rather than unknown.

SUBWORD STRATEGY

Approach 3: Subword Tokenization, the Practical Default

Running Sentence: Subword Tokenization

`[research]` `[ers]` `[fine]` `[-]` `[tune]` `[large]` `[-]` `[scale]` `[model]` `[s]`

Frequent stems become single tokens. Hyphens, prefixes, and suffixes become reusable pieces, so words stay decomposable rather than unknown.

Word-level

`4 tokens`

Short, but brittle. Breaks on hyphens, new jargon, and multilingual text.

Character-level

`~40 tokens`

Full coverage, but very long. Quadratic attention cost makes it impractical.

Subword

`~10 tokens`

Near-word efficiency with no unknown tokens. English prose averages about 0.75 words per token.

SUBWORD STRATEGY

Approach 3: Subword Tokenization, the Practical Default

Running Sentence: Subword Tokenization

`[research]` `[ers]` `[fine]` `[-]` `[tune]` `[large]` `[-]` `[scale]` `[model]` `[s]`

Frequent stems become single tokens. Hyphens, prefixes, and suffixes become reusable pieces, so words stay decomposable rather than unknown.

Word-level

`4 tokens`

Short, but brittle. Breaks on hyphens, new jargon, and multilingual text.

Character-level

`~40 tokens`

Full coverage, but very long. Quadratic attention cost makes it impractical.

Subword

`~10 tokens`

Near-word efficiency with no unknown tokens. English prose averages about 0.75 words per token.

\+ **Near-word efficiency with broad coverage.** That is why subword tokenization became the modern default.

SUBWORD STRATEGY

Approach 3: Subword Tokenization, the Practical Default

Running Sentence: Subword Tokenization

`[research]` `[ers]` `[fine]` `[-]` `[tune]` `[large]` `[-]` `[scale]` `[model]` `[s]`

Frequent stems become single tokens. Hyphens, prefixes, and suffixes become reusable pieces, so words stay decomposable rather than unknown.

Word-level

`4 tokens`

Short, but brittle. Breaks on hyphens, new jargon, and multilingual text.

Character-level

`~40 tokens`

Full coverage, but very long. Quadratic attention cost makes it impractical.

Subword

`~10 tokens`

Near-word efficiency with no unknown tokens. English prose averages about 0.75 words per token.

\+ **Near-word efficiency with broad coverage.** That is why subword tokenization became the modern default.

¡ One tradeoff remains: characters inside a token are no longer individually visible. This matters again at the

How the Tokenizer Learns These Pieces

We start with a **character-level vocabulary**. Now let the corpus teach us which adjacent pairs deserve their own tokens.

Start: characters → **Count adjacent pairs** → **Merge most frequent** → **Repeat N times**

Core Rule

Last slide, the fallback was character-level text. BPE starts there too: count adjacent pairs in the corpus, merge the most frequent pair, and repeat. Those learned merges become the vocabulary.

Why BPE Works

BPE is driven by frequency, not grammar. That is why stems and endings like **-ing** and **-ed** can emerge naturally when they keep showing up together.

¡ **BPE is a frequency-based compression algorithm.** It builds the vocabulary the data demands, not the one a linguist would design.

? Watch BPE build up from characters on a tiny action corpus.

BPE WALKTHROUGH

BPE on a tiny action corpus

CORPUS STATE

START FROM A CHARACTER-LEVEL VOCABULARY: EVERY WORD IS JUST LETTERS (</W> MARKS WORD END)

```
h u g </w>
h u g s </w>
h u g g e d </w>
h u g g i n g </w>
s m i l e d </w>
s m i l i n g </w>
w a v e d </w>
w a v i n g </w>
```

Start at characters, count adjacent pairs, and let repetition decide what gets promoted.

VOCABULARY BUILT SO FAR

BASE VOCABULARY

`h` `u` `g` `s` `e` `d` `i` `n` `m` `l` `w` `a` `v`

`h u → 4` `u g → 4`

`i n → 3` `e d → 3`

There is a tie at the top. We will follow the **hug** stem first, then come back for reusable endings.

BPE WALKTHROUGH

BPE on a tiny action corpus

CORPUS STATE

MERGE 1: **H + U → HU** (FREQ: 4)

```
hu g </w>
hu g s </w>
hu g g e d </w>
hu g g i n g </w>
s m i l e d </w>
s m i l i n g </w>
w a v e d </w>
w a v i n g </w>
```

VOCABULARY BUILT SO FAR

MERGED CHUNKS SO FAR

`hu`

`hu g → 4`	`i n → 3`
`e d → 3`	`n g → 3`

Same corpus, one level up: **hu + g** is still the most common pair.

BPE WALKTHROUGH

BPE on a tiny action corpus

CORPUS STATE

MERGE 2: **HU + G → HUG** (FREQ: 4)

```
hug </w>
hug s </w>
hug g e d </w>
hug g i n g </w>
s m i l e d </w>
s m i l i n g </w>
w a v e d </w>
w a v i n g </w>
```

VOCABULARY BUILT SO FAR

MERGED CHUNKS SO FAR

`hu` `hug`

`i n → 3`

`e d → 3`

`n g → 3`

`hug g → 2`

The stem is built. Next we let shared endings compete.

BPE WALKTHROUGH

BPE on a tiny action corpus

CORPUS STATE

MERGE 3: **I + N → IN** (FREQ: 3)

```
hug </w>
hug s </w>
hug g e d </w>
hug g in g </w>
s m i l e d </w>
s m i l in g </w>
w a v e d </w>
w a v in g </w>
```

VOCABULARY BUILT SO FAR

MERGED CHUNKS SO FAR

`hu` `hug` `in`

`in g → 3` `e d → 3`

`hug g → 2` `l in → 1`

One more merge turns a recurring ending into a reusable chunk.

BPE WALKTHROUGH

BPE on a tiny action corpus

CORPUS STATE

MERGE 4: **IN + G → ING** (FREQ: 3)

```
hug </w>
hug s </w>
hug g e d </w>
hug g ing </w>
s m i l e d </w>
s m i l ing </w>
w a v e d </w>
w a v ing </w>
```

VOCABULARY BUILT SO FAR

MERGED CHUNKS SO FAR

`hu` `hug` `in` `ing`

`e d → 3`

`hug g → 2`

`l ing → 1`

`v ing → 1`

! Aha: **ing** just became reusable. Three different words now share the same ending token.

BPE WALKTHROUGH

BPE on a tiny action corpus

CORPUS STATE

MERGE 5: **E + D → ED** (FREQ: 3)

```
hug </w>
hug s </w>
hug g ed </w>
hug g ing </w>
s m i l ed </w>
s m i l ing </w>
w a v ed </w>
w a v ing </w>
```

VOCABULARY BUILT SO FAR

REUSABLE ENDINGS UNLOCKED

`hu` `hug` `in` `ing` `ed`

! Now **ed** joins the vocabulary too. The tokenizer is learning reusable endings, not memorizing whole words.

BPE TAKEAWAY

BPE builds a vocabulary of reusable chunks

Vocabulary After These Merges

BASE CHARACTERS

`h` `u` `g` `s` `e` `d` `i` `n` `m` `l` `w` `a` `v`

STEM-BUILDING CHUNKS

`hu` `hug` `in`

REUSABLE SUFFIX CHUNKS

`ing` `ed`

Gray = base characters · Blue = stem-building merges · Green = **ing** · Orange = **ed**

Reusable Chunks in Action

hugs	→	`hug` `s`
hugging	→	`hug` `g` `ing`
hugged	→	`hug` `g` `ed`
smiling	→	`s` `m` `i` `l` `ing`
waved	→	`w` `a` `v` `ed`

¡ The tokenizer has now learned reusable endings like **ing** and **ed**, so common word forms can reuse chunks instead of being rebuilt from scratch every time.

\+ **BPE starts from characters and promotes repeated pairs into reusable chunks.** That is how pieces like **hug**, **ing**, and **ed** enter the vocabulary.

WHY IT MATTERS

Why Tokenization Shapes Reasoning

Core Mechanism

The model computes on **token vectors**, not character vectors. When one token covers many letters, character-level reasoning must be reconstructed from subword pieces.

"strawberry" → `[straw]` `[berry]` 2 token vectors. No separate letter vectors.

Cosma et al., *"The Strawberry Problem: Emergence of Character-level Understanding in Tokenized Language Models"*, EMNLP 2025.

Where It Breaks

- Counting specific letters in a word.
- Extracting the nth character from a string.
- Detecting repeated characters.
- Reversing a string exactly.

Why It Works So Well

+ Subword pieces let the model handle **unseen words** by composing familiar parts instead of needing every word in the vocabulary.
+ Sequences stay much **shorter than character-level**, which makes attention cheaper and preserves more context.
+ The vocabulary stays much **smaller than word-level**, reducing the size of the embedding and output layers.
+ That tradeoff is why tokenization is so widely used: it is efficient, flexible, and good enough for most language understanding tasks.

From token IDs to vectors

A token ID is simply a row index in a learned embedding matrix.

$$T \in \{1, \ldots, |V|\}^n, \quad X^{(0)} = E[T] \in \mathbb{R}^{n \times d_{\text{model}}}$$

Token IDs: [812, 1452, 5021] → **Row lookup in E** → **Output: $x_1, x_2, x_3 \in \mathbb{R}^{d_{\text{model}}}$**

Three Retrieved Rows

⋮	⋯				
$E[812, :]$	"river"	+0.14	-0.77	+0.32	...
$E[1452, :]$	"bank"	+0.11	-0.70	+0.28	...
$E[5021, :]$	"loan"	-0.62	+0.04	+0.80	...
⋮	⋯				

Scale of the Full Table

$$E \in \mathbb{R}^{|V| \times d_{\text{model}}}$$

Examples:
GPT-2 small: $|V| = 50{,}257, \quad d_{\text{model}} = 768$
$|E| = 50{,}257 \times 768 = 38{,}597{,}376$

GPT-4 tokenizer: $|V| \approx 100{,}000$
Exact d_{model} is not public, but the table still scales as $|V| \times d_{\text{model}}$

KEY PROPERTIES

What Embeddings Provide, and What They Still Lack

What Embeddings Already Provide

+ Learned table $E \in \mathbb{R}^{|V| \times d_{\text{model}}}$: one trainable row per token type.
+ Often one of the largest parameter blocks, scaling with $|V| \times d_{\text{model}}$.
+ Outputs $X^{(0)}$ already match model width — later blocks consume them directly.
+ Useful structure appears as directions, neighborhoods, and relative offsets.

What Later Layers Must Add

- Sense is unresolved: one row can support multiple intended meanings.
- No token-token interaction yet — rows cannot use neighboring words.
- Row identity alone does not specify sentence role or dependency structure.
- Later layers must refine these vectors into context-dependent states.

→ Different projections reveal different relationships; context resolves the intended meaning.

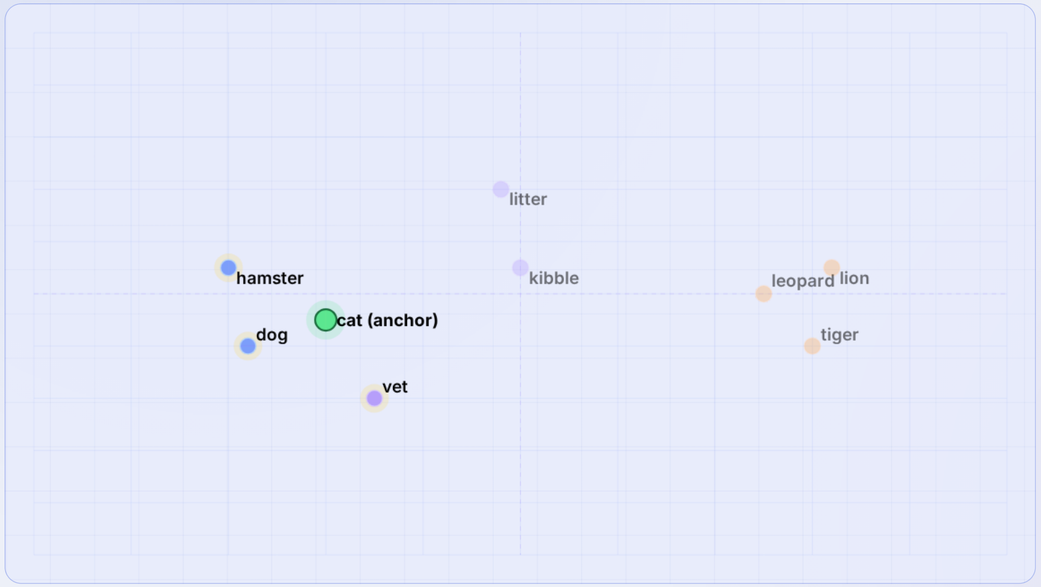
GEOMETRY

One Embedding Space, Multiple Semantic Projections

Each token is one high-dimensional vector. These plots are different 2D views of the same space.

Change the Lens, Not the Space

Pet / companionship | Feline family | Care context



Active lens: **Pet / companionship**

This view emphasizes: **pet and companionship features**

Nearest to **cat** in this view:

`dog` `hamster` `vet`

$$z^{(\text{lens})} = P_{\text{lens}} x,\ P_{\text{lens}} \in \mathbb{R}^{2 \times d_{\text{model}}}$$

The underlying vectors do not move. Only the 2D projection changes, revealing different relationships.

GEOMETRY

One Embedding Space, Multiple Semantic Projections

Each token is one high-dimensional vector. These plots are different 2D views of the same space.

Change the Lens, Not the Space

Pet / companionship | Feline family | Care context

Active lens: **Feline family**

This view emphasizes: **feline-family features**

Nearest to **cat** in this view:

`leopard` `tiger` `lion`

$$z^{(\text{lens})} = P_{\text{lens}} x,\ P_{\text{lens}} \in \mathbb{R}^{2 \times d_{\text{model}}}$$

The underlying vectors do not move. Only the 2D projection changes, revealing different relationships.

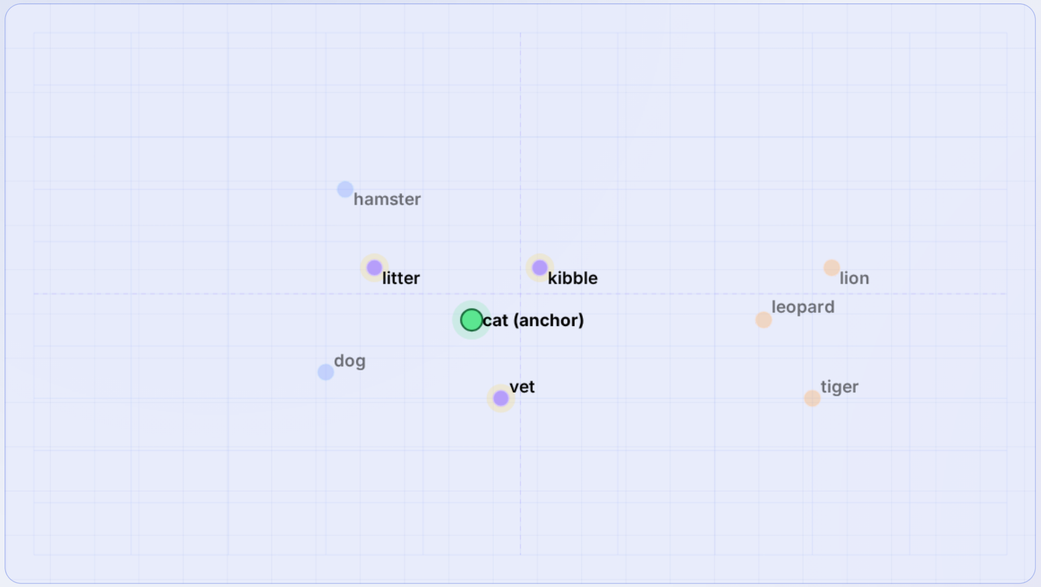
GEOMETRY

One Embedding Space, Multiple Semantic Projections

Each token is one high-dimensional vector. These plots are different 2D views of the same space.

Change the Lens, Not the Space

Pet / companionship | Feline family | **Care context**



Active lens: **Care context**

This view emphasizes: **care and ownership features**

Nearest to **cat** in this view:

`litter` `kibble` `vet`

$$z^{(\text{lens})} = P_{\text{lens}} x,\ P_{\text{lens}} \in \mathbb{R}^{2 \times d_{\text{model}}}$$

The underlying vectors do not move. Only the 2D projection changes, revealing different relationships.

BRIDGE

TRANSITION

Why embeddings are not enough

Same Row, Different Meaning

```
The fisherman sat on the bank of the river.
```

```
She applied for a loan at the bank.
```

$$h_{\text{bank}}^{(0)} = E[\text{bank}]$$

At input, both sentences start from the same **bank** embedding.

+ The row tells us which token this is.
+ The neighboring words tell us which sense is intended.
+ Attention is the mechanism that mixes in those neighbors.

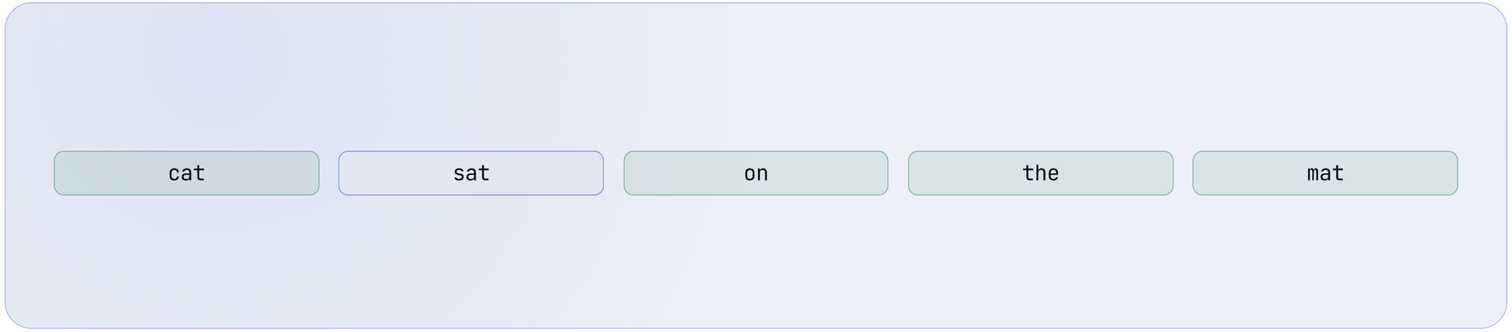
→ Attention lets a token read from surrounding words.

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.

Embeddings are static; attention makes them context-aware.

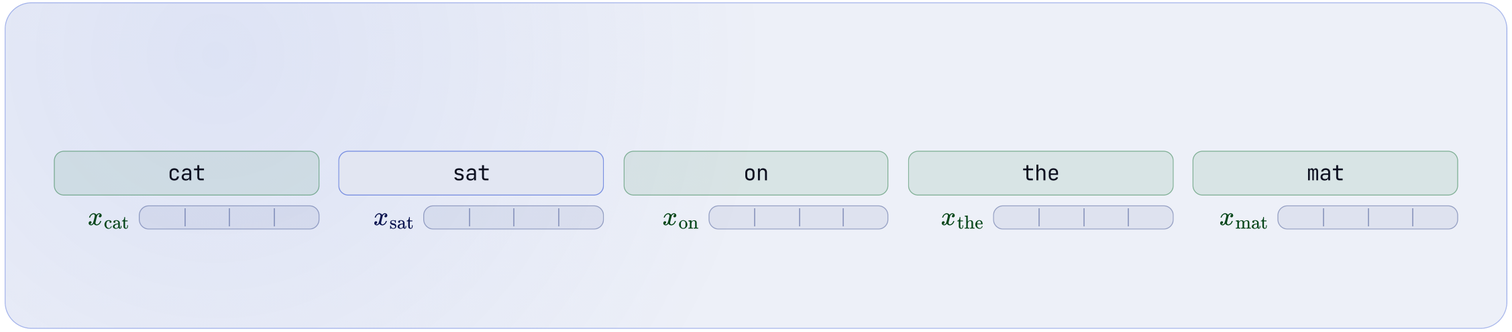


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.

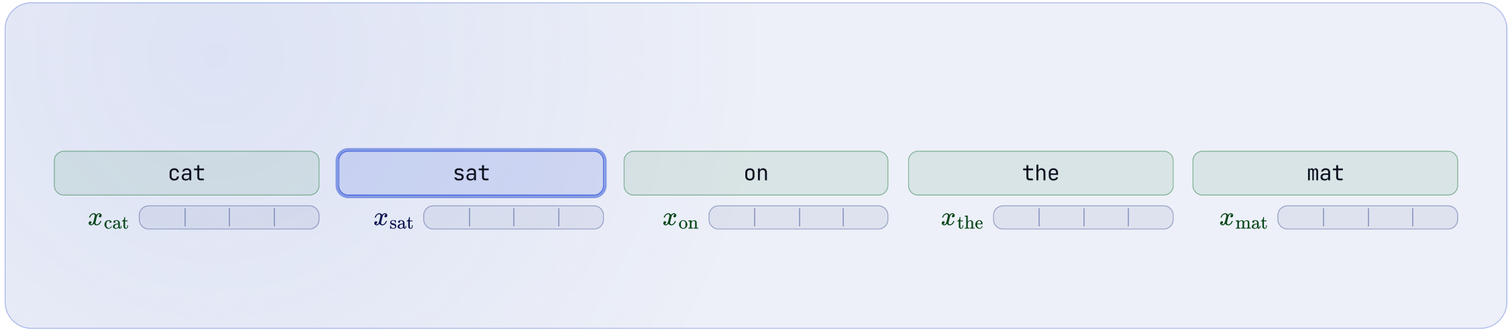


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.

