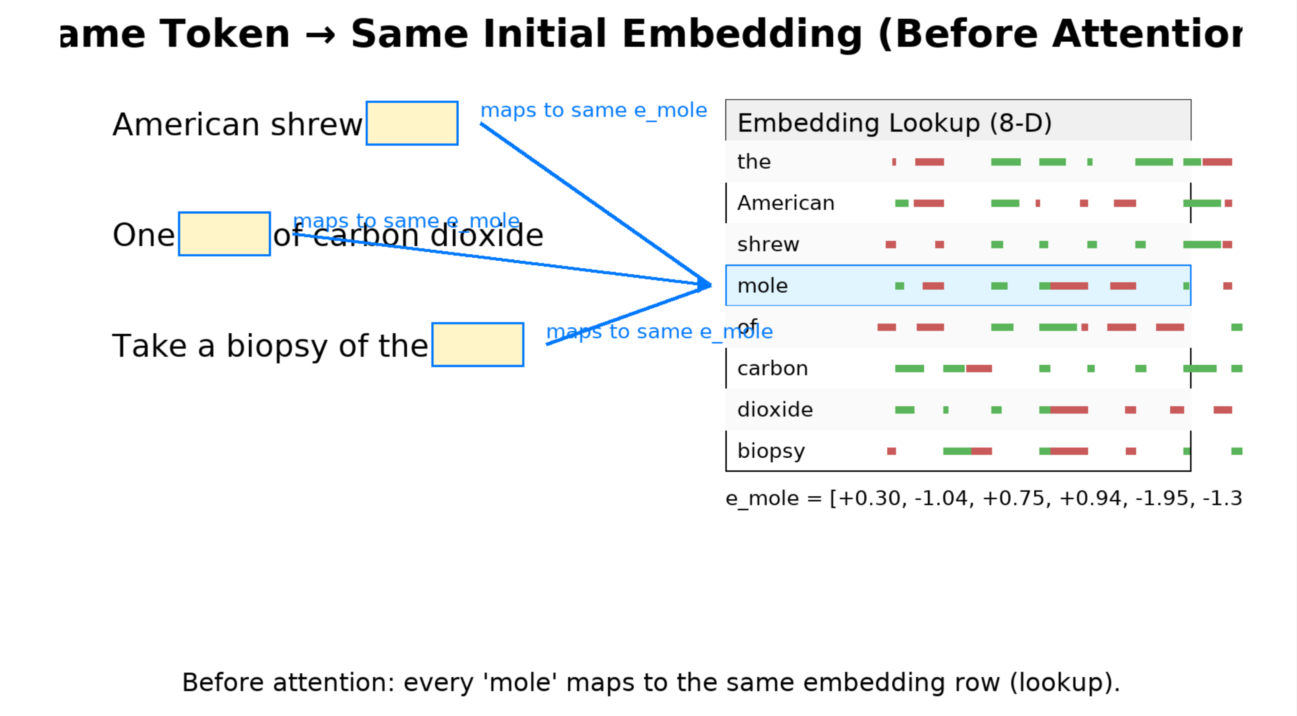
Attention

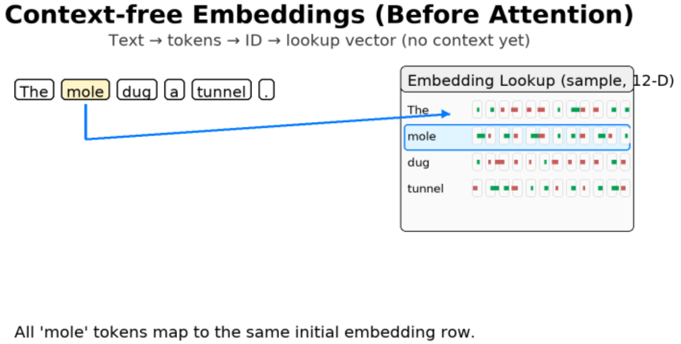
- By Polisetty Sai Teja

Recap

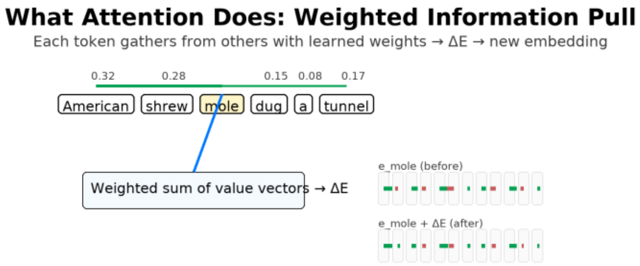
- **Embeddings first:** Transformers turn text into high-dimensional vectors called *embeddings*.
- **Lookup, not meaning (yet):** At this stage, an embedding is a learned lookup — the token **"mole"** maps to the same vector in every sentence.
- **What transformers add:** Through **attention**, the model reads surrounding words and **steers** each token's vector toward the meaning required by its context.
- **Why it matters:** Context-aware vectors help the model both **understand neighbors** and **predict the next token** more accurately.