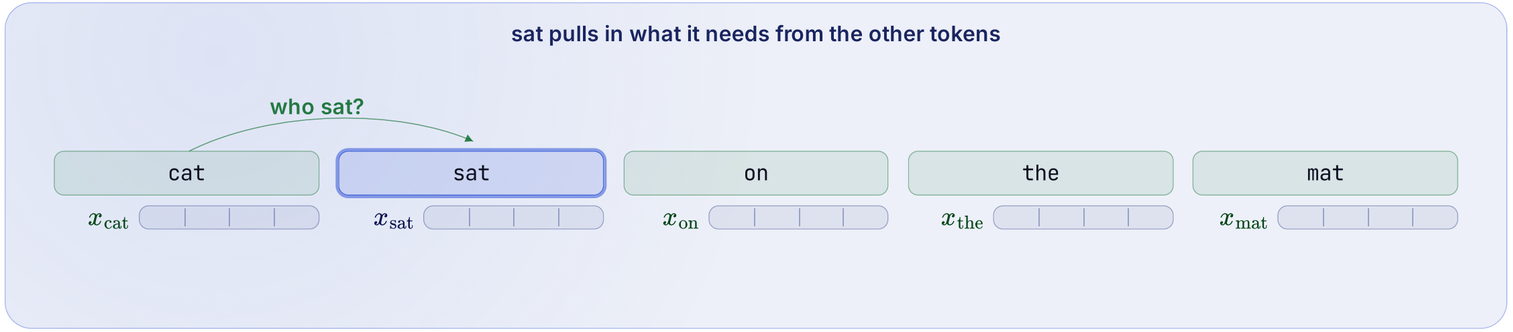


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.

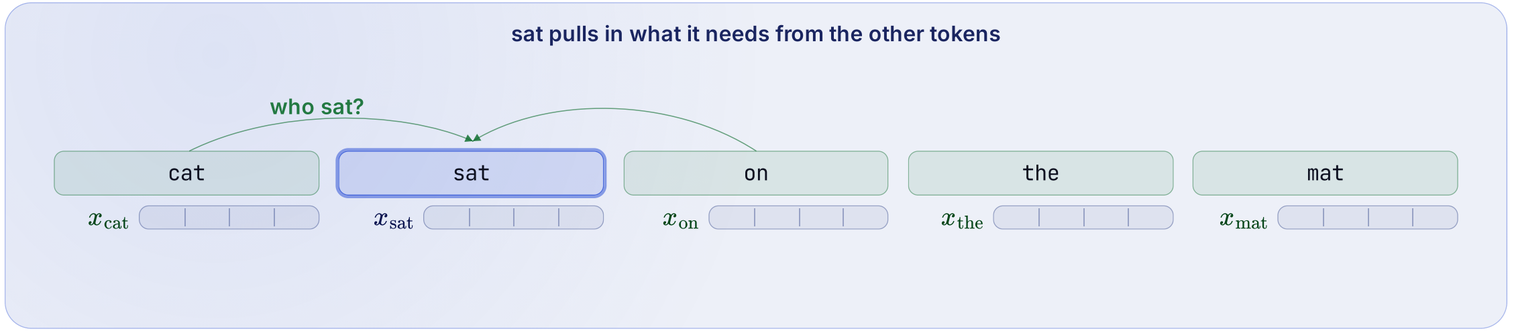


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.

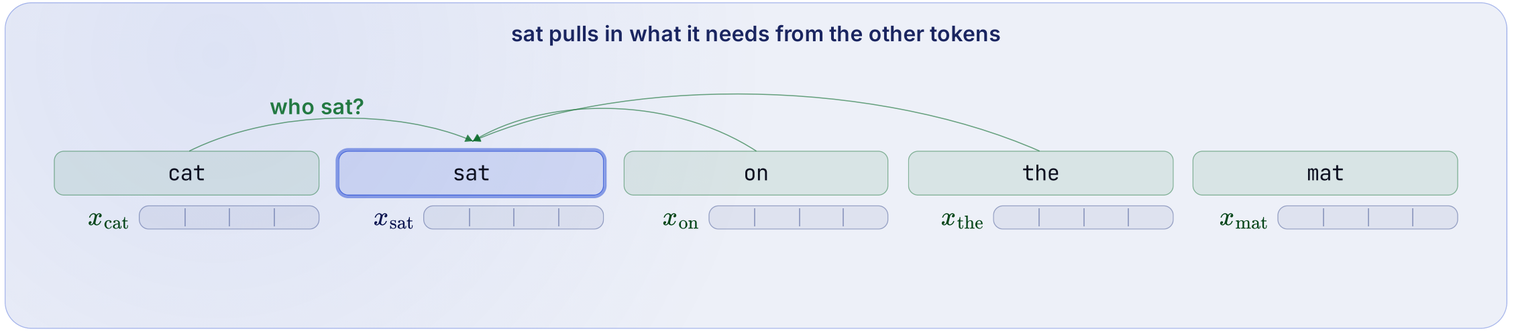


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.

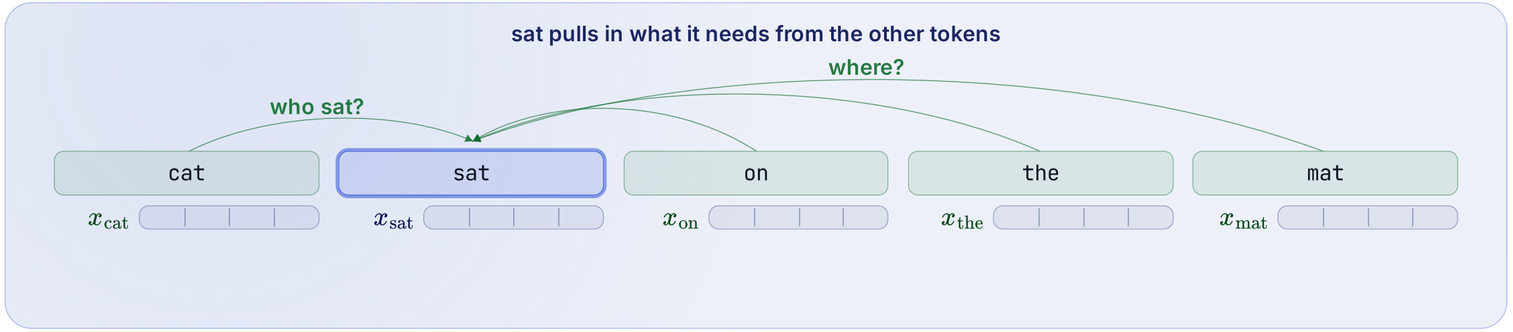


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.

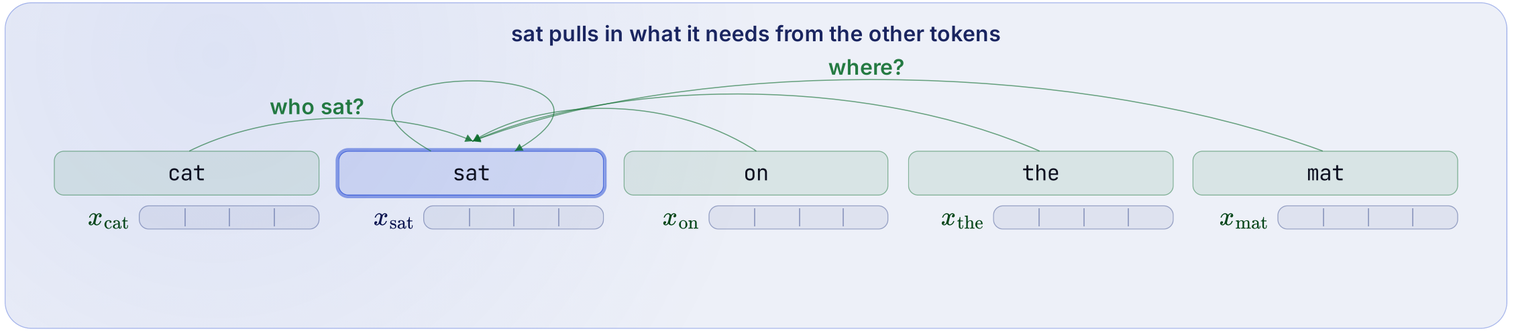


Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.
Embeddings are static; attention makes them context-aware.



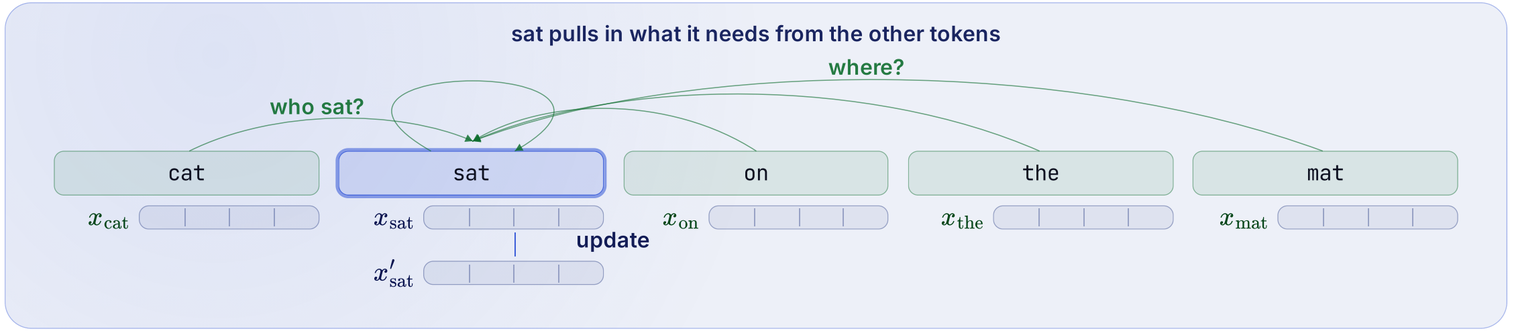
Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Attention lets tokens read from each other

Each token updates its representation using the rest of the sequence.

Embeddings are static; attention makes them context-aware.



Attention now needs a scoring rule: what matters, and what gets copied forward?

MECHANISM

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

Attention creates three role-specific versions of each token state.

MECHANISM

Step 1 — Create Q, K, and V for each token

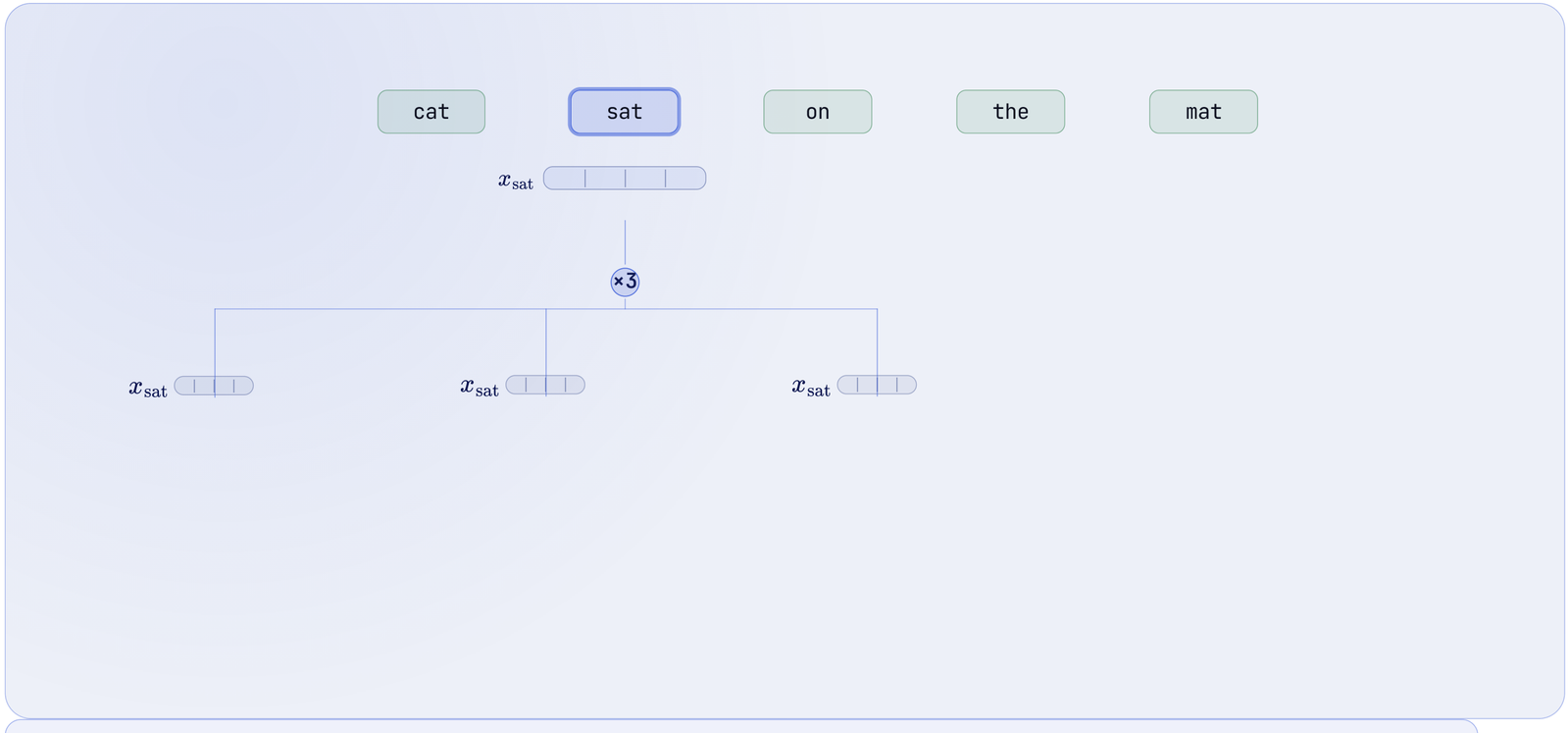
Project the same token state into three learned subspaces.

Start from the current token state for sat : x_{sat} .

MECHANISM

Step 1 — Create Q, K, and V for each token

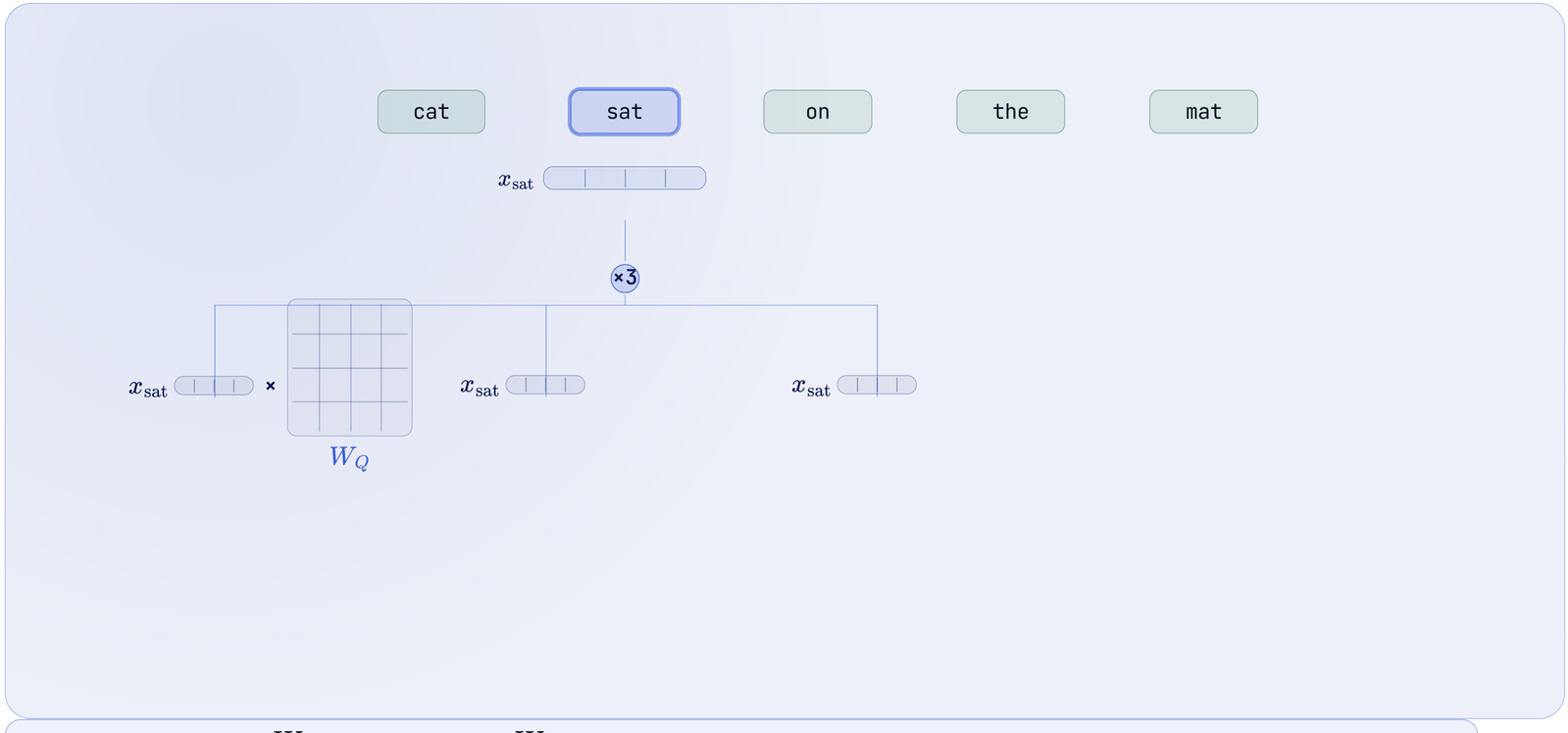
Project the same token state into three learned subspaces.



Copy x_{sat} into three branches: one each for Q, K, and V.

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

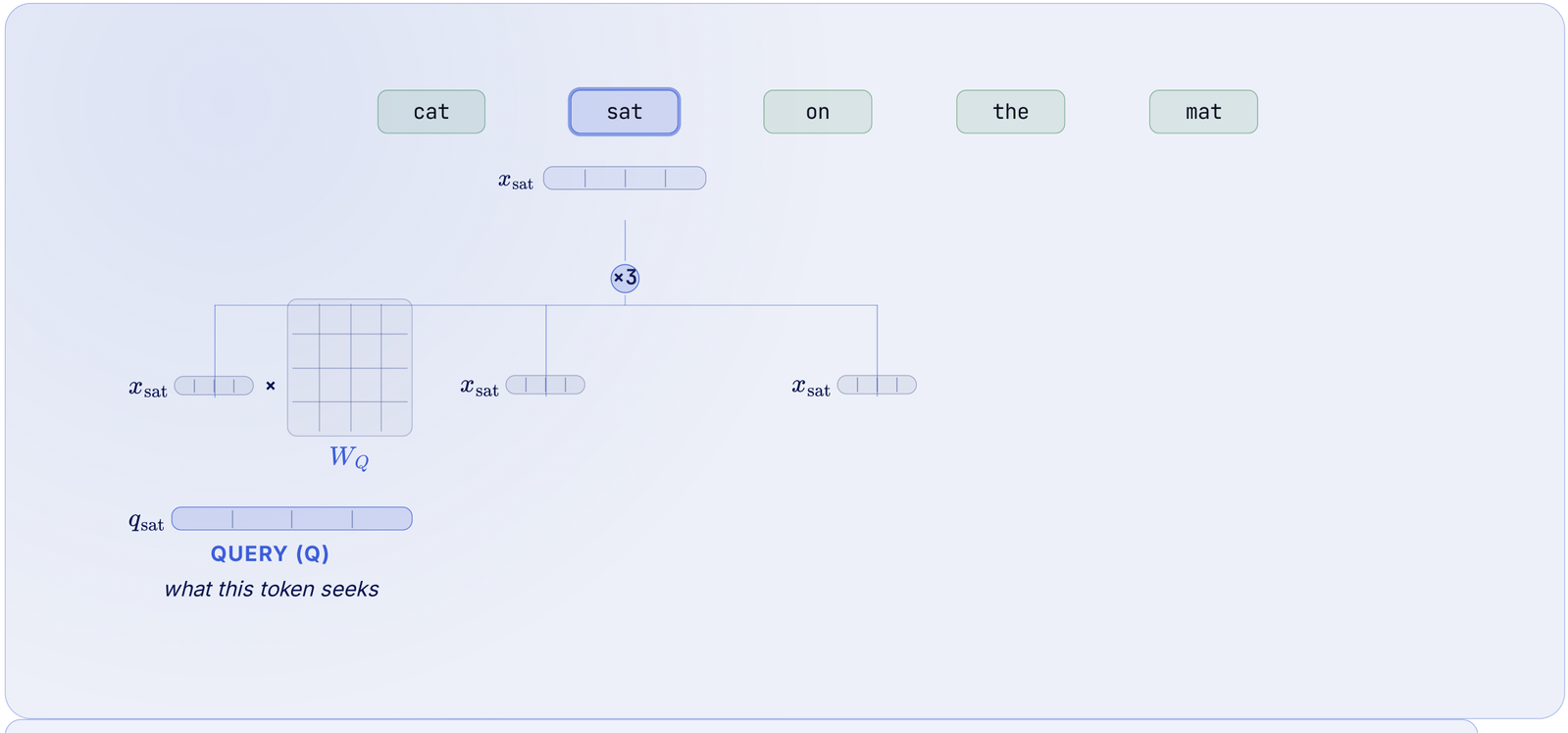


Apply the query matrix W_Q to get $q_{\text{sat}} = x_{\text{sat}} W_Q$.

MECHANISM

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

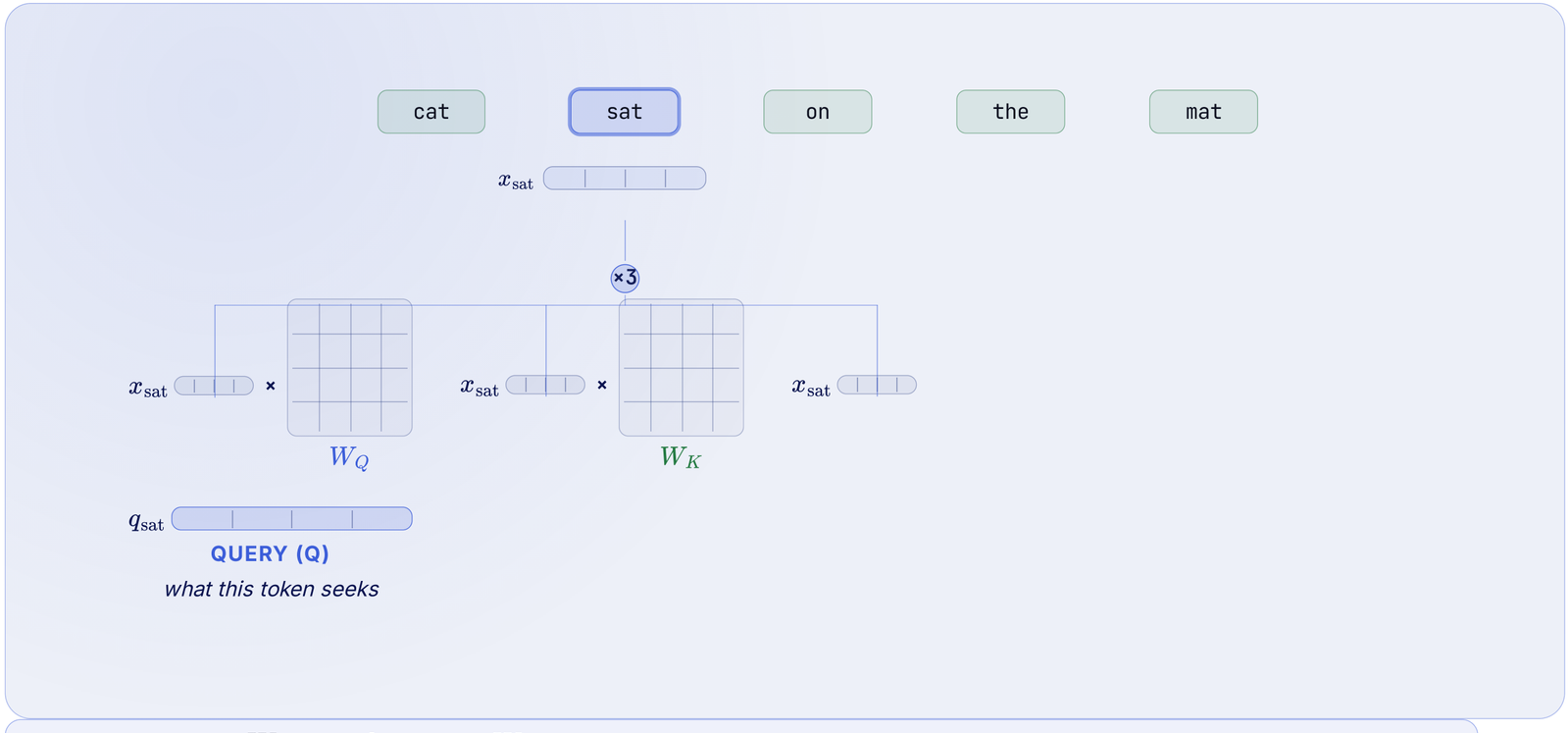


Query q_{sat} : what this token seeks.

MECHANISM

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

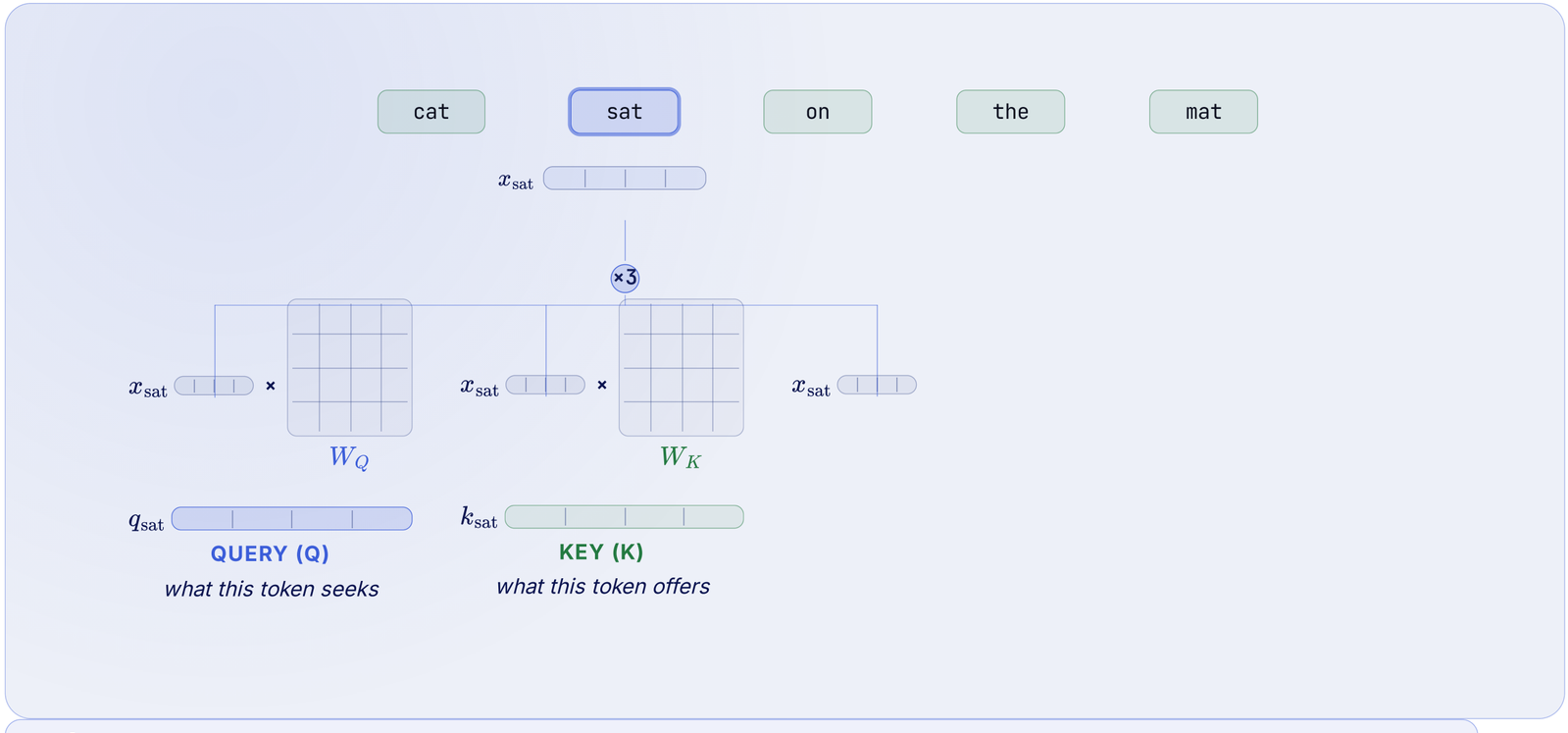


Apply the key matrix W_K to get $k_{\text{sat}} = x_{\text{sat}} W_K$.

MECHANISM

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

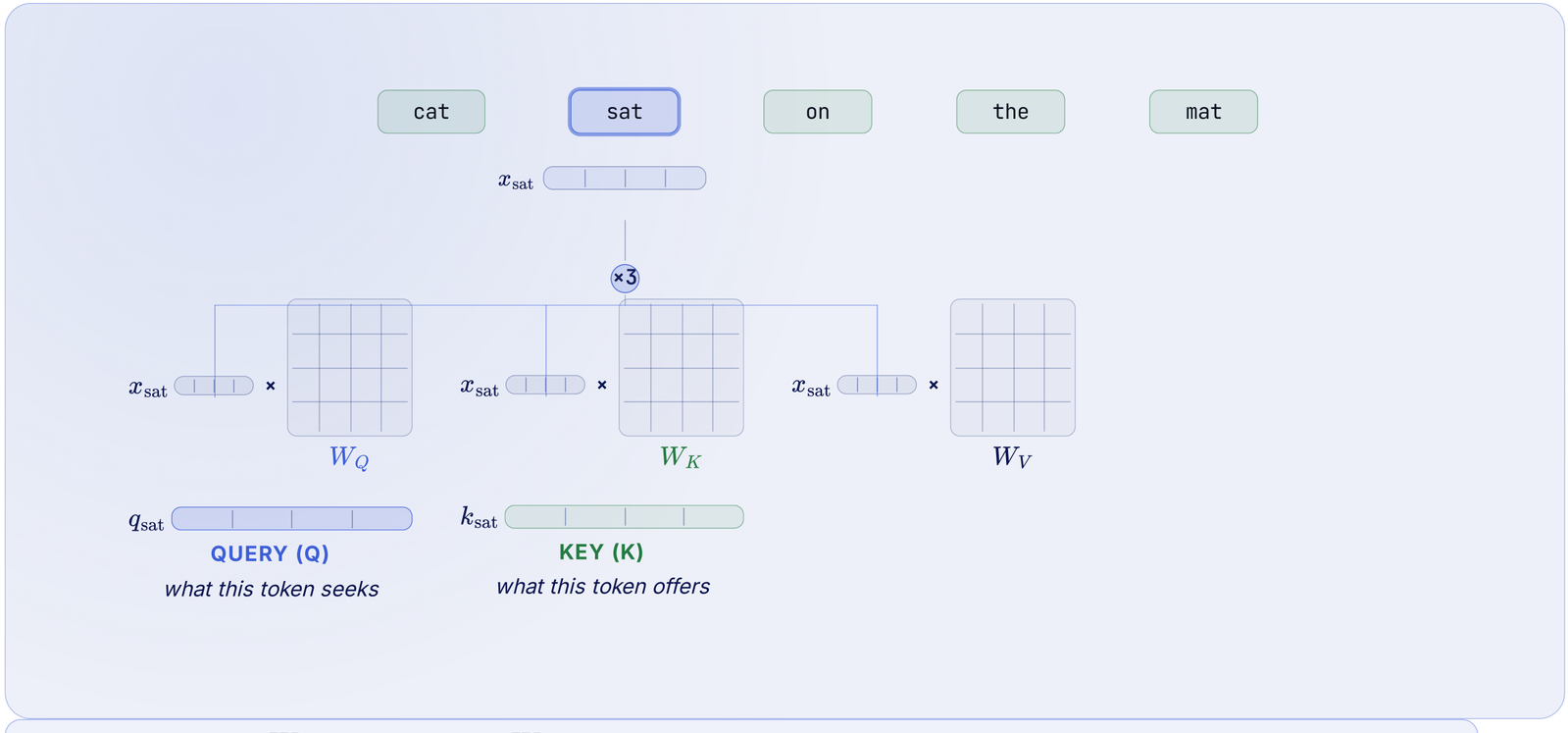


Key k_{sat} : what this token offers other tokens.

MECHANISM

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

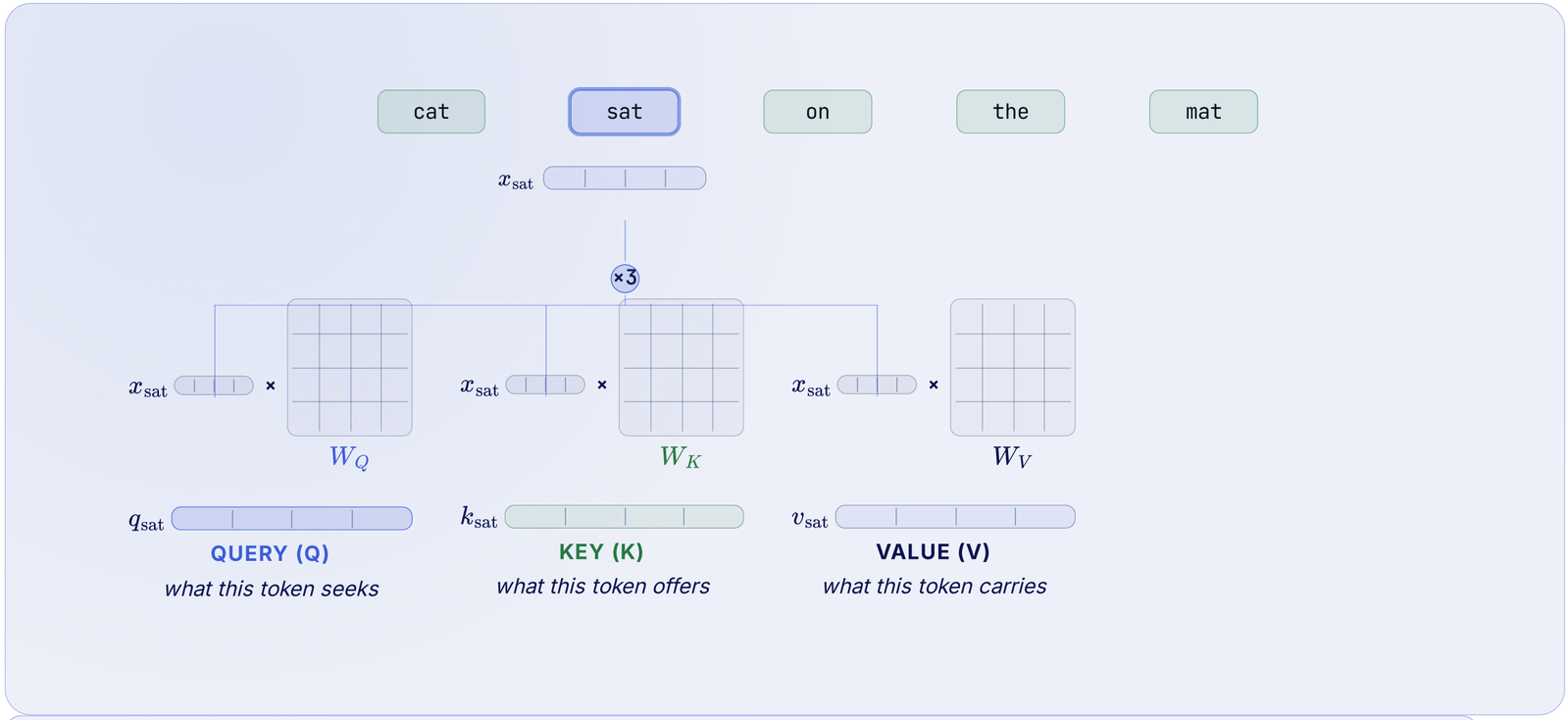


Apply the value matrix W_V to get $v_{\text{sat}} = x_{\text{sat}} W_V$.

MECHANISM

Step 1 — Create Q, K, and V for each token

Project the same token state into three learned subspaces.

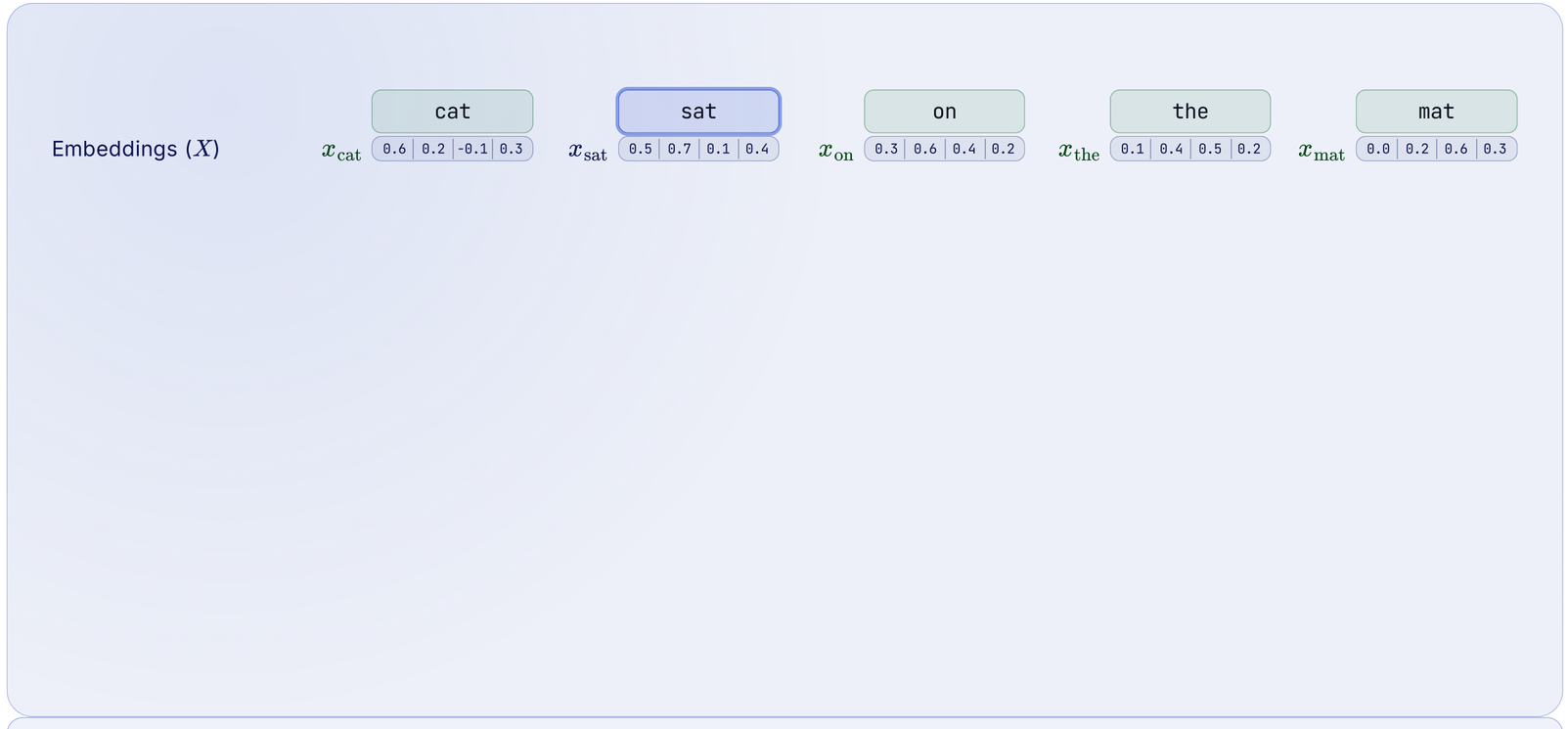


Next, score every k_j against q_{sat} : $s_j = q_{\text{sat}}^{\mathsf{T}} k_j$.

MECHANISM

Step 2 — Compute attention scores

Compare q_{sat} with every key k_j

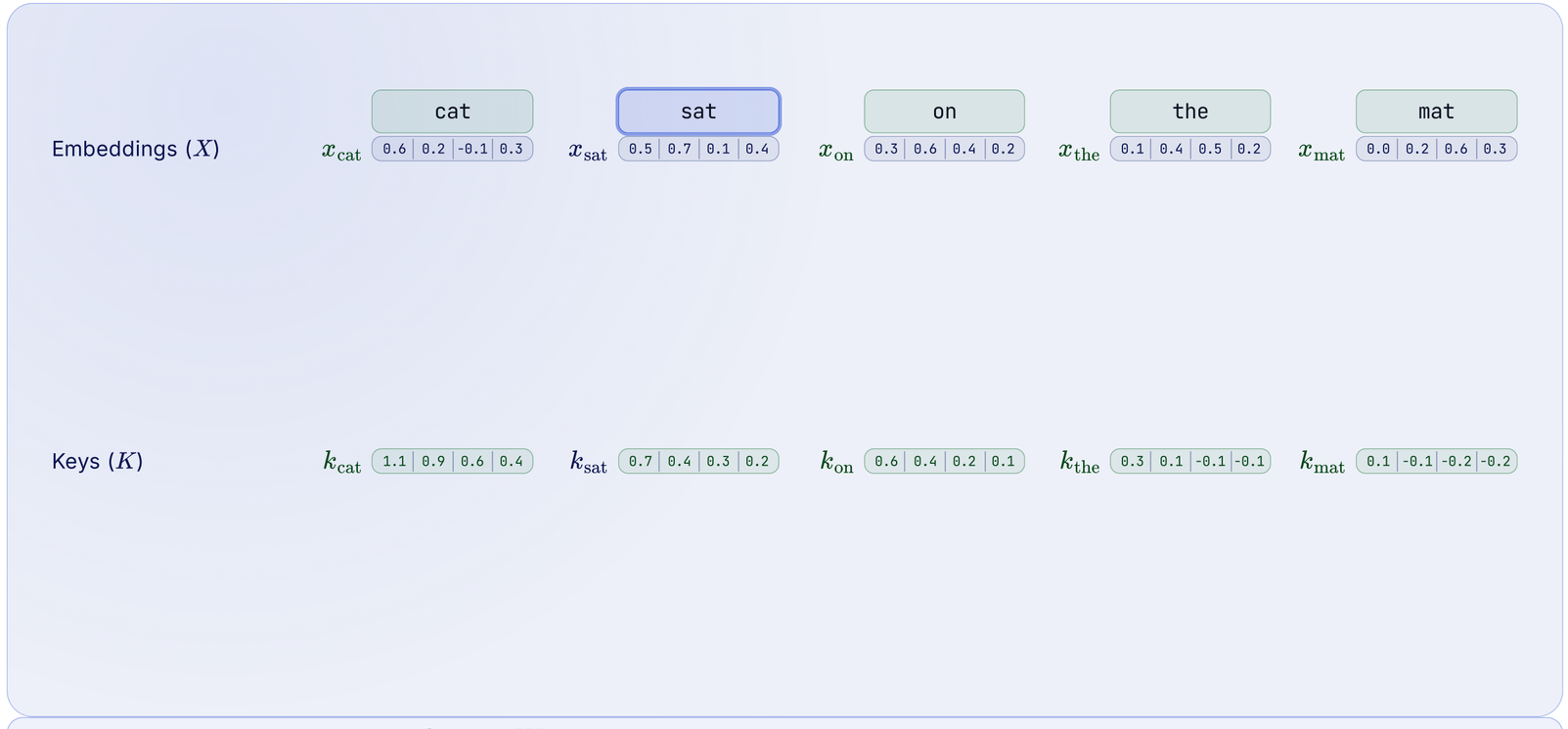


Start from embeddings $x_1, \ldots, x_n$.