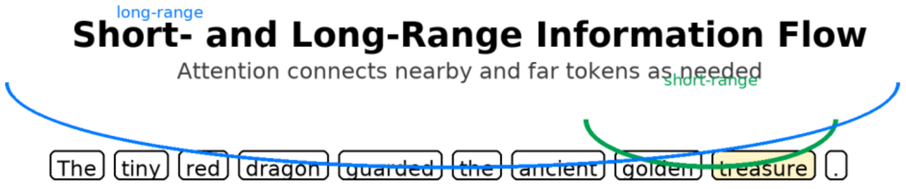
Core Idea of Attention



Start point: Tokens begin as context-free embeddings (look up vectors).

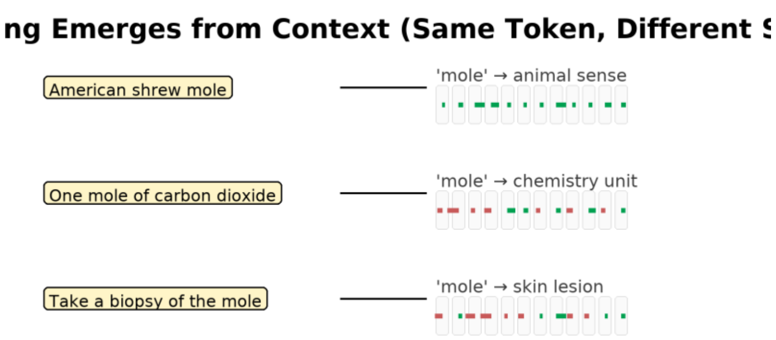


What attention does: Each token computes how much to pay attention to every other token and pulls in their information as a weighted sum—producing a context-specific update (ΔE) to its embedding.

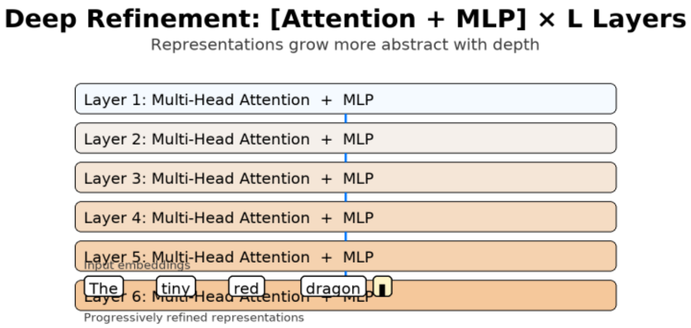


Short & long range: These weights connect both nearby words and far-apart words, so information can jump across the sequence(short- and long-distance dependencies).