MECHANISM

Step 2 — Compute attention scores

Compare q_{sat} with every key k_j

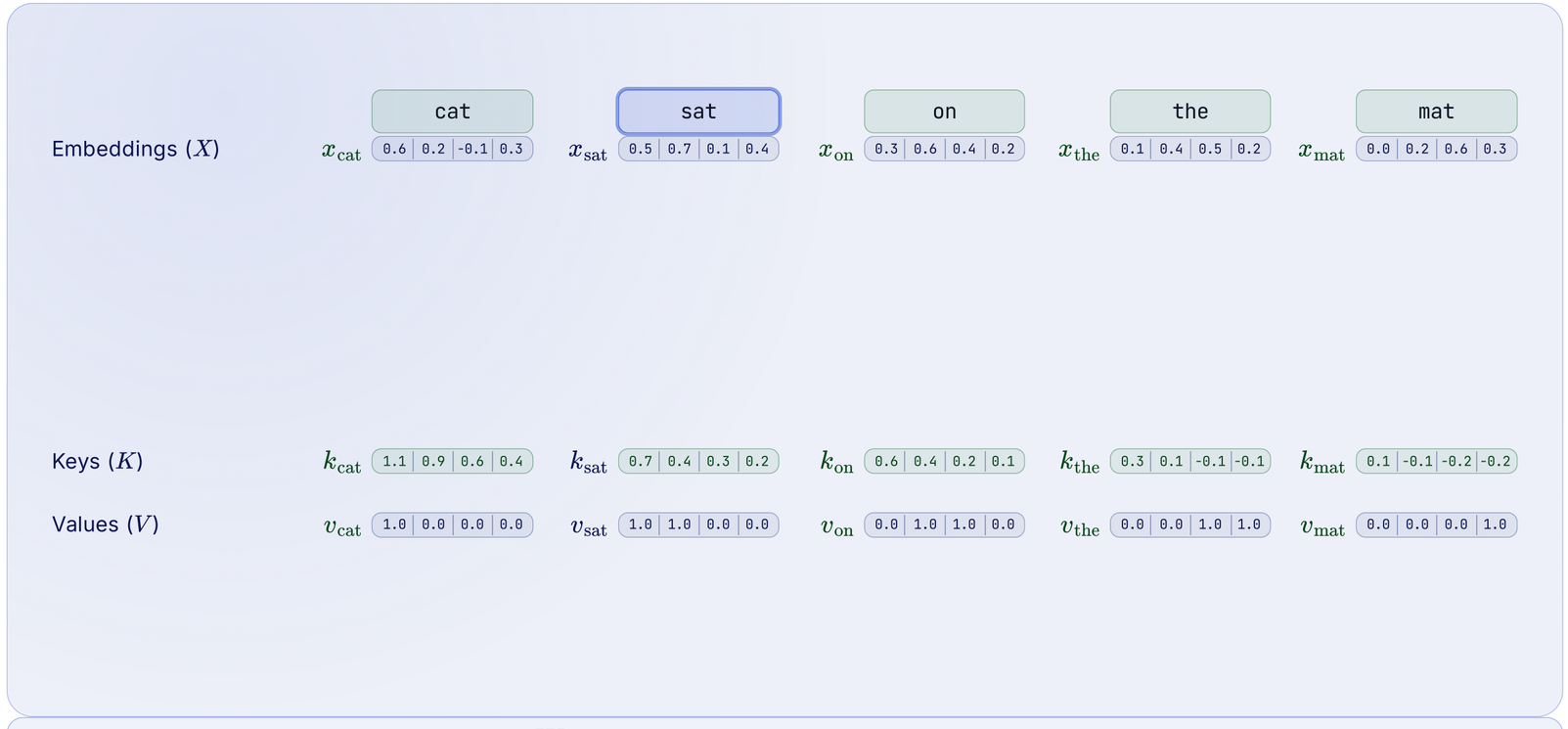


Project each token into a key vector: $k_j = x_j W_K$.

MECHANISM

Step 2 — Compute attention scores

Compare q_{sat} with every key k_j

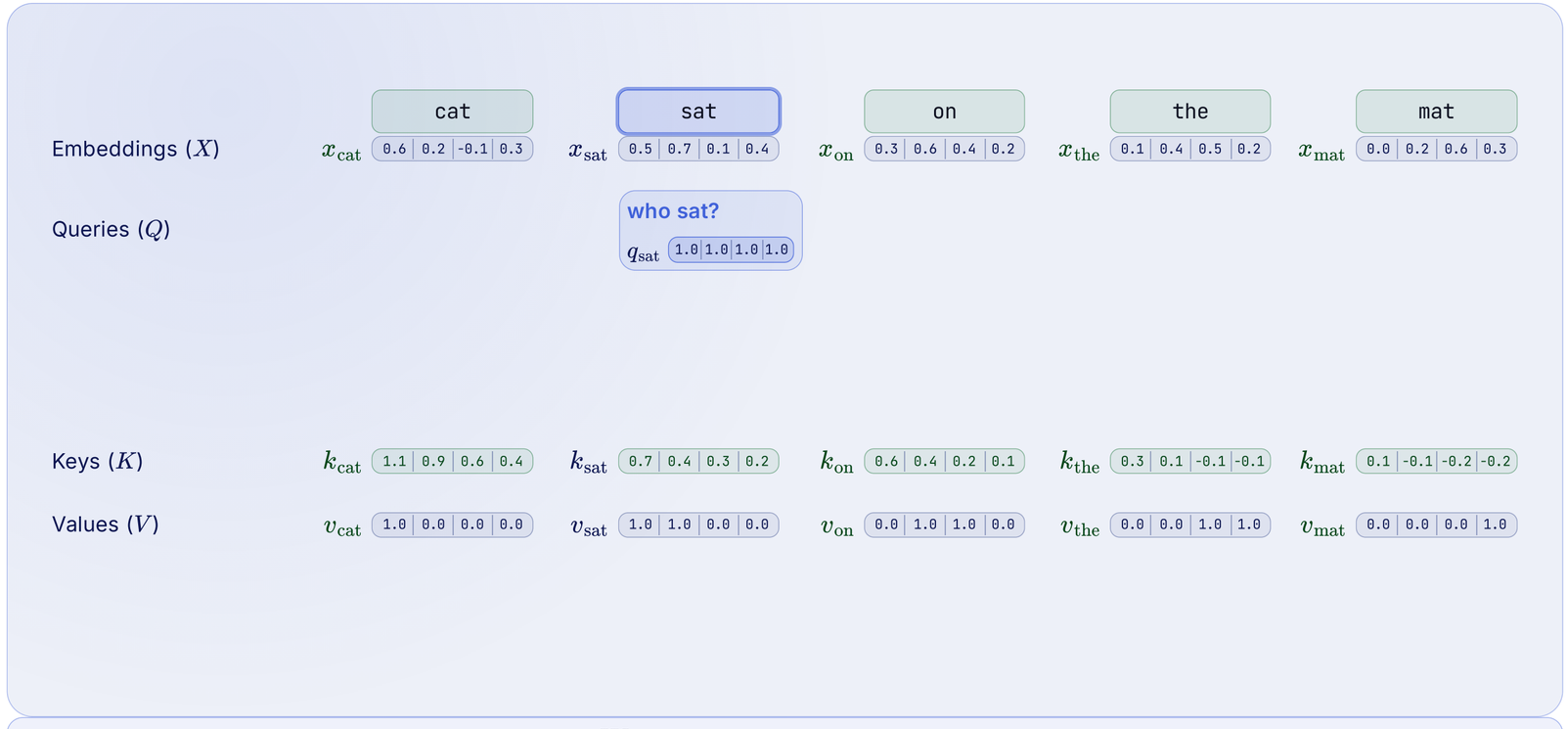


Project each token into a value vector: $v_j = x_j W_V$. These are the vectors copied forward.

MECHANISM

Step 2 — Compute attention scores

Compare q_{sat} with every key k_j

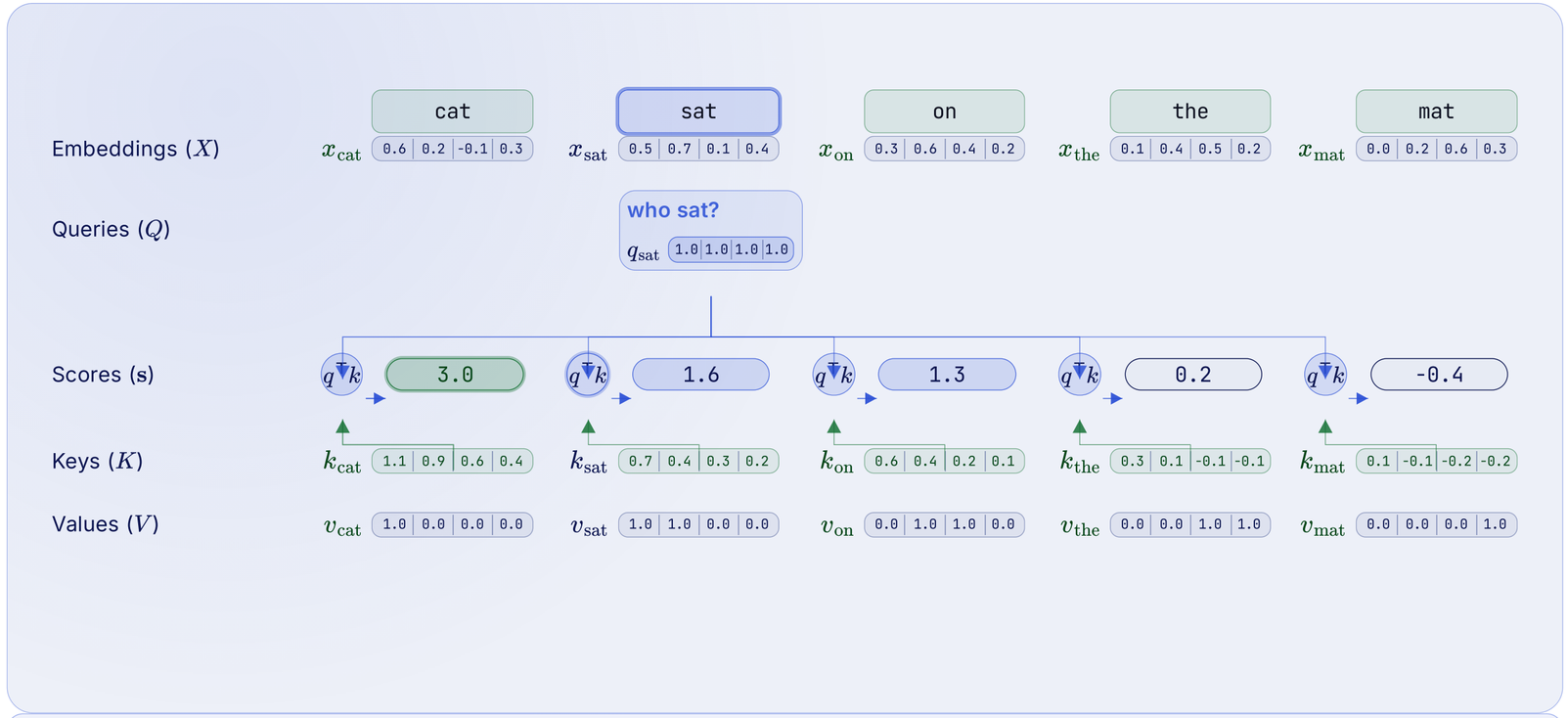


For the focus token, build a query vector: $q_{\text{sat}} = x_{\text{sat}} W_Q$.

MECHANISM

Step 2 — Compute attention scores

Compare q_{sat} with every key k_j

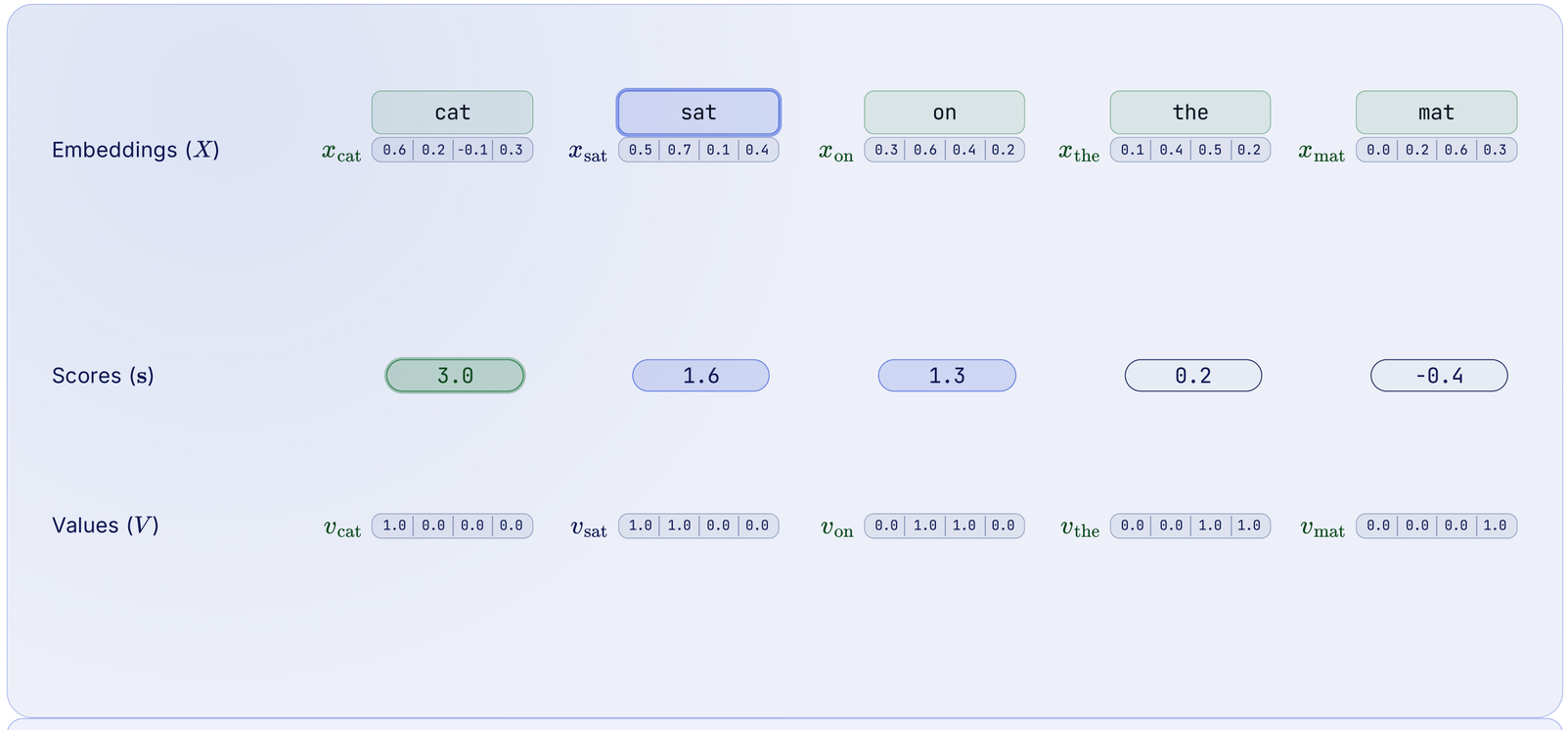


Score each token by comparing q_{sat} with k_j : $s_j = q_{\text{sat}}^{\top} k_j$.

MECHANISM

Step 2 — Compute attention scores

Compare q_{sat} with every key k_j

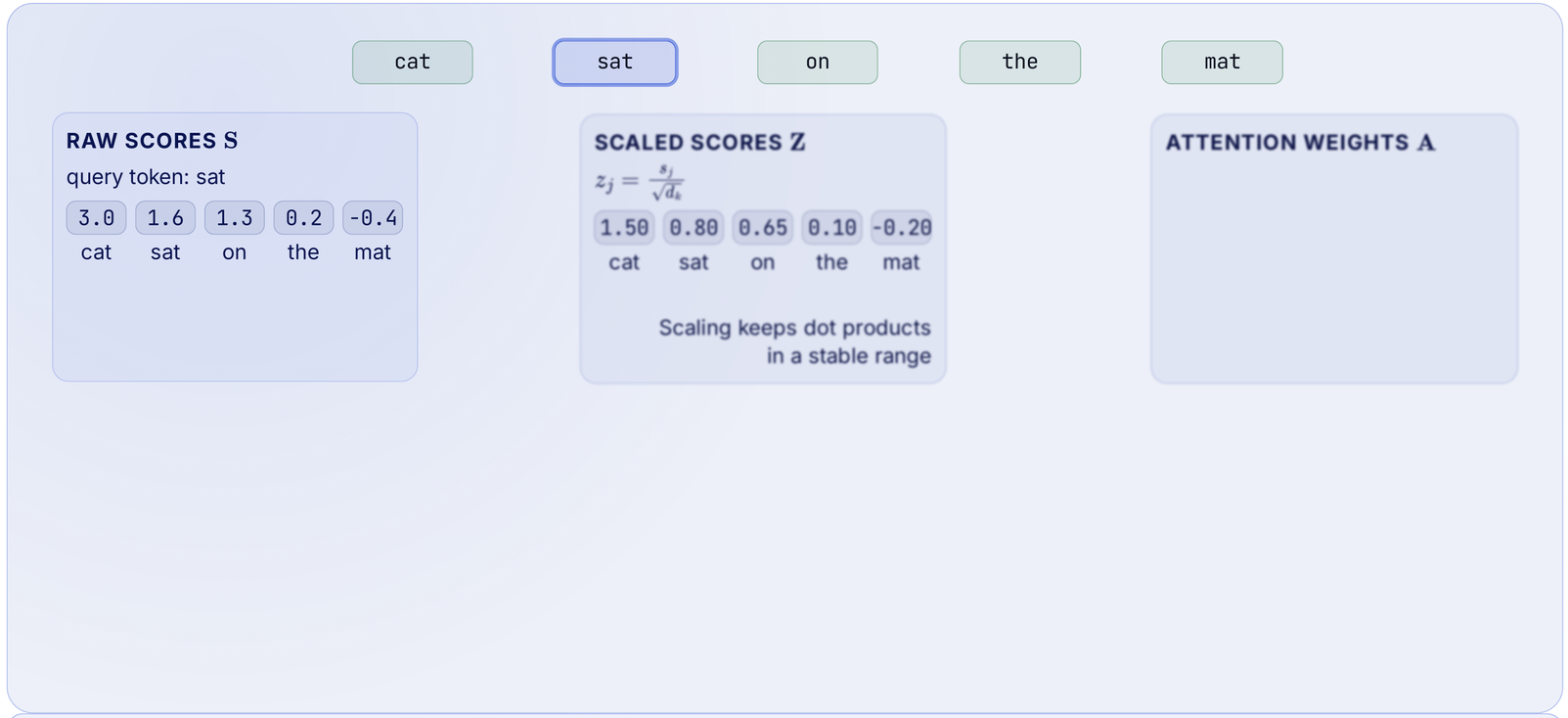


The scores are ready. The next step turns them into attention weights.

MECHANISM

Step 3 — From scores to attention weights

Scale the scores, then softmax them into weights.

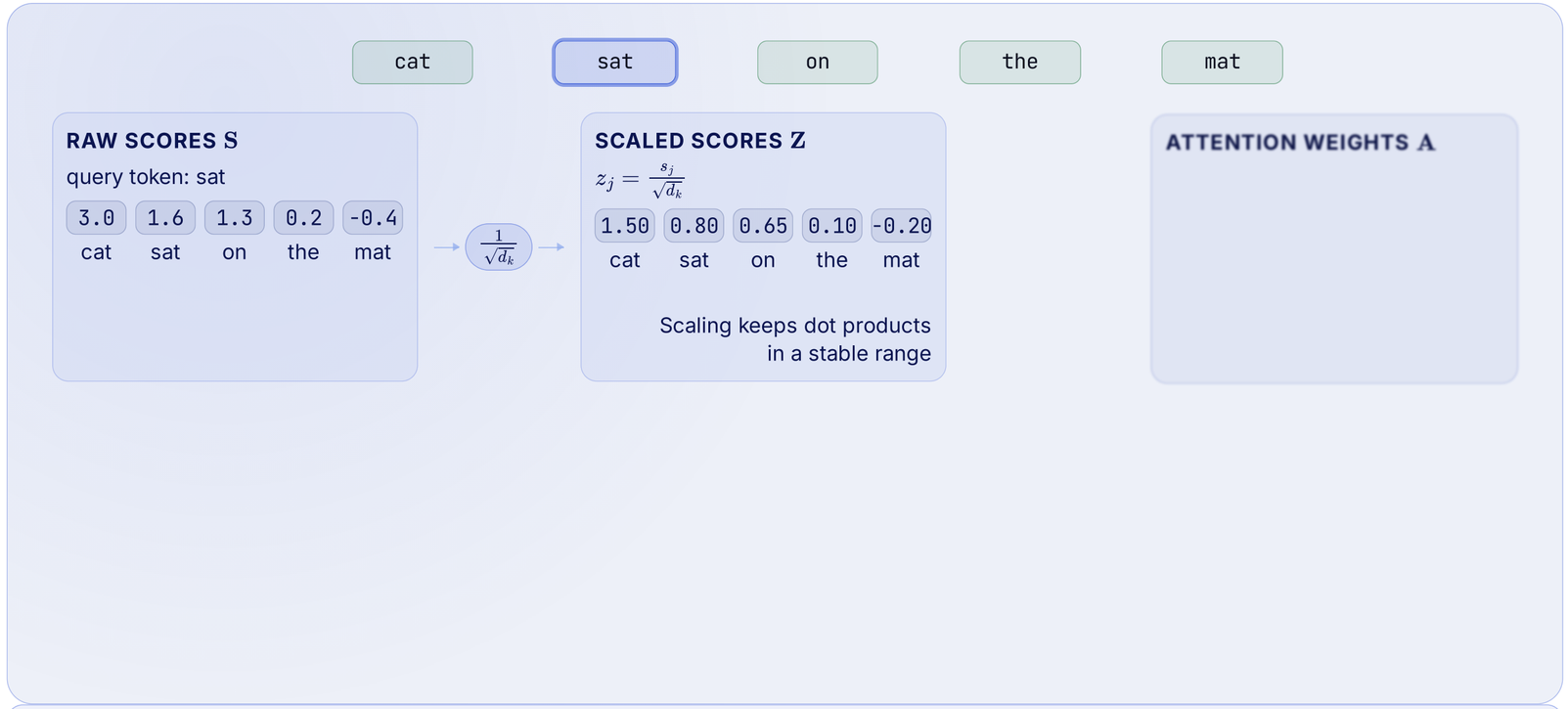


Raw similarity scores compare q_{sat} with each key k_j : $s_j = q_{\text{sat}}^{\top} k_j$.

MECHANISM

Step 3 — From scores to attention weights

Scale the scores, then softmax them into weights.

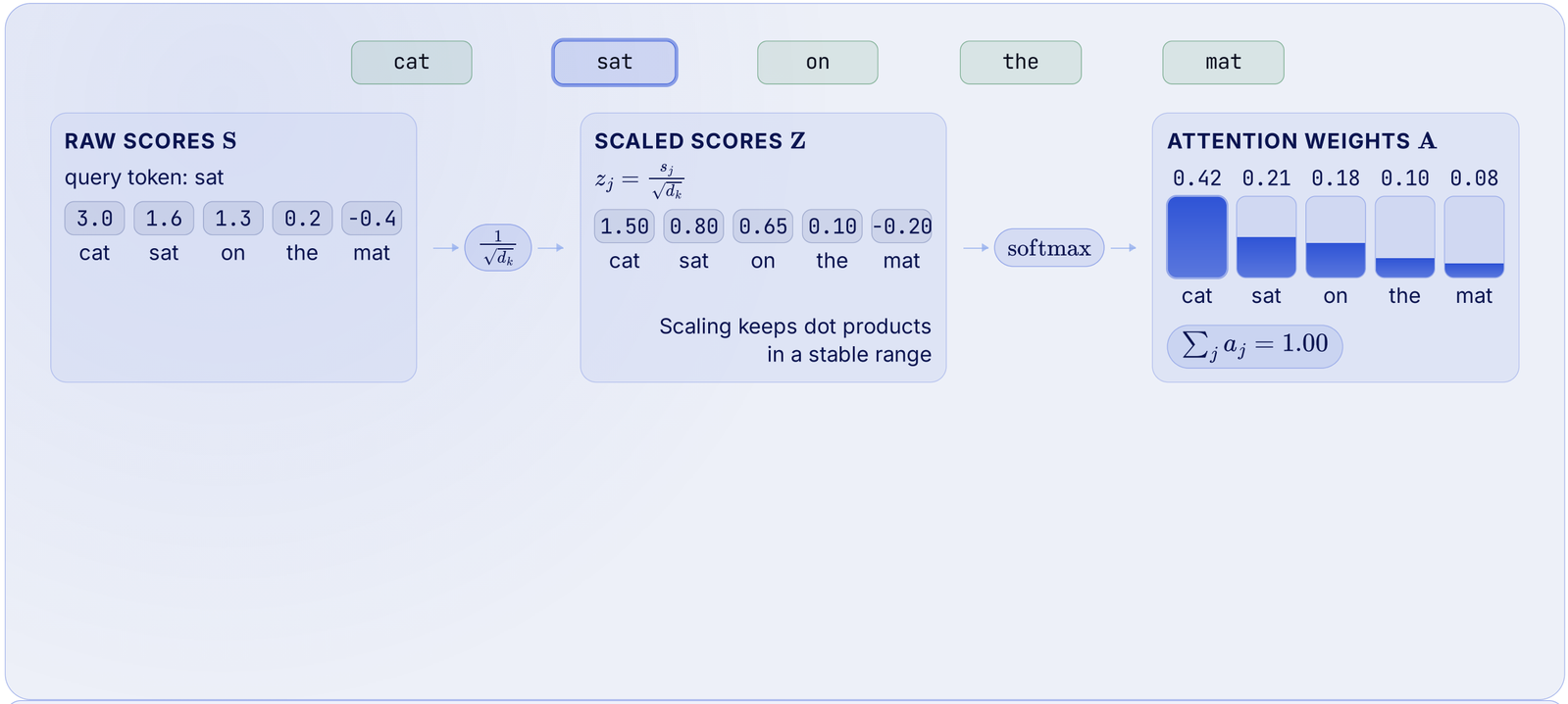


Scale each score into $z_j = \frac{s_j}{\sqrt{d_k}}$ to keep magnitudes stable before $\mathrm{softmax}$.

MECHANISM

Step 3 — From scores to attention weights

Scale the scores, then softmax them into weights.



Normalize row-wise: $a_j = \frac{\exp(z_j)}{\sum_\ell \exp(z_\ell)}$.

MECHANISM

Step 3 — From scores to attention weights

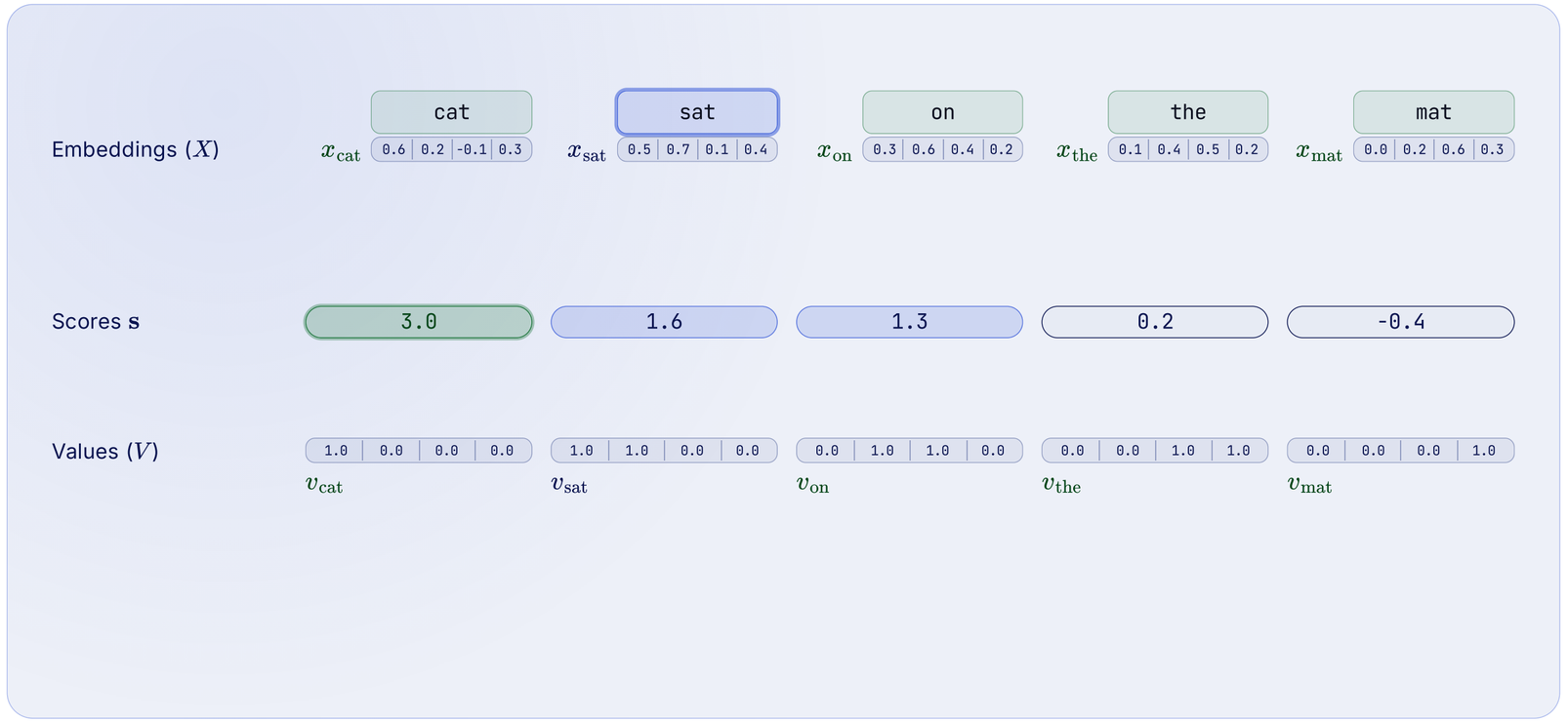
Scale the scores, then softmax them into weights.

These weights determine how much each value vector v_j contributes to the update.

MECHANISM

Step 4 — Mix values, then add the residual

Mix value vectors with attention weights, then add back x_{sat}

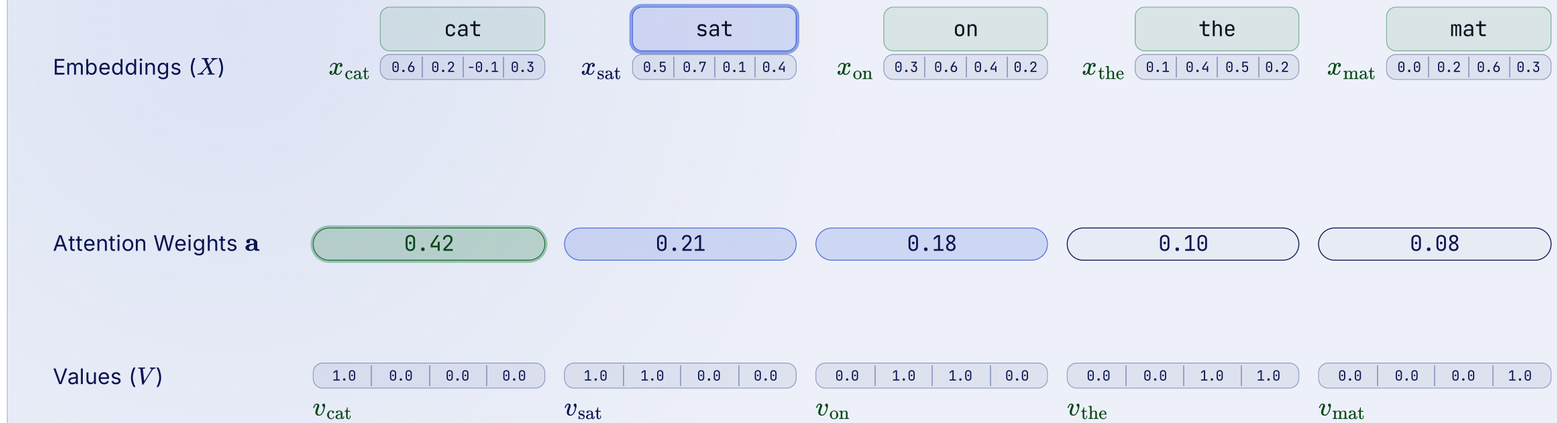


Start from the Step 3 scores for q_{sat} .

MECHANISM

Step 4 — Mix values, then add the residual

Mix value vectors with attention weights, then add back x_{sat}

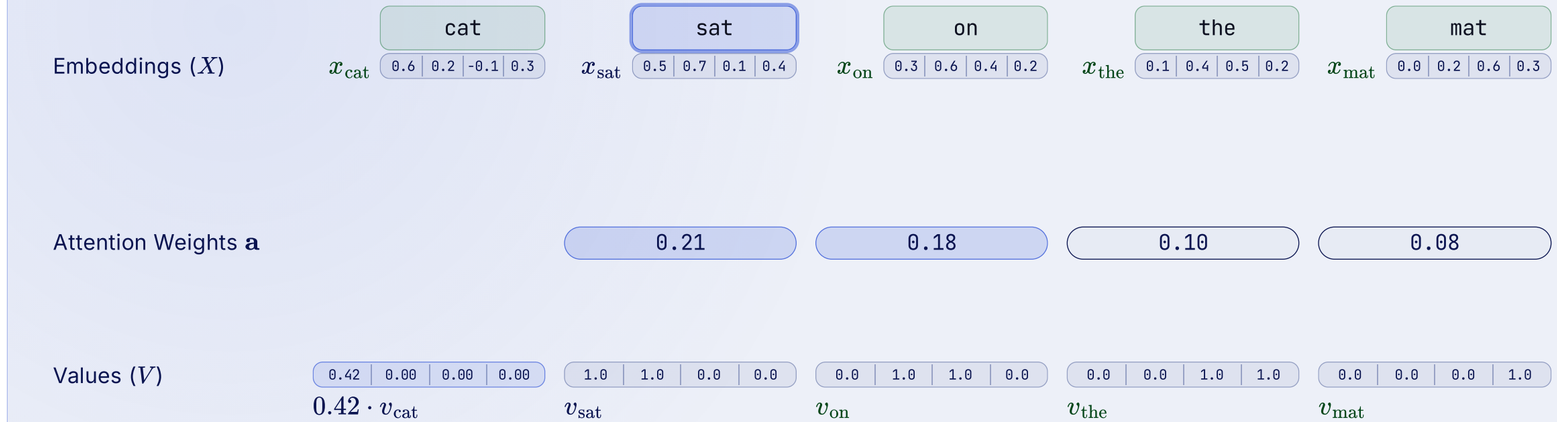


Convert them into attention weights a_j .

MECHANISM

Step 4 — Mix values, then add the residual

Mix value vectors with attention weights, then add back x_{sat}

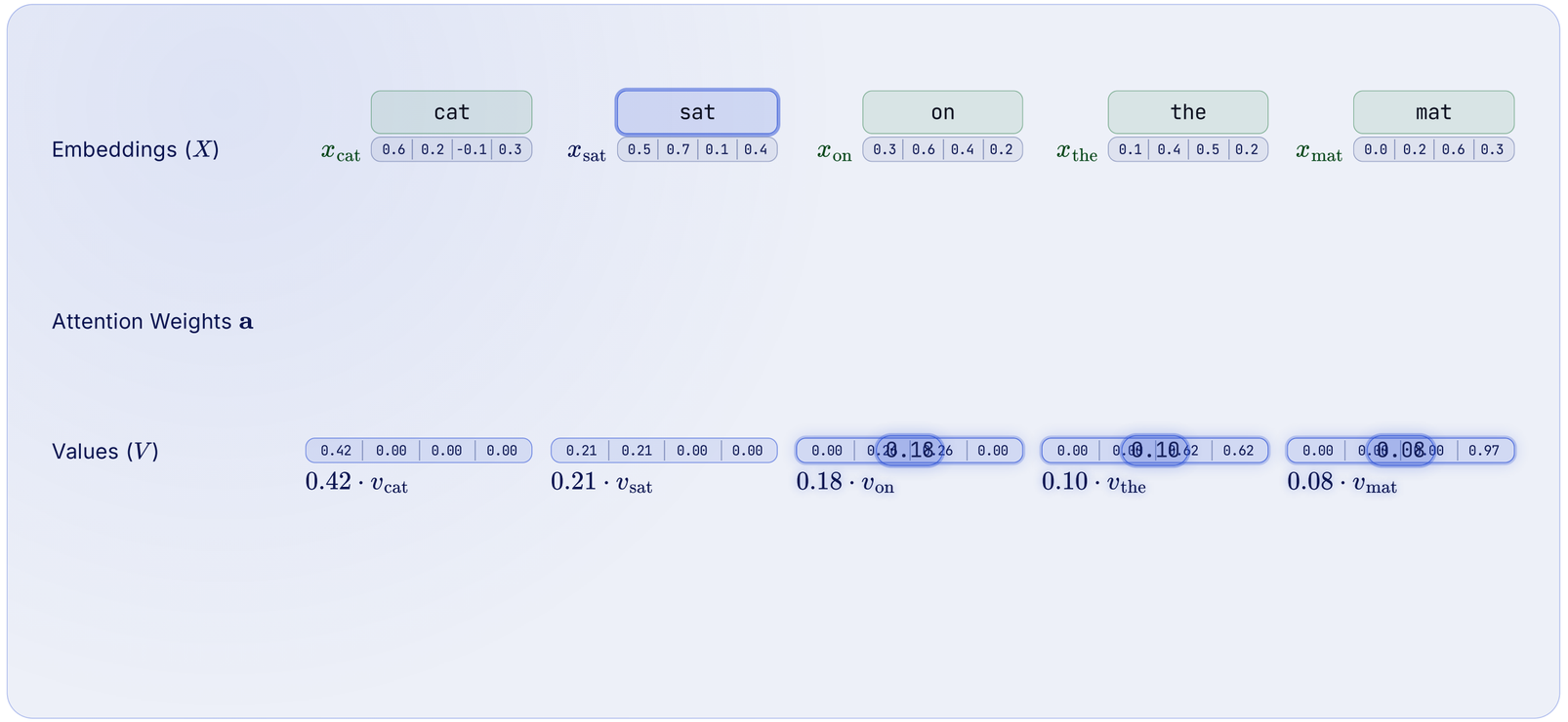


Apply the first weight to its value vector.

MECHANISM

Step 4 — Mix values, then add the residual

Mix value vectors with attention weights, then add back x_{sat}



Apply the remaining weights to the other value vectors.

MECHANISM

Step 4 — Mix values, then add the residual

Mix value vectors with attention weights, then add back x_{sat}

Sum the weighted values into o_{sat} .

MECHANISM

Step 4 — Mix values, then add the residual

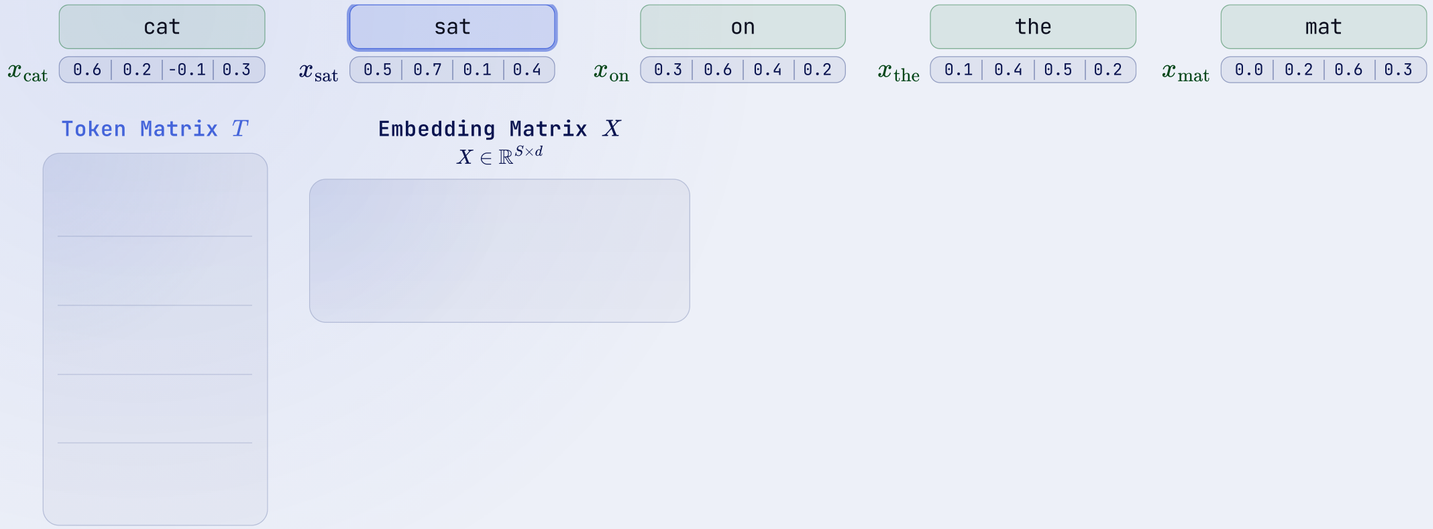
Mix value vectors with attention weights, then add back x_{sat}

Add the residual: $x'_{\text{sat}} = x_{\text{sat}} + o_{\text{sat}}$.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

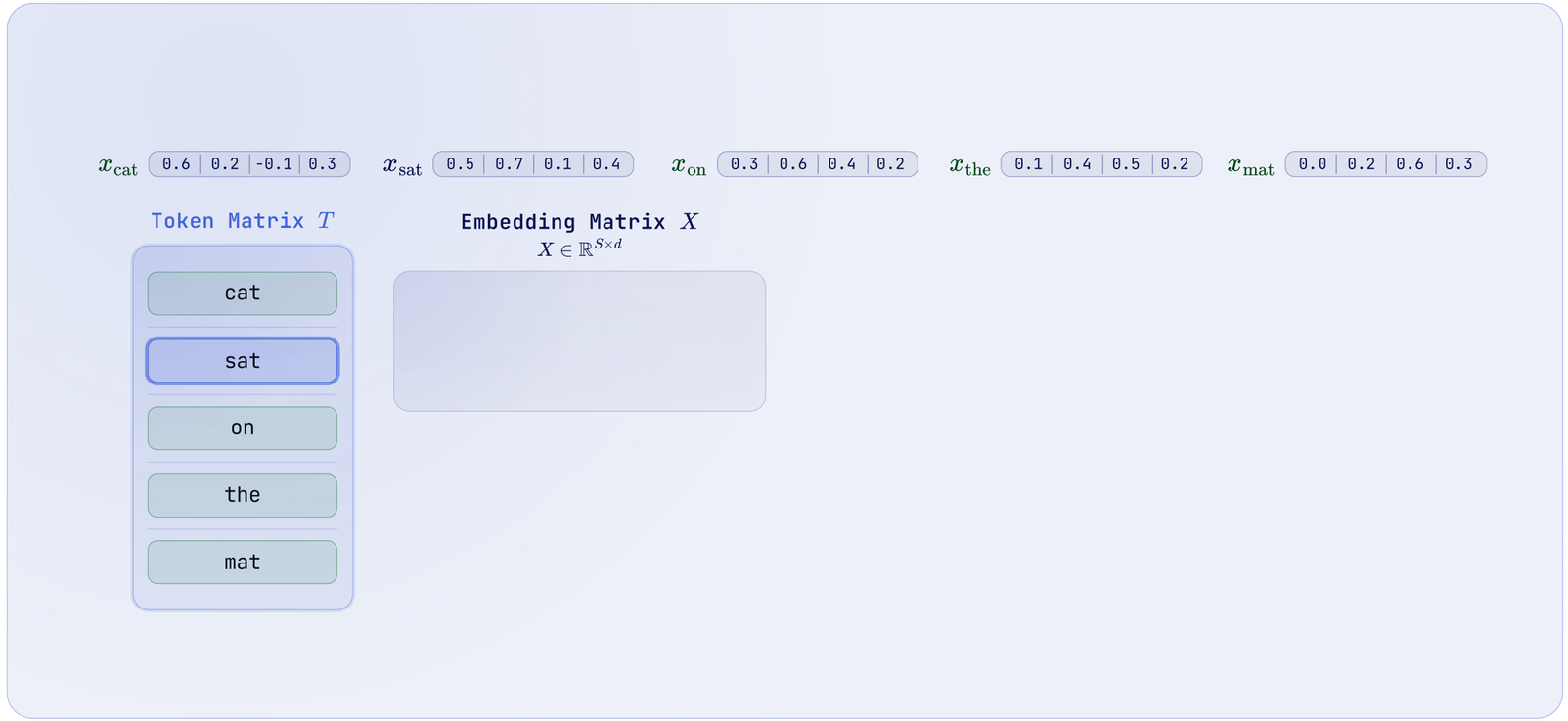


Start from the same sequence: tokens and their embedding rows.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

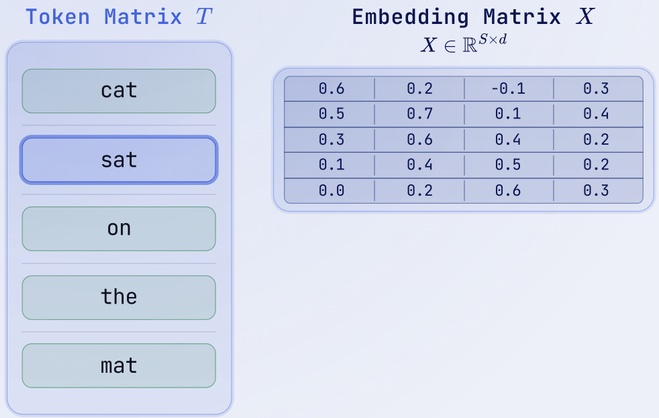


Collect the sequence into a compact token matrix T.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.