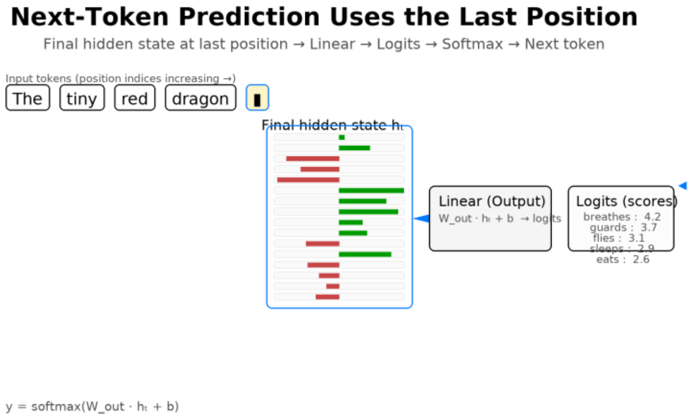
Core Idea of Attention



Meaning via context: The same token (e.g., *mole*) is steered to different regions of the space depending on its neighbours—animal vs chemistry unit vs skin lesion.



Deep refinement: Repeat in attention (plus MLPs) over many layers progressively sharpens meaning.



- **Prediction:** For next-token prediction, the model uses the **final, last-position representation** produced after all layers.

Single-Head Attention - What & Why

Goal: Let a token "borrow" the right info from other tokens.

Projections: $Q_i = E_i W_Q,\; K_i = E_i W_K,\; V_i = E_i W_V$

Scores: $s_{ij} = \frac{Q_i \cdot K_j}{\sqrt{d_k}}$ (apply causal **mask** if predicting left→right)

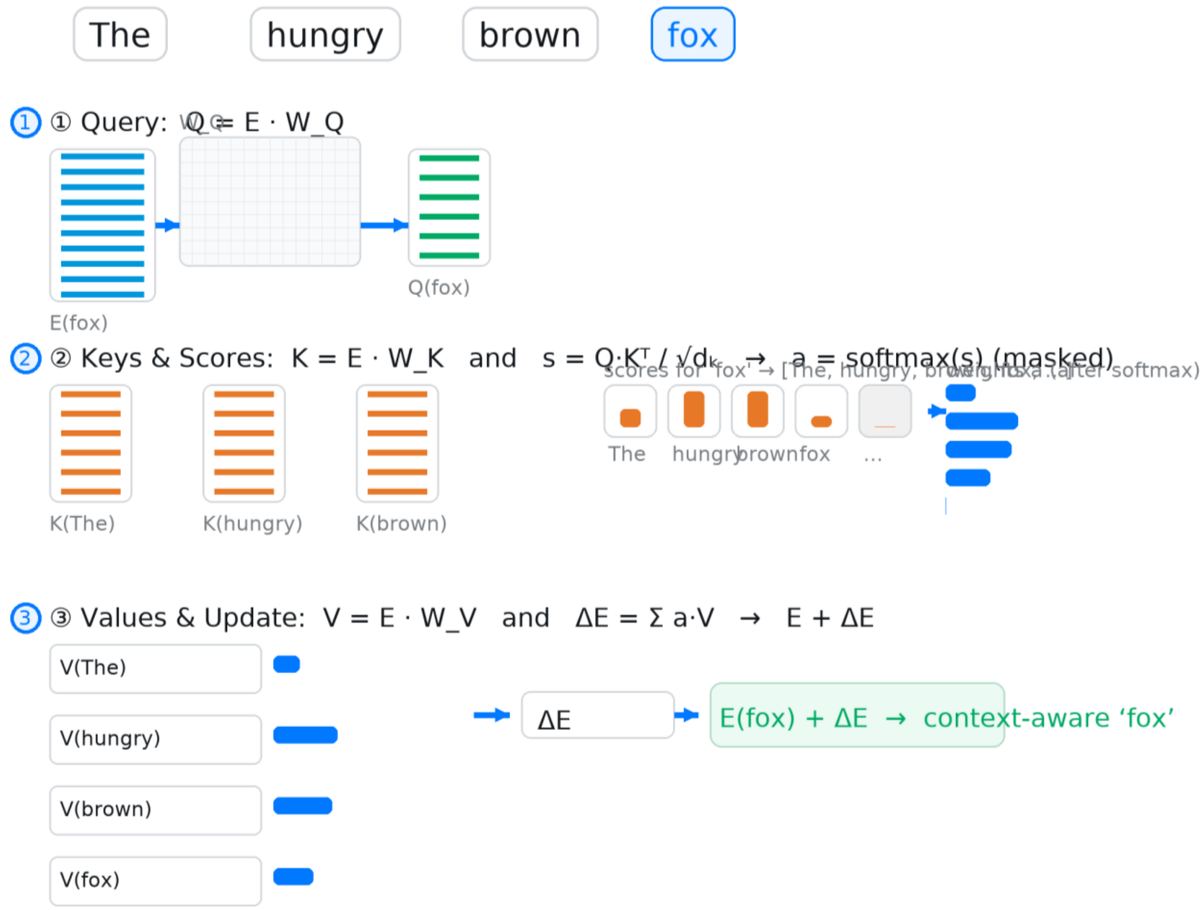
Weights: $a_{ij} = \mathrm{softmax}_j(s_{ij})$