Embedding Matrix X

$X \in \mathbb{R}^{S \times d}$

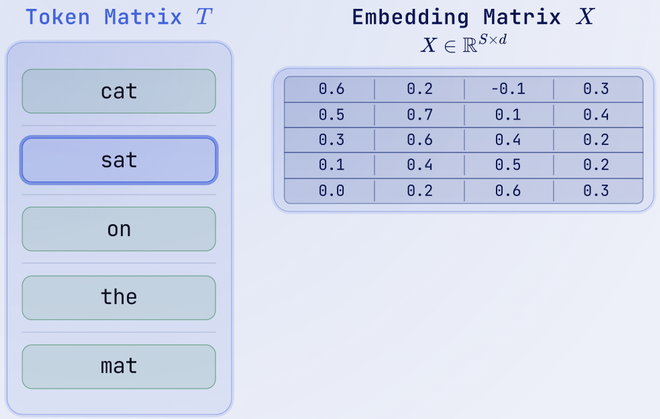
0.6	0.2	-0.1	0.3
0.5	0.7	0.1	0.4
0.3	0.6	0.4	0.2
0.1	0.4	0.5	0.2
0.0	0.2	0.6	0.3

Collect the embedding rows into the embedding matrix X.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

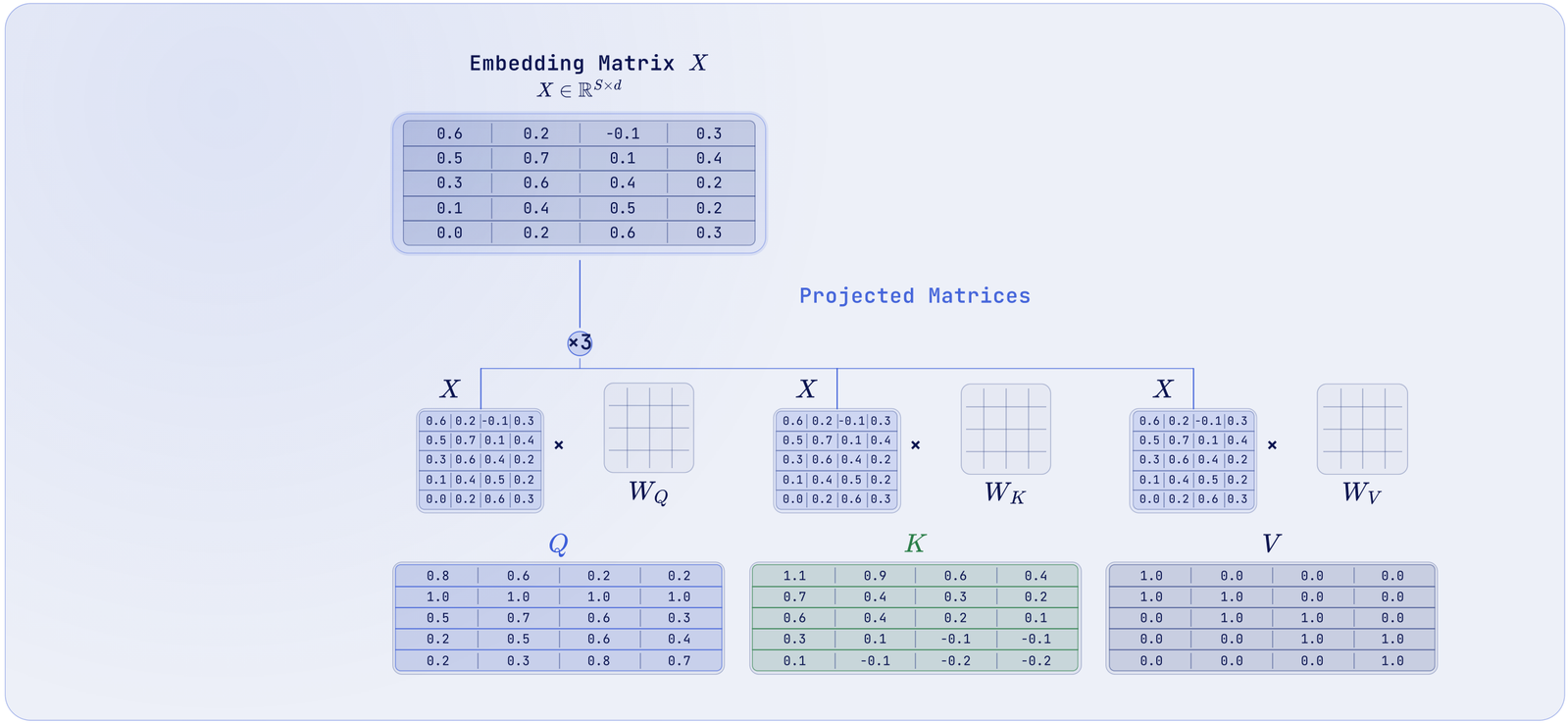


Lets collapse the original sequence view and keep the compact matrices.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.



Project X into Q, K, and V for the full sequence at once.

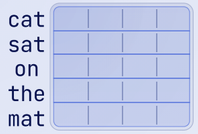
MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

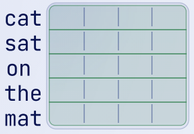
Q

$Q \in \mathbb{R}^{S \times d_k}$



K

$K \in \mathbb{R}^{S \times d_k}$



Bring Q and K to the center as the matrices used for score computation.

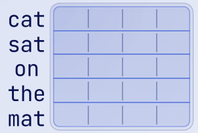
MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Q

$Q \in \mathbb{R}^{S \times d_k}$



K^T

$K^\mathsf{T} \in \mathbb{R}^{d_k \times S}$

Transpose K so the matrix dimensions line up for multiplication.

MECHANISM

Attention in matrix form

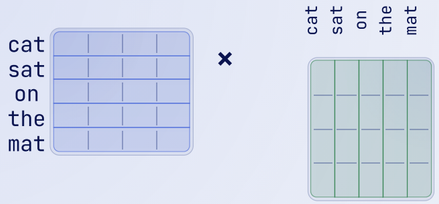
Stack tokens into matrices, then compute the full sequence at once.

Q

$Q \in \mathbb{R}^{S \times d_k}$

K^T

$K^\mathsf{T} \in \mathbb{R}^{d_k \times S}$



Raw Scores S

$S \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.6	0.9	0.8	0.3	-0.1
sat	3.0	1.6	1.3	0.2	-0.4
on	1.7	0.9	0.7	0.1	-0.2
the	1.2	0.6	0.5	0.0	-0.2
mat	1.3	0.6	0.5	-0.1	-0.3

Compute raw attention scores for the whole sequence: $S = QK^\mathsf{T}$.

MECHANISM

Attention in matrix form

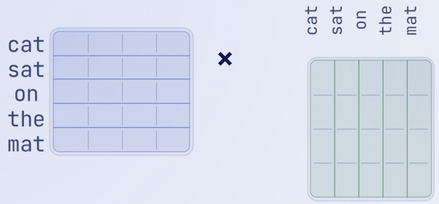
Stack tokens into matrices, then compute the full sequence at once.

Q

$Q \in \mathbb{R}^{S \times d_k}$

$K^\top$

$K^\top \in \mathbb{R}^{d_k \times S}$



Raw Scores S

$S \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.6	0.9	0.8	0.3	-0.1
sat	3.0	1.6	1.3	0.2	-0.4
on	1.7	0.9	0.7	0.1	-0.2
the	1.2	0.6	0.5	0.0	-0.2
mat	1.3	0.6	0.5	-0.1	-0.3

Without a causal mask, earlier tokens can attend to later tokens.
That leaks future information and breaks left-to-right generation.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Q

$Q \in \mathbb{R}^{S \times d_k}$

$K^\top$

$K^\top \in \mathbb{R}^{d_k \times S}$

Raw Scores S

$S \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.6	-∞	-∞	-∞	-∞
sat	3.0	1.6	-∞	-∞	-∞
on	1.7	0.9	0.7	-∞	-∞
the	1.2	0.6	0.5	0.0	-∞
mat	1.3	0.6	0.5	-0.1	-0.3

The fix is a causal mask.

Replace every future-position score with $-\infty$, leaving only the current and earlier tokens available.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Q

$Q \in \mathbb{R}^{S \times d_k}$

$K^\top$

$K^\top \in \mathbb{R}^{d_k \times S}$

Raw Scores S

$S \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.6	-∞	-∞	-∞	-∞
sat	3.0	1.6	-∞	-∞	-∞
on	1.7	0.9	0.7	-∞	-∞
the	1.2	0.6	0.5	0.0	-∞
mat	1.3	0.6	0.5	-0.1	-0.3

Blocked cells show $-\infty$, not 0.
Softmax will turn those entries into exactly zero attention.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Masked Scores S

$S \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.6	-∞	-∞	-∞	-∞
sat	3.0	1.6	-∞	-∞	-∞
on	1.7	0.9	0.7	-∞	-∞
the	1.2	0.6	0.5	0.0	-∞
mat	1.3	0.6	0.5	-0.1	-0.3

Focus on the masked score matrix row by row.
Attention normalizes each query row independently.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Scaled Scores Z

$$Z \in \mathbb{R}^{S \times S}$$

	cat	sat	on	the	mat
cat	0.8	-∞	-∞	-∞	-∞
sat	1.5	0.8	-∞	-∞	-∞
on	0.8	0.4	0.4	-∞	-∞
the	0.6	0.3	0.2	0.0	-∞
mat	0.6	0.3	0.2	-0.0	-0.2

$$\frac{1}{\sqrt{d_k}}$$

Scale the allowed scores: $z_{ij} = \frac{s_{ij}}{\sqrt{d_k}}$.

Masked future positions stay at $-\infty$.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Attention Matrix A

$A \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.00	0.00	0.00	0.00	0.00
sat	0.67	0.33	0.00	0.00	0.00
on	0.43	0.29	0.27	0.00	0.00
the	0.33	0.25	0.23	0.18	0.00
mat	0.29	0.22	0.20	0.15	0.14

$\sum_j a_{ij} = 1$

$$a_{ij} = \frac{\exp(z_{ij})}{\sum_{\ell=1}^{S} \exp(z_{i\ell})}$$

$\exp(-\infty) = 0$ and masked cells become 0.

Because $\exp(-\infty) = 0$, masked future entries become zero attention.
Each row becomes a valid distribution.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.

Attention Matrix A

$A \in \mathbb{R}^{S \times S}$

	cat	sat	on	the	mat
cat	1.00	0.00	0.00	0.00	0.00
sat	0.67	0.33	0.00	0.00	0.00
on	0.43	0.29	0.27	0.00	0.00
the	0.33	0.25	0.23	0.18	0.00
mat	0.29	0.22	0.20	0.15	0.14

×

Value Matrix V

$V \in \mathbb{R}^{S \times d_v}$

1.0	0.0	0.0	0.0
1.0	1.0	0.0	0.0
0.0	1.0	1.0	0.0
0.0	0.0	1.0	1.0
0.0	0.0	0.0	1.0

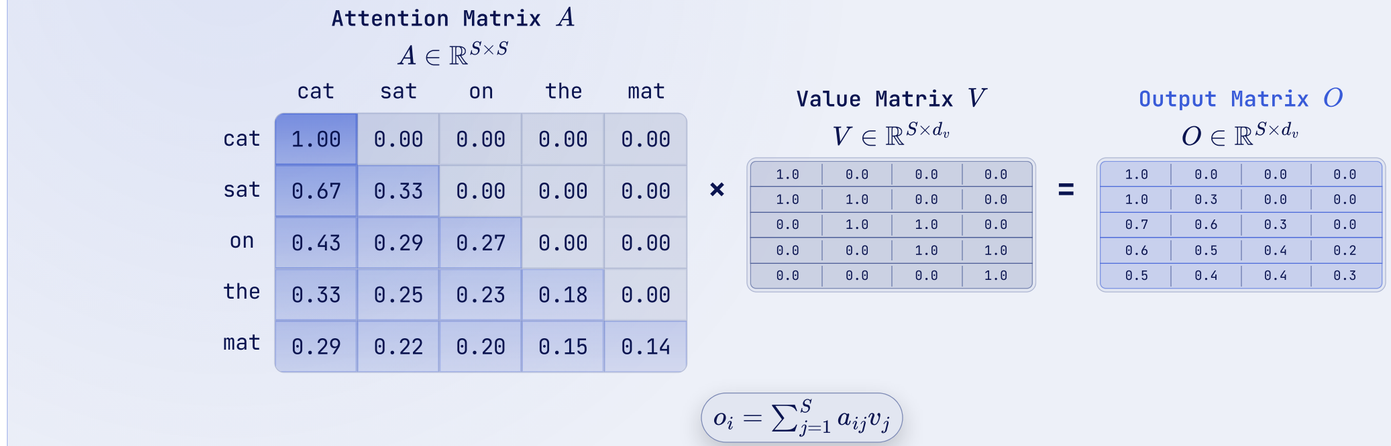
Bring in the value matrix V.

The attention matrix A determines how much of each value row to mix into each output row.

MECHANISM

Attention in matrix form

Stack tokens into matrices, then compute the full sequence at once.



Compute the weighted sum for the full sequence: $O = AV$.
Each output row is a weighted combination of value rows.

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

EMBEDDING MATRIX X

$$X \in \mathbb{R}^{S \times d}$$

0.6	0.2	-0.1	0.3
0.5	0.7	0.1	0.4
0.3	0.6	0.4	0.2
0.1	0.4	0.5	0.2
0.0	0.2	0.6	0.3

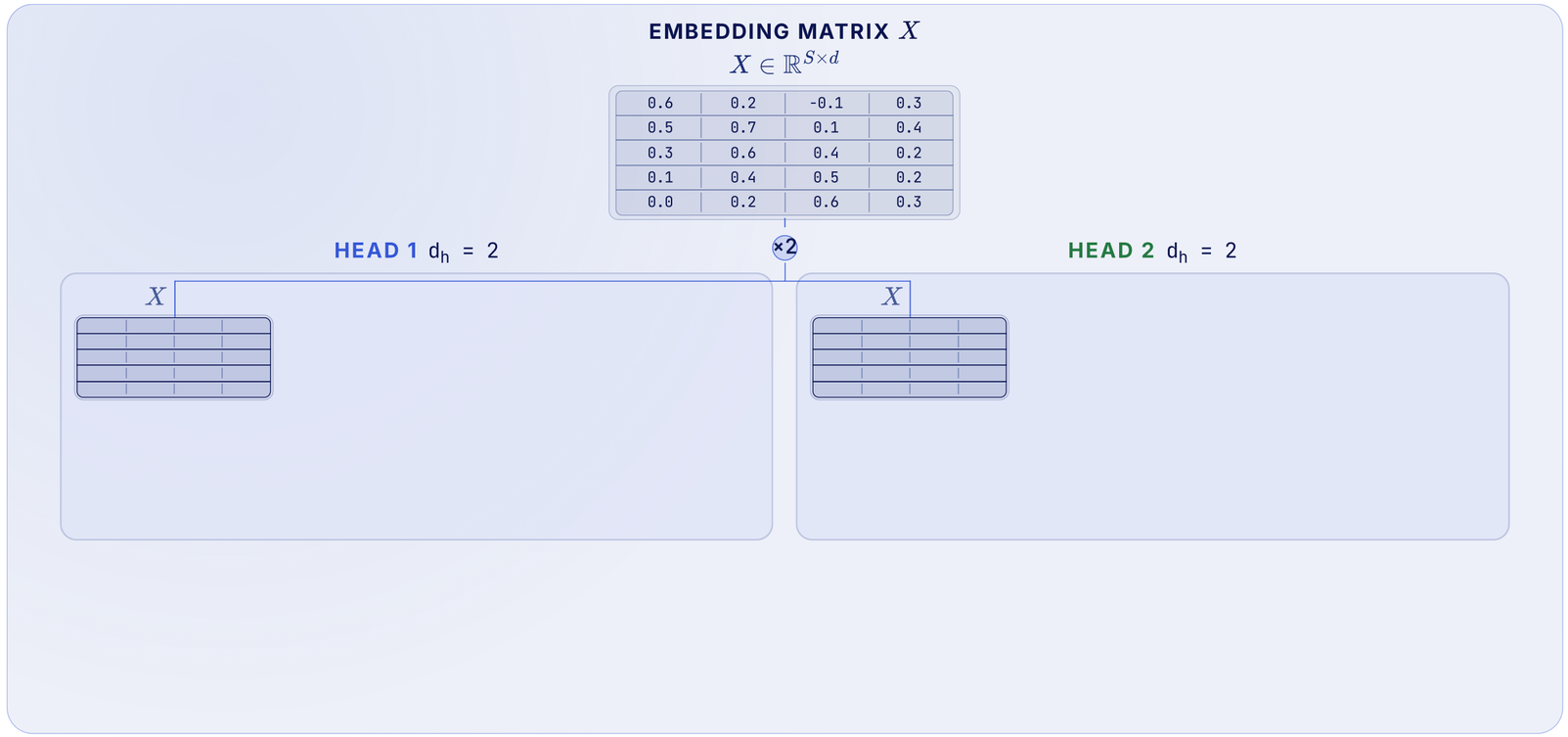
HEAD 1 $d_h = 2$

HEAD 2 $d_h = 2$

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

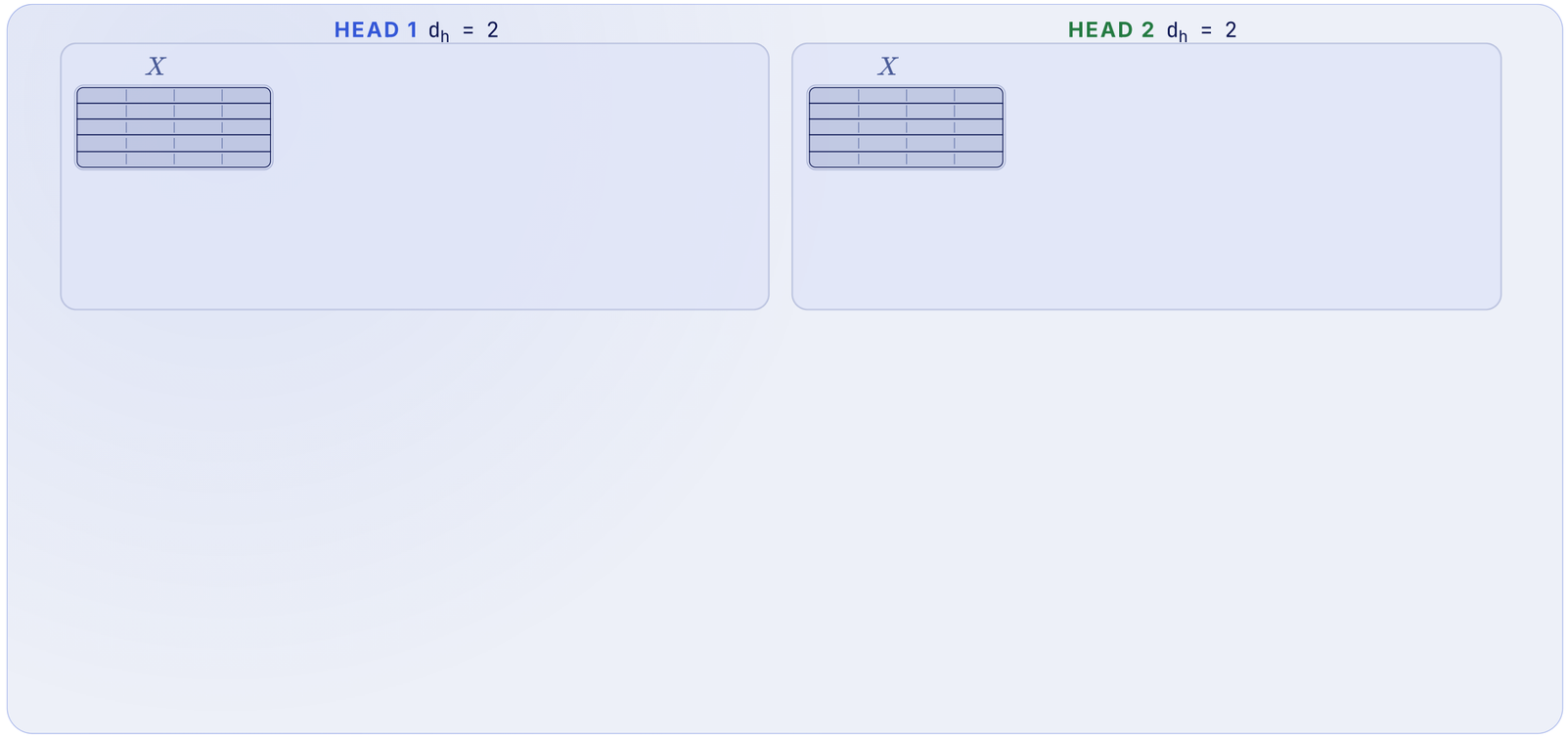


Make two copies of X, one for each head, so both attention paths can run in parallel.

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

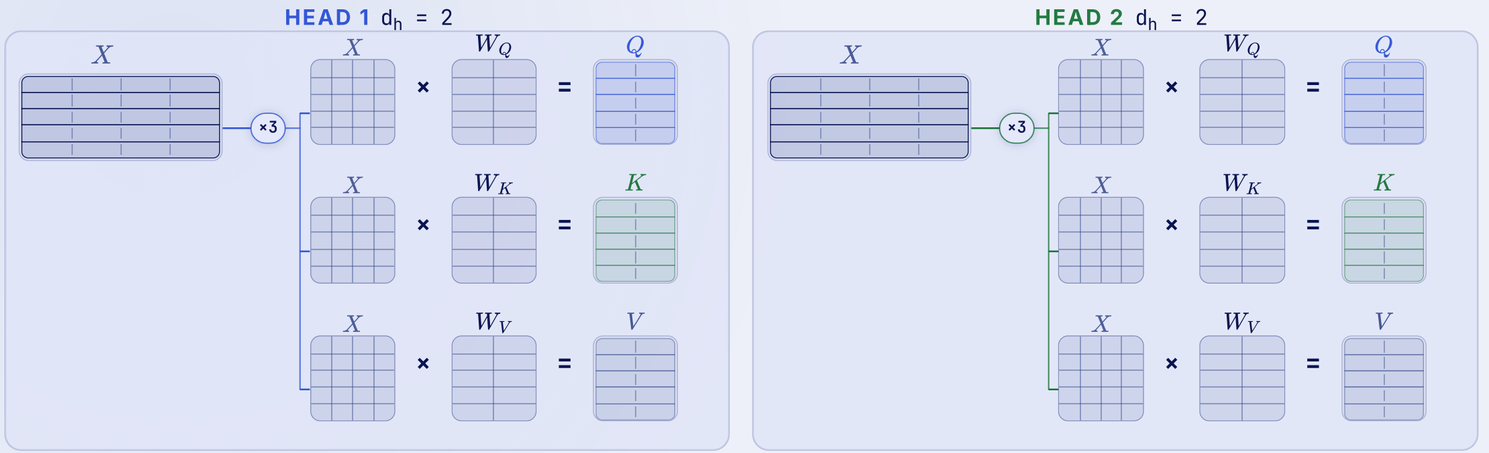


Each head starts from the same input sequence X, then computes attention in its own learned subspace.

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

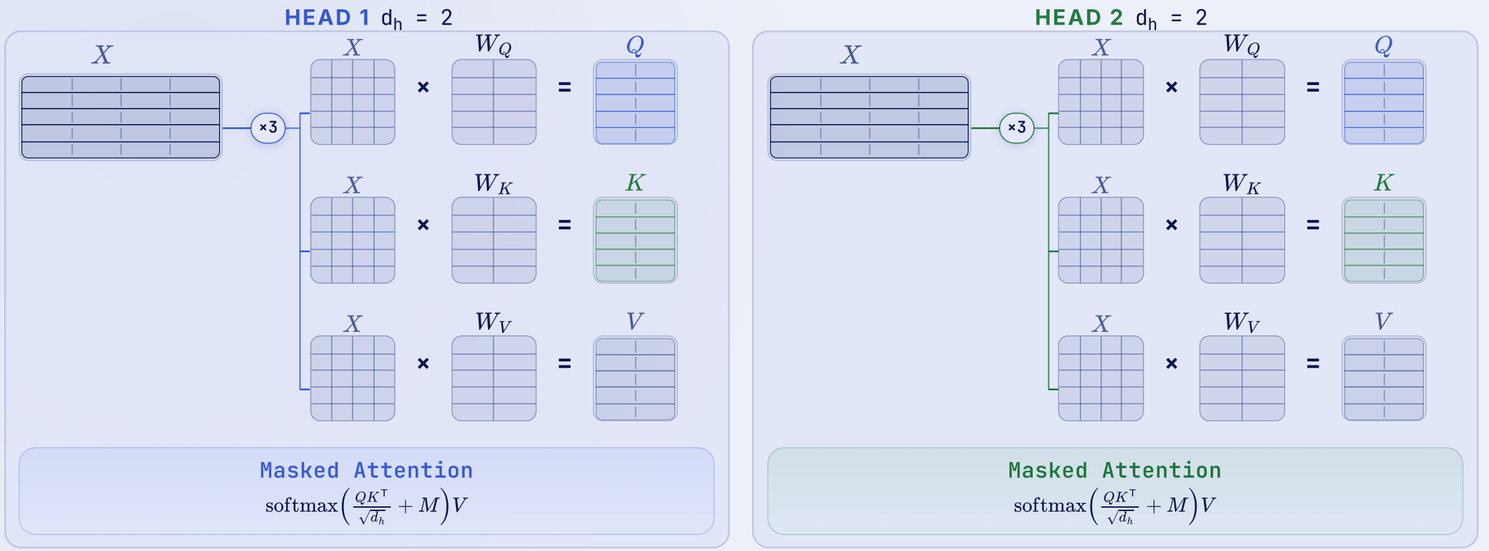


Each head applies its own learned projection matrices W_Q, W_K, and W_V to the same X, producing different head-specific Q, K, and V.

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

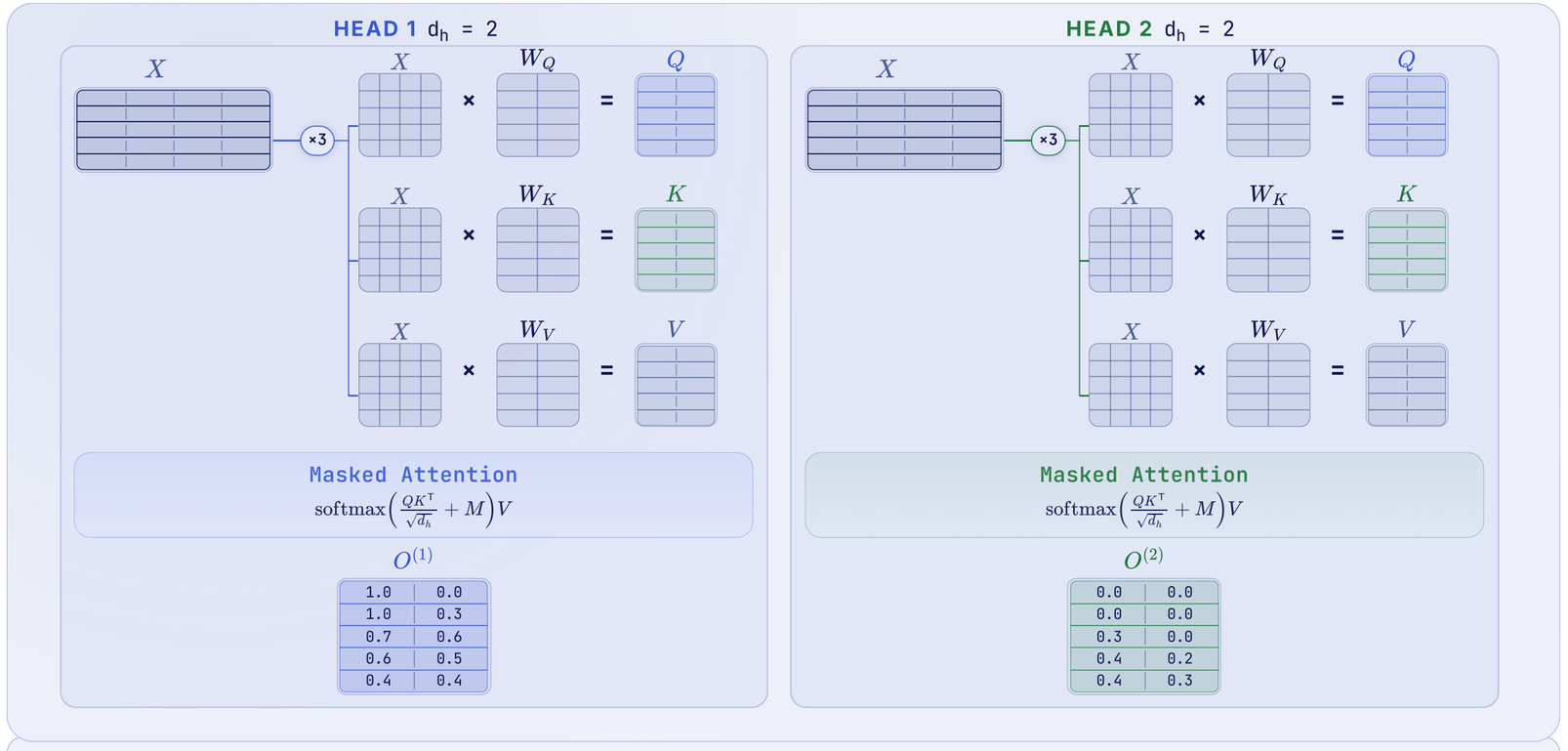


Each head now runs masked attention in parallel. Different learned projections let different heads focus on different relationships in the same sequence.

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.



The model now has one context representation per head.

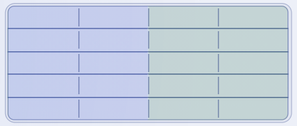
MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

Concatenated Heads

$$\mathrm{Concat}(O^{(1)}, O^{(2)}) \in \mathbb{R}^{S \times d}$$

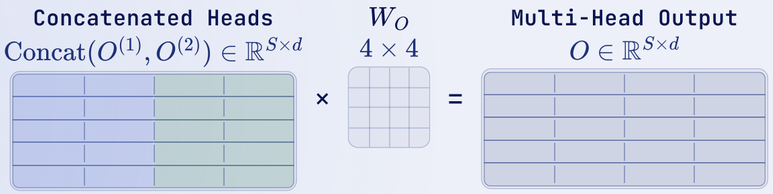


Concatenate the head outputs side by side to form one wider sequence matrix.

MECHANISM

Multi-Head Attention

Different heads attend in different learned subspaces, then their outputs are recombined.

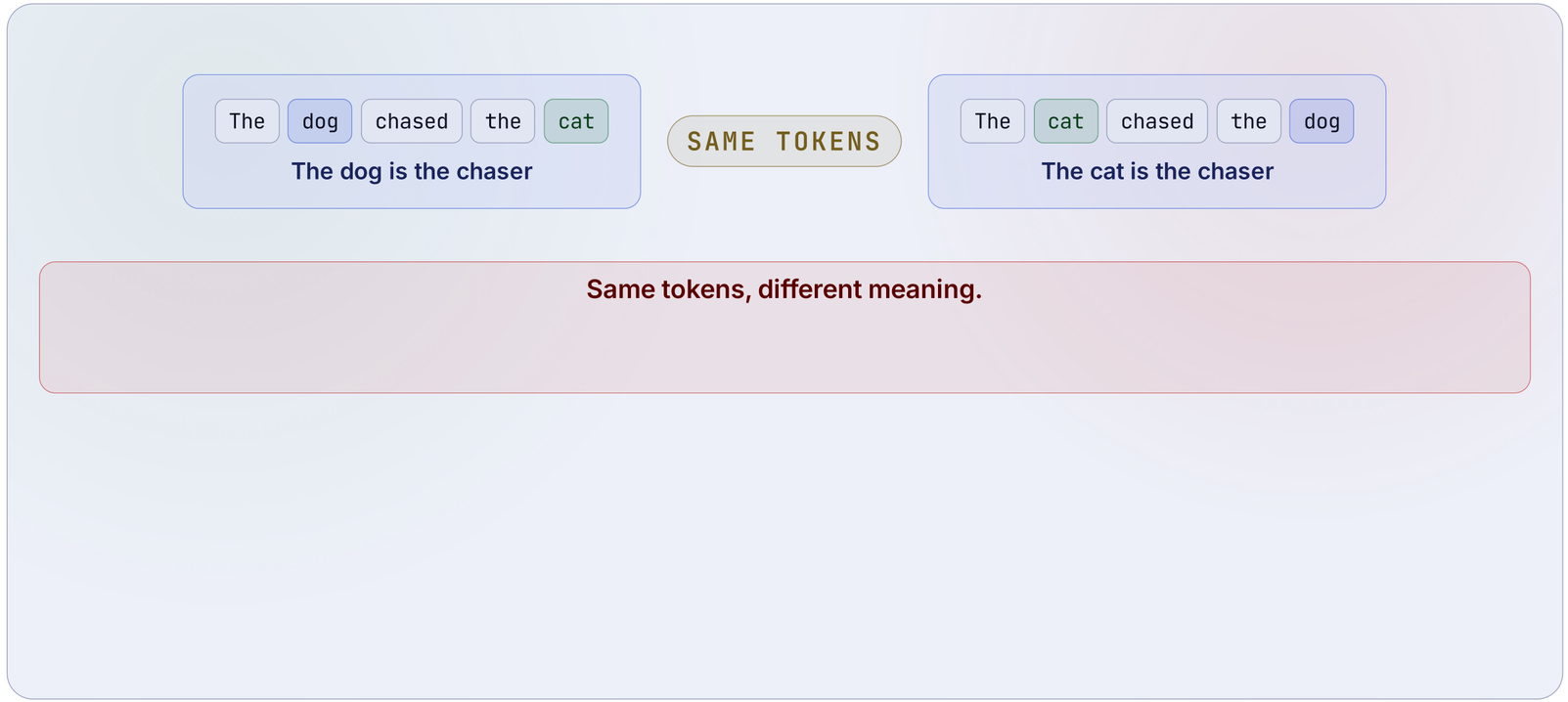


Apply the output projection W_O to mix the concatenated head outputs back into one shared representation.

PROBLEM

Self-Attention Compares Vectors, Not Word Order

Without positional signals, self-attention is permutation equivariant: reorder the token rows in, and the outputs reorder the same way.

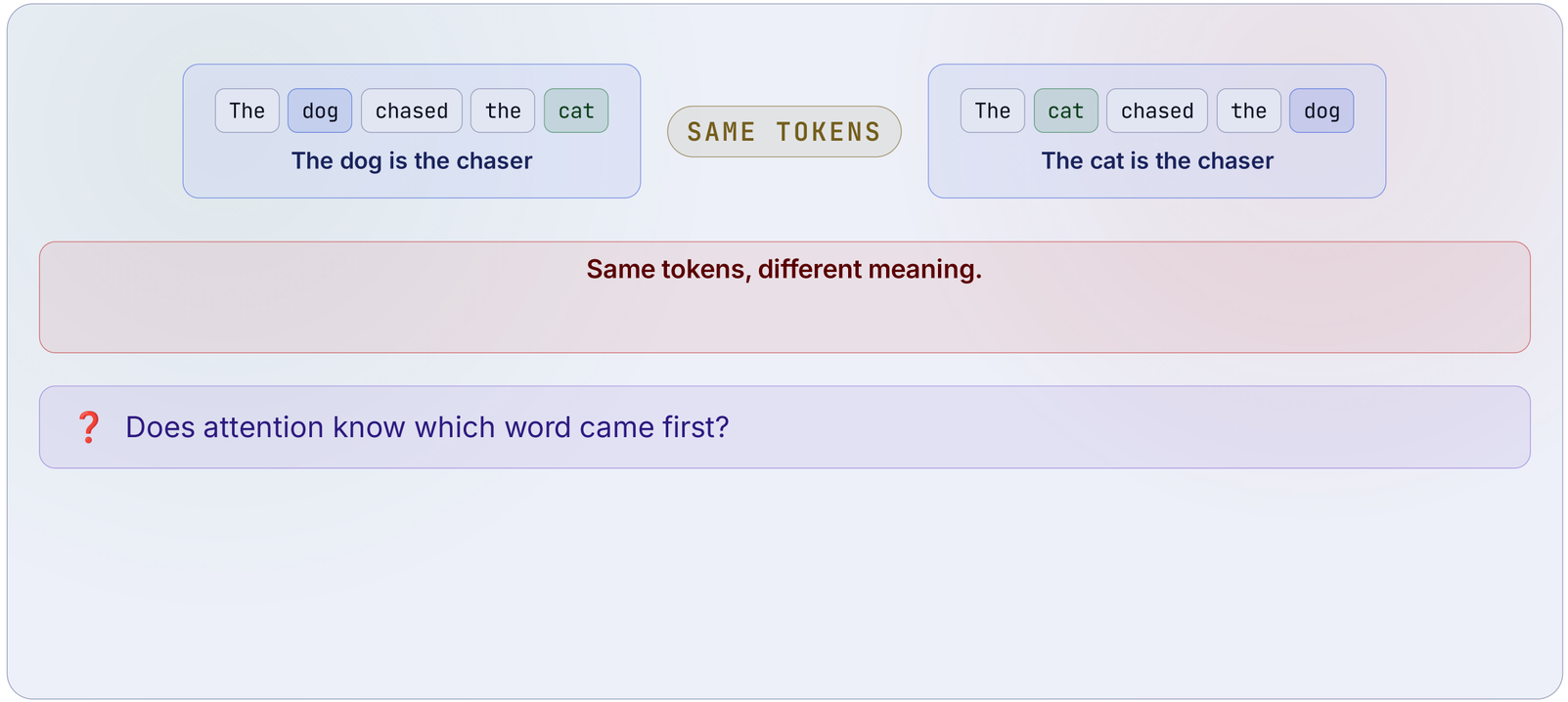


Position must be added before attention so order changes the input.

PROBLEM

Self-Attention Compares Vectors, Not Word Order

Without positional signals, self-attention is permutation equivariant: reorder the token rows in, and the outputs reorder the same way.



Position must be added before attention so order changes the input.

PROBLEM

Self-Attention Compares Vectors, Not Word Order

Without positional signals, self-attention is permutation equivariant: reorder the token rows in, and the outputs reorder the same way.

Same tokens, different meaning.

❓ Does attention know which word came first?

No. The layer is permutation equivariant without positions.

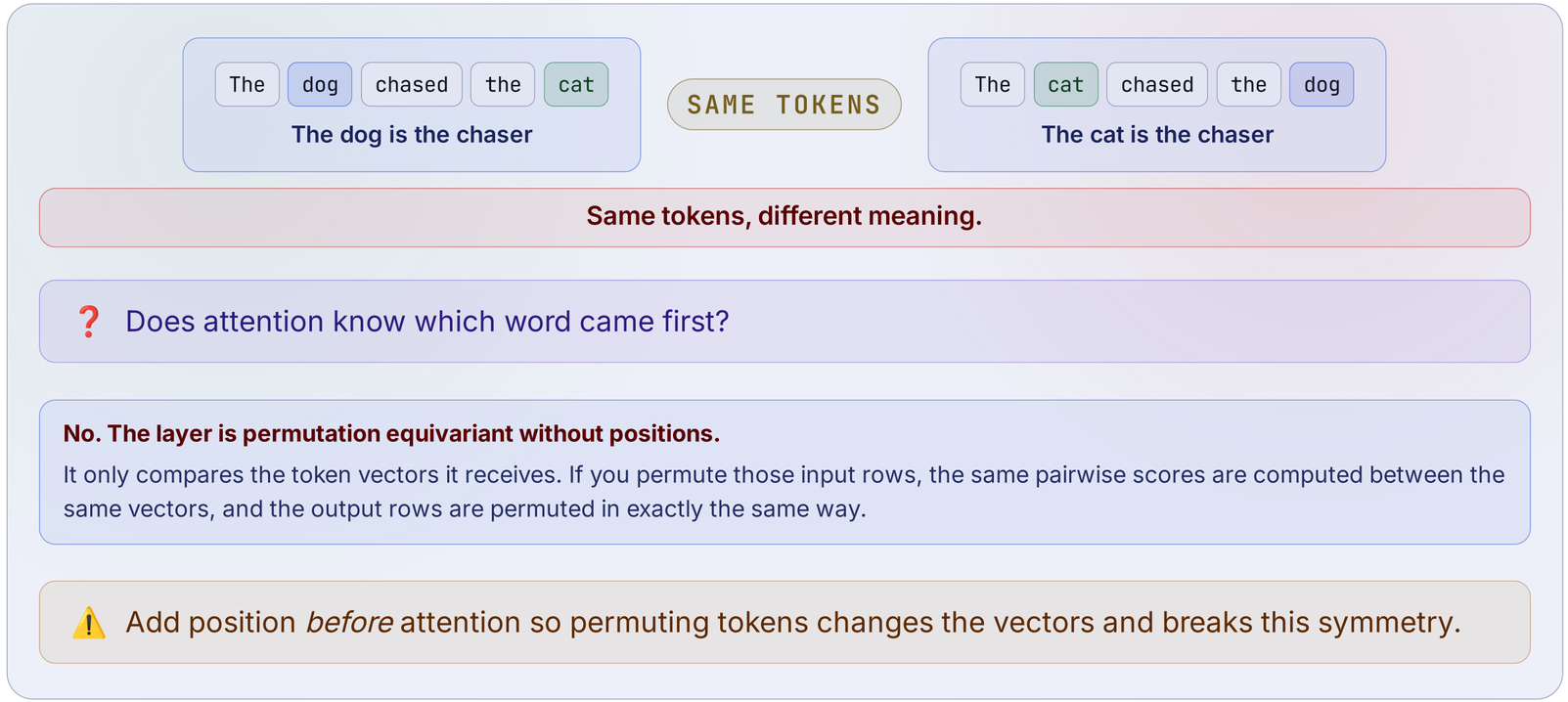
It only compares the token vectors it receives. If you permute those input rows, the same pairwise scores are computed between the same vectors, and the output rows are permuted in exactly the same way.

Position must be added before attention so order changes the input.

PROBLEM

Self-Attention Compares Vectors, Not Word Order

Without positional signals, self-attention is permutation equivariant: reorder the token rows in, and the outputs reorder the same way.



Position must be added before attention so order changes the input.

Designing Positional Encoding

Attention needs order. Build a position signal from first principles.

1 TARGET | 2 INTEGER | 3 BINARY | 4 SIN/COS | 5 RELATIVE | 6 ROPE

STEP 1 · GOALS

What must a position code do?

Start with the requirements, then choose the formula.

THREE DESIGN GOALS

1. **Distinct slots**
   Slot 1 and slot 100 must look different.
2. **Stable meaning**
   Position 5 should mean the same in short and long contexts.
3. **Local smoothness**
   Nearby slots should get nearby codes.

token embedding x_i + position code p_i =

input to attention $h_i^{(0)}$

WHY THIS MATTERS

The | dog | chased | another | dog

Both **dog** tokens share an embedding; position distinguishes them.

Good position codes are distinct, length-stable, and locally smooth.