Update: $\Delta E_i = \sum_j a_{ij} V_j;\; \mathrm{output} = E_i + \Delta E_i$

Intuition: Q = what I'm looking for, K = what I offer, V = what to send.

Single-Head Attention - Cont...

- **Step 1** — Build the Query (Q)
  - Target token: "fox" in "The hungry brown fox jumped over the sleepy dog."
- **Step 2:** Keys ($K\ i$) + "I'm an adjective" tagging
- **Step 3:** Scores(Dot product Q·K → relevance score) + scaling (attention weights - probability distribution) + causal mask(prevents future words from leaking information) → weights
- **Step 4:** Weighted sum of Values → ΔE4 → updated 'fox' - **attention pattern grid**
- **Step 5:** Update the embedding.



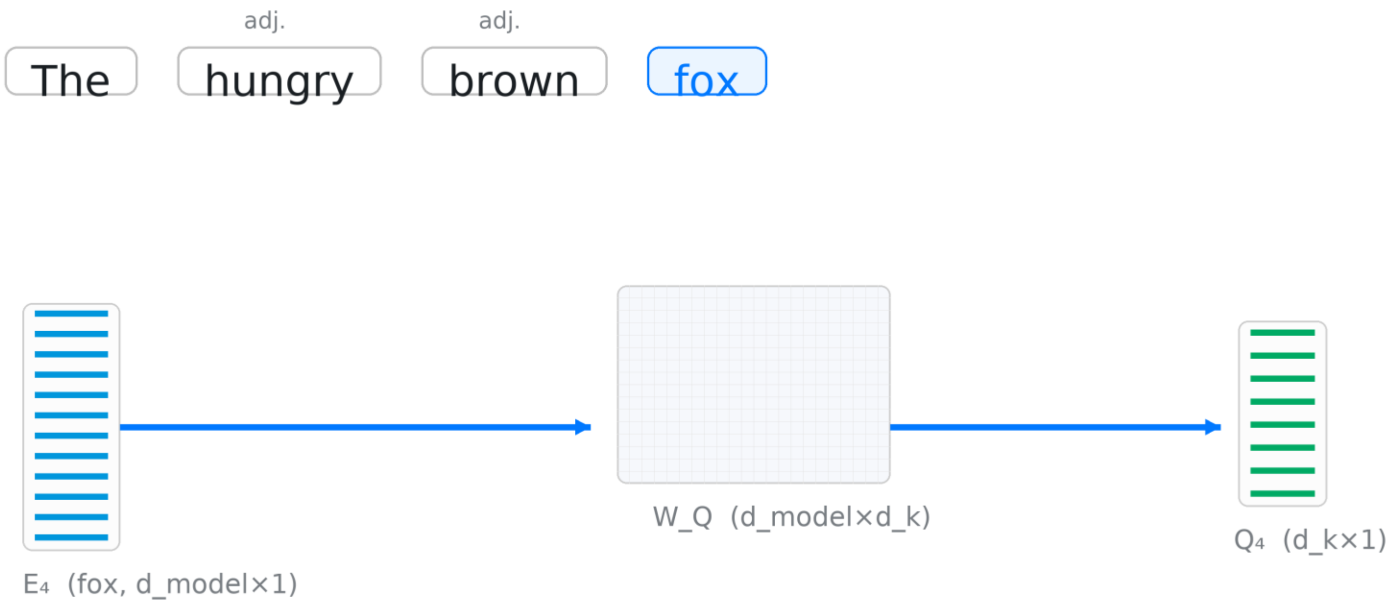
S1: Query Vector

- Idea: At "fox", the model forms a question: "Are there adjectives before me?"

- **Operation:** $Q_4 = E_4 \cdot W_Q$ projects the "fox" embedding into a **query space** (d_k e.g., 128)

- **Why:** Q_4 encodes what "fox" should look for (modifiers like **hungry**, **brown**).

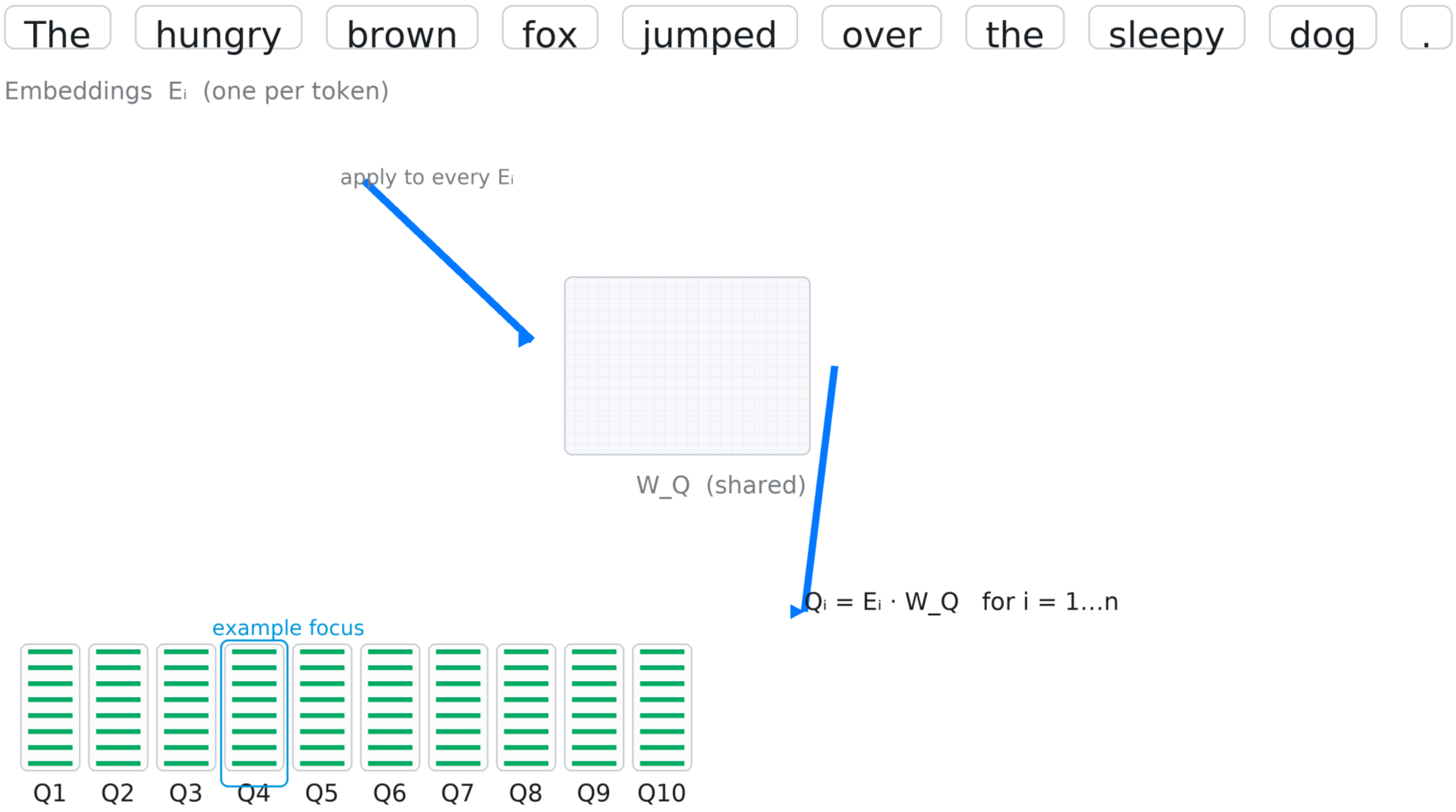
- **Scope:** We compute a **query Qi** for **every** token; we're highlighting Q4 only for clarity.



Build the query for the focus token: $Q_4 = E_4 \cdot W\_Q$

- By Sai Teja Polisetty

Query vector - cont....



Shared projection W_Q maps every embedding E_i into the query space → Q_i (smaller d_k).

Key vector

Idea: Each token advertises what it is. For our head, adjectives like hungry, brown, sleepy emit a "I'm an adjective" signal so nouns (e.g., fox, dog) can find them.

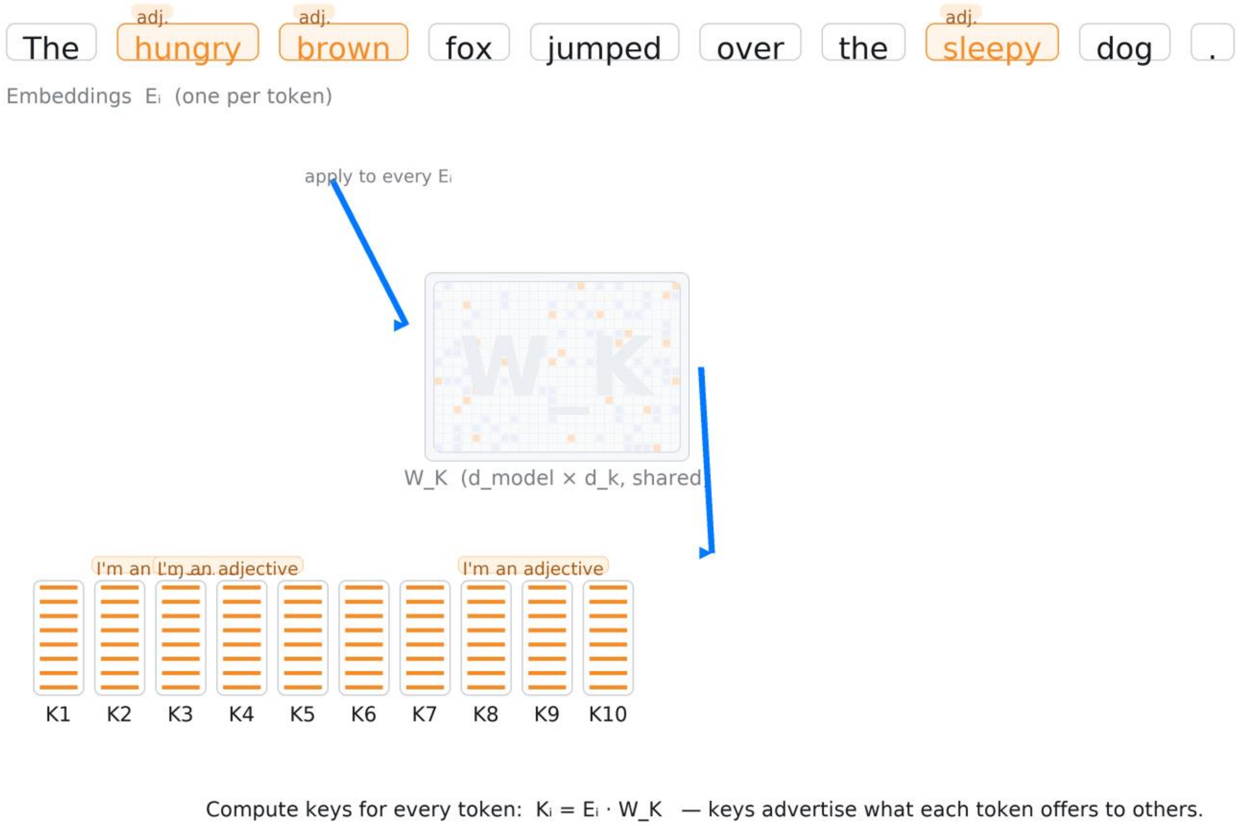
Operation: $K_i = E_i \cdot W_k$

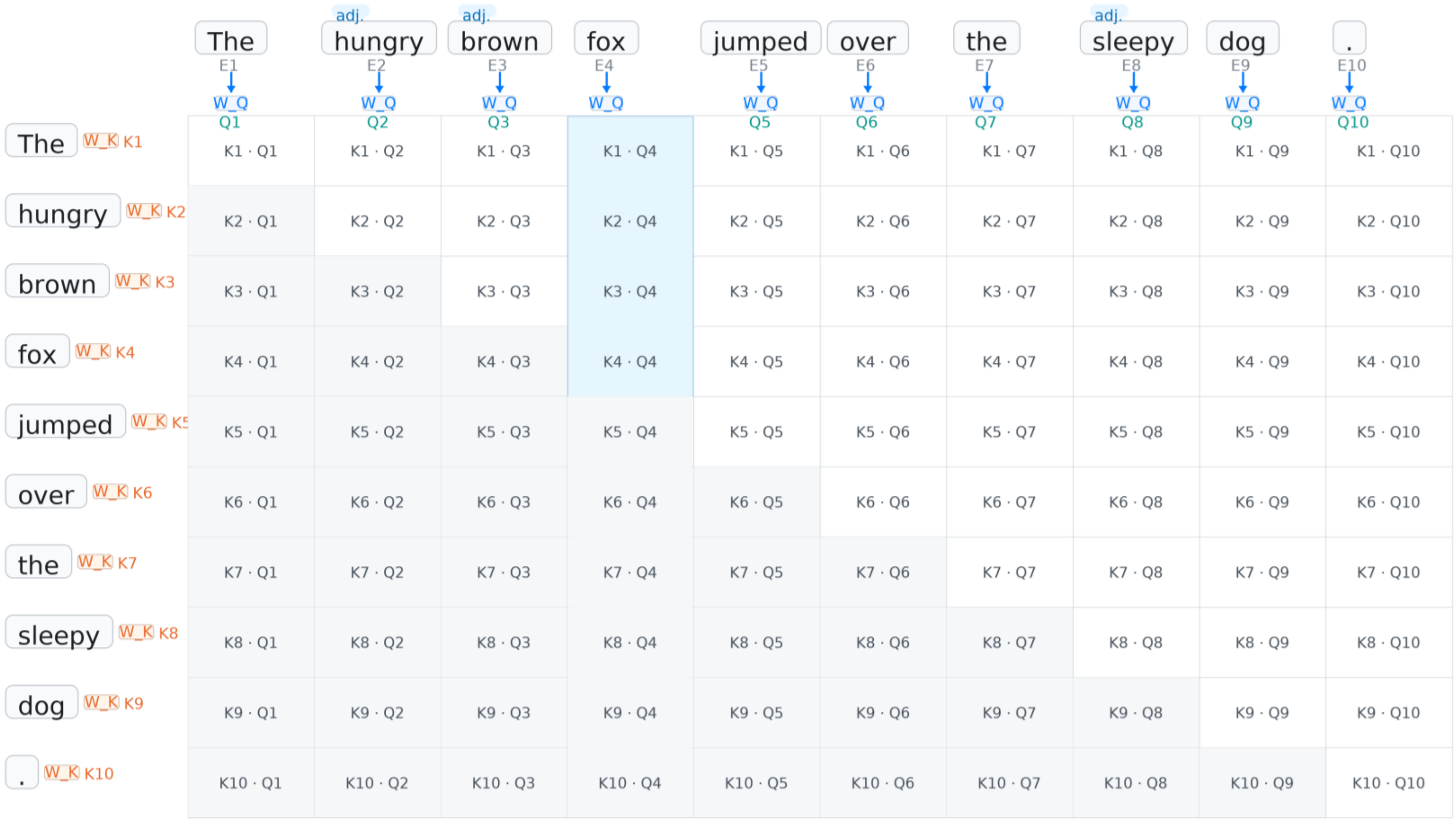
- E_i : embedding of token I (size d model)
- W_k: learned