DESIGN JOURNEY

Designing Positional Encoding

Attention needs order. Build a position signal from first principles.

1 TARGET | 2 INTEGER | 3 BINARY | 4 SIN/COS | 5 RELATIVE | 6 ROPE

STEP 2 · RAW INDEX

Why not just add the index?

A raw index is too large, too uniform, and length-dependent.

WHY RAW INDICES FAIL

A Scale mismatch
252 overwhelms embedding values near 0.

B Same scalar everywhere
One scalar across every dimension adds almost no structure.

C Length-dependent meaning
Normalizing makes the same slot mean different things.

SCALE MISMATCH

SHORT SEQUENCE
$5/10 = 0.5$

LONG SEQUENCE
$5/10000 = 0.0005$

Raw indices overwhelm embeddings and change meaning across context lengths.

Designing Positional Encoding

Attention needs order. Build a position signal from first principles.

STEP 3 · BINARY

Binary shows the right pattern, but it jumps

Binary keeps values bounded and multi-rate, but nearby slots can still change abruptly.

WHAT BINARY GETS RIGHT

A Bounded values
The code stays small.

B Multiple scales
Some bits flip fast, others slowly.

C Not smooth
One step can flip several bits at once.

BINARY CODE ACROSS POSITIONS

Position	0	1	2	3	4	5	6	7
Bit 0 FAST	0	1	0	1	0	1	0	1
Bit 1 MEDIUM	0	0	1	1	0	0	1	1
Bit 2 SLOW	0	0	0	0	1	1	1	1

WHY IT REMAINS HARD TO LEARN

- Adjacent slots can still flip multiple bits at once.

Binary shows multiscale structure, but its jumps are too abrupt.

Designing Positional Encoding

Attention needs order. Build a position signal from first principles.

1 TARGET | 2 INTEGER | 3 BINARY | 4 SIN/COS | 5 RELATIVE | 6 ROPE

STEP 4 · SIN/COS

Sinusoids keep the scales and smooth the changes

Different dimensions vary at different speeds, but nearby positions stay close.

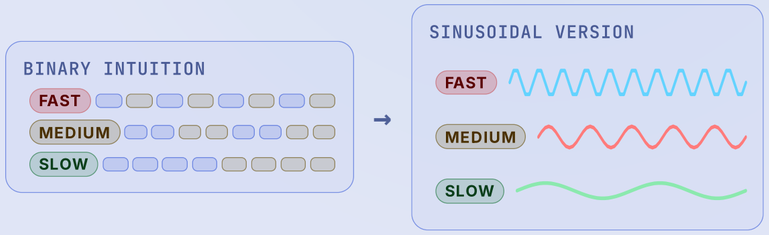
WHAT SINUSOIDS IMPROVE

A One dimension, one wave
Each row changes smoothly across position.

B Multiple scales
Early rows move fast; later rows drift slowly.

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

ROWS = DIMENSIONS, X-AXIS = POSITION



Sinusoids keep multiscale structure while staying smooth.

Designing Positional Encoding

Attention needs order. Build a position signal from first principles.

1 TARGET | 2 INTEGER | 3 BINARY | 4 SIN/COS | 5 RELATIVE | 6 ROPE

STEP 5 · RELATIVE

Attention wants relative gap in the score

Absolute position says where a token sits. Attention scores often need how far apart the compared tokens are.

WHAT ATTENTION NEEDS

A Absolute vs. relative

Absolute position answers “which slot is this?” Relative position answers “how far is this token from the token it is being compared against?”

B Scores are geometric

Attention compares queries and keys with a dot product, so changing the relative gap should change the angle and therefore the score.

SAME SENTENCE, TWO POSITION VIEWS

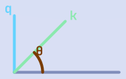
0	1	2	3	4
The	dog	chased	another	dog
-2	-1	0	+1	+2

WHY THIS SHOULD AFFECT ATTENTION

score between token m and token n:
$s_{mn} = q_m^\top k_n$

$$q_m \cdot k_n = |q_m|\,|k_n|\,\cos\theta_{mn}$$

Relative gap changes angle → angle changes score.



Step 6 needs a position scheme that makes score geometry reflect relative offset.

Attention scores come from dot products, so a useful position scheme should make score geometry depend on relative offset.

Designing Positional Encoding

Attention needs order. Build a position signal from first principles.

1 TARGET | 2 INTEGER | 3 BINARY | 4 SIN/COS | 5 RELATIVE | 6 ROPE

STEP 6 · ROPE

RoPE makes attention scores depend on relative gap

Rotate Q and K by absolute position; the score ends up depending on $(m - n)$.

HOW ROPE WORKS

A Inject position at scoring time
Rotate Q/K instead of adding a vector to the embedding.

B Keep the multiscale ladder
Each 2D pair uses its own frequency.

C Turn absolute rotations into relative offset
Same gap, same score pattern.

1. ROTATE Q AND K BY ABSOLUTE POSITION

query at position m: rotate by $m\theta_i$

key at position n: rotate by $n\theta_i$

Angle encodes the gap.

2. THE DOT PRODUCT CANCELS ABSOLUTE ROTATION

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

Same gap, same relative score anywhere.

RoPE rotates Q/K so attention scores reflect relative offset.

MECHANISM

Attention Mixes, FFN Transforms

Attention shares information across tokens. The FFN then updates each token independently.

Attention shares information across tokens. The FFN updates each token independently and usually holds most of the parameters.

MECHANISM

Attention Mixes, FFN Transforms

Attention shares information across tokens. The FFN then updates each token independently.

ATTENTION
Mixes information **across** tokens
Cross-token context

→

FFN
Transforms **each** token independently
Per-token transformation

PER TOKEN: EXPAND, ACTIVATE, CONTRACT

token state d_{model} → expand 4× W_1 → GELU activate → contract W_2 → updated state d_{model}

$$\text{FFN}(h_i) = W_2\,\text{GELU}(W_1 h_i + b_1) + b_2$$

The same small network runs at every position: shared weights, different inputs.

PARAMETER REALITY

Per block, attention is roughly $4d_{\text{model}}^2$ parameters, while the FFN is roughly $8d_{\text{model}}^2$. That is why, in large dense LLMs, the FFN usually outweighs attention in total parameters.

MODEL	TOTAL	ATTENTION	FFN	OTHER
GPT-3 13B	**12.95B**	**4.23B** 32.6%	**8.46B** 65.3%	**0.27B** 2.1%
GPT-3 175B	**174.60B**	**57.99B** 33.2%	**115.97B** 66.4%	**0.65B** 0.4%

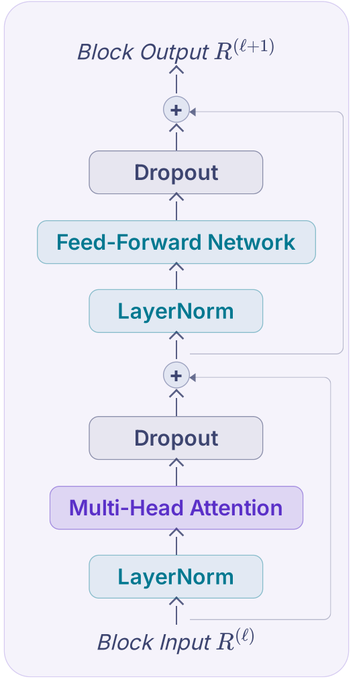
Other = token embeddings + positional embeddings + LayerNorm parameters.

Attention shares information across tokens. The FFN updates each token independently and usually holds most of the parameters.

SYNTHESIS

One Transformer Block, Revisited

Same diagram as slide 3. This time, every box has a clear role.



READ THE BLOCK BOTTOM TO TOP

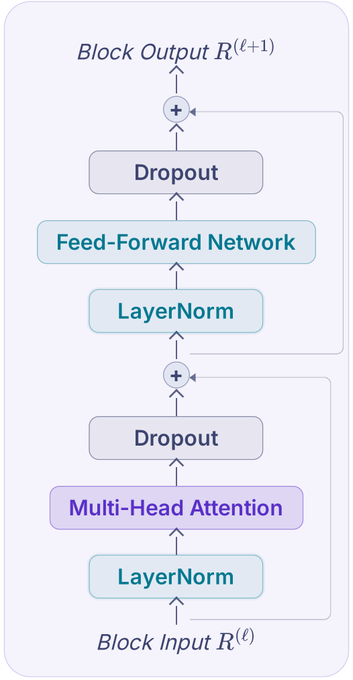
LayerNorm → attention → residual add → LayerNorm → FFN → residual add. Both sublayers write into the same residual stream.

One block = LayerNorm, attention, residual add, LayerNorm, FFN, residual add.

SYNTHESIS

One Transformer Block, Revisited

Same diagram as slide 3. This time, every box has a clear role.



READ THE BLOCK BOTTOM TO TOP

LayerNorm → attention → residual add → LayerNorm → FFN → residual add. Both sublayers write into the same residual stream.

LAYERNORM

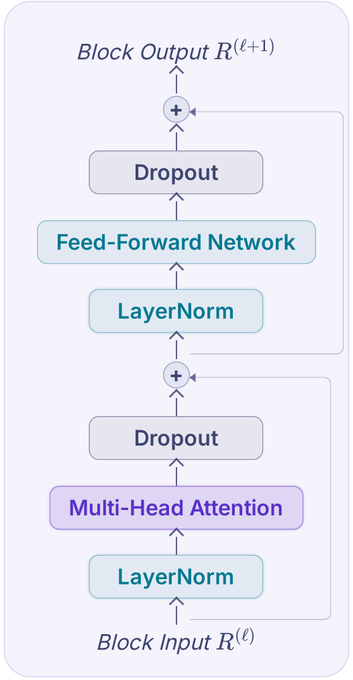
Normalizes each row before the sublayer reads it.

LayerNorm normalizes each row before a sublayer reads it.

SYNTHESIS

One Transformer Block, Revisited

Same diagram as slide 3. This time, every box has a clear role.



READ THE BLOCK BOTTOM TO TOP

LayerNorm → attention → residual add → LayerNorm → FFN → residual add. Both sublayers write into the same residual stream.

LAYERNORM

Normalizes each row before the sublayer reads it.

MULTI-HEAD ATTENTION

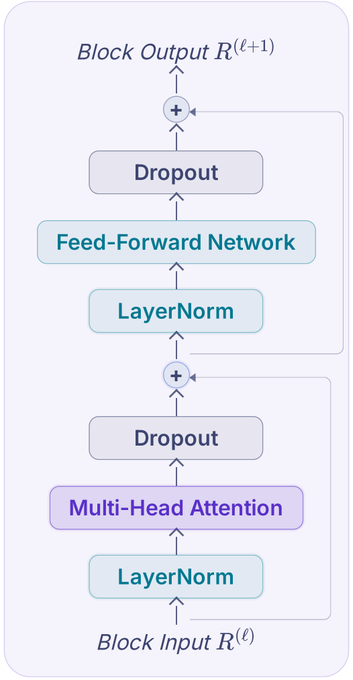
Attention mixes information across earlier tokens and writes a context update.

Attention mixes across tokens and writes a context update.

SYNTHESIS

One Transformer Block, Revisited

Same diagram as slide 3. This time, every box has a clear role.



READ THE BLOCK BOTTOM TO TOP

LayerNorm → attention → residual add → LayerNorm → FFN → residual add. Both sublayers write into the same residual stream.

LAYERNORM

Normalizes each row before the sublayer reads it.

MULTI-HEAD ATTENTION

Attention mixes information across earlier tokens and writes a context update.

RESIDUAL ADD (+)

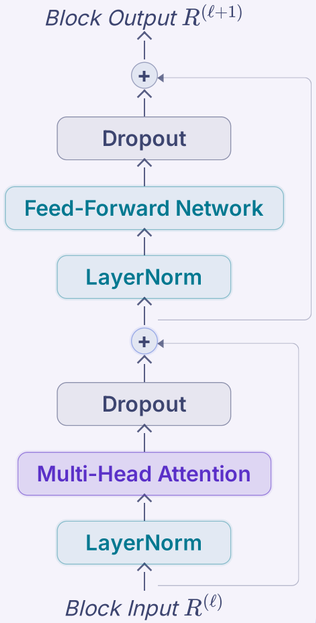
Writes the attention update back into the same stream.

Residual add writes that update back into the same stream.

SYNTHESIS

One Transformer Block, Revisited

Same diagram as slide 3. This time, every box has a clear role.



READ THE BLOCK BOTTOM TO TOP

LayerNorm → attention → residual add → LayerNorm → FFN → residual add. Both sublayers write into the same residual stream.

LAYERNORM

Normalizes each row before the sublayer reads it.

MULTI-HEAD ATTENTION

Attention mixes information across earlier tokens and writes a context update.

RESIDUAL ADD (+)

Writes the attention update back into the same stream.

FEED-FORWARD NETWORK

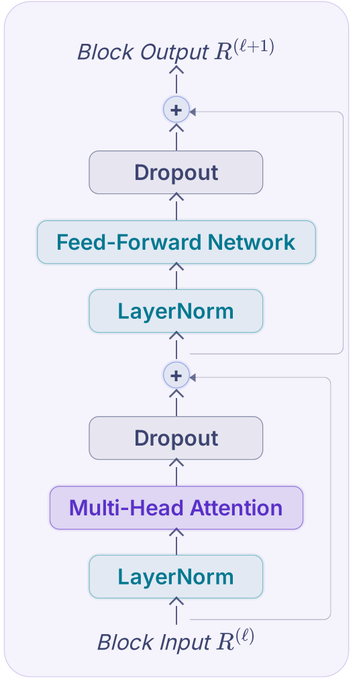
The same two-layer MLP updates each row independently.

The FFN updates each row independently, then writes its own residual update.

SYNTHESIS

One Transformer Block, Revisited

Same diagram as slide 3. This time, every box has a clear role.



READ THE BLOCK BOTTOM TO TOP

LayerNorm → attention → residual add → LayerNorm → FFN → residual add. Both sublayers write into the same residual stream.

LAYERNORM

Normalizes each row before the sublayer reads it.

MULTI-HEAD ATTENTION

Attention mixes information across earlier tokens and writes a context update.

RESIDUAL ADD (+)

Writes the attention update back into the same stream.

FEED-FORWARD NETWORK

The same two-layer MLP updates each row independently.

TWO UPDATE EQUATIONS

$$R_{\text{mid}} = R^{(\ell)} + \text{MHA}(\text{LN}(R^{(\ell)}))$$

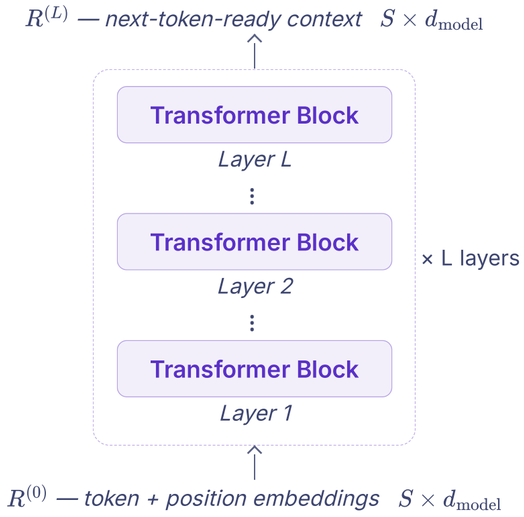
$$R^{(\ell+1)} = R_{\text{mid}} + \text{FFN}(\text{LN}(R_{\text{mid}}))$$

A full model stacks this same Transformer block layer after layer.

CONCEPT

Same Block, More Depth

The block repeats L times. The shape stays $S \times d_{\text{model}}$; each layer further refines the residual stream.



WHAT DEPTH MEANS

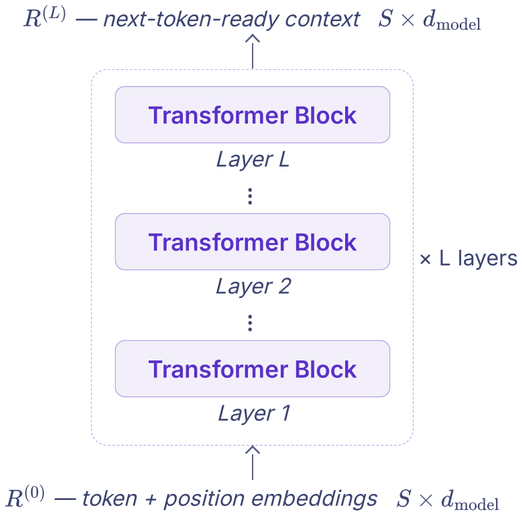
Each layer runs the same transformer block again. The sequence shape stays fixed; the representation becomes more refined.

Depth keeps the same sequence shape and refines meaning layer by layer.

CONCEPT

Same Block, More Depth

The block repeats L times. The shape stays $S \times d_{\text{model}}$; each layer further refines the residual stream.



WHAT DEPTH MEANS

Each layer runs the same transformer block again. The sequence shape stays fixed; the representation becomes more refined.

One block = one refinement pass — attention and FFN read the current stream and write back an update.

Each new block starts from the updated stream — later layers build on what earlier layers already produced.

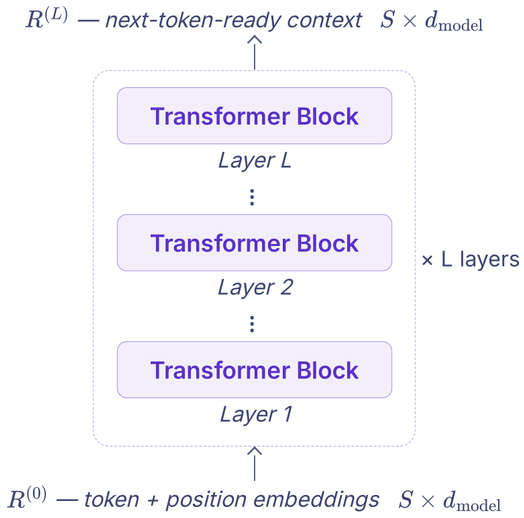
Depth adds repeated computation — the model gets multiple chances to combine and refine information.

Depth keeps the same sequence shape and refines meaning layer by layer.

CONCEPT

Same Block, More Depth

The block repeats L times. The shape stays $S \times d_{\text{model}}$; each layer further refines the residual stream.



WHAT DEPTH MEANS

Each layer runs the same transformer block again. The sequence shape stays fixed; the representation becomes more refined.

One block = one refinement pass — attention and FFN read the current stream and write back an update.

Each new block starts from the updated stream — later layers build on what earlier layers already produced.

Depth adds repeated computation — the model gets multiple chances to combine and refine information.

📖 GPT-2 small uses **12** layers. GPT-3 uses **96**. Same block, more refinement passes.

Depth keeps the same sequence shape and refines meaning layer by layer.

Predict One Token, Append, Repeat

Take the last hidden state, score the vocabulary, form a distribution, then decode one token.

CONTEXT

The
cat
sat
on
the

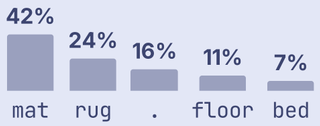
Only the final hidden state scores the next token: $h_t^{(L)}$

→

LM HEAD

$$\ell = W_U \, h_t^{(L)}$$

softmax → next-token distribution



The model produces scores; decoding selects the next token.

→

GREEDY DECODING (ARGMAX)

Always choose the highest-probability token.

mat

One full forward pass produces scores over the vocabulary.

Predict One Token, Append, Repeat

Take the last hidden state, score the vocabulary, form a distribution, then decode one token.

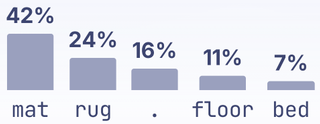
Greedy decoding (argmax) is one way to select the next token.

From Probabilities to Behavior

The model outputs one distribution. Decoding determines how that distribution becomes an actual token.

ONE DISTRIBUTION

$\ell = W_U\, h_t^{(L)}$ → softmax → **next-token distribution**



Same distribution. Different selection rule.

Greedy (argmax)

Always pick the highest probability token.

CHOSEN TOKEN `mat`

Temperature Sampling

Adjust sharpness, then sample.

EXAMPLE SAMPLE `rug`

Top-k Sampling

Restrict the candidate set, then sample.

KEEP `mat` `rug` `.`

EXAMPLE SAMPLE `.`

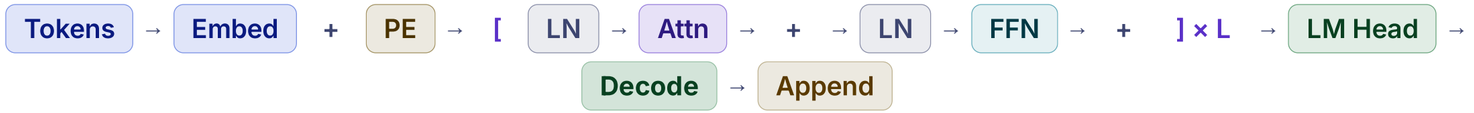
The transformer produces scores. Decoding turns those scores into text.

The distribution stays fixed; decoding changes how the next token is chosen.

CLOSING

You Now Know the Transformer

This is the full loop.



↩ autoregressive loop: append and run again

Each new token changes the context → the next distribution changes

Tokenize → Represent → Understand × L → Predict → Decode → Append → Repeat or Stop

Stop at EOS, a stop sequence, or the max token limit

Tokenize → Represent → Understand × L → Predict → Decode → Append → Repeat or Stop.

You Now Know the Transformer

This is the full loop.

Tokens → Embed + PE → [LN → Attn → + → LN → FFN → +] × L → LM Head → Decode → Append

↩ *autoregressive loop: append and run again*

Each new token changes the context → the next distribution changes

Tokenize → Represent → Understand × L → Predict → Decode → Append → Repeat or Stop

Stop at EOS, a stop sequence, or the max token limit

Tokenize

The model reads subword units, not whole words. Vocabulary boundaries shape what it can represent cleanly.

Represent

Embeddings plus position turn tokens into vectors the model can compare and update.

Understand

Stacked blocks use attention and FFN to refine the residual stream while keeping sequence shape fixed.

Predict + Decode

The LM head scores the vocabulary. Decoding selects one token, appends it, and the new context changes the next distribution.

✓ From tokenization through decoding, this is the forward pass that drives generation.

This is the full forward pass, from tokenization to generation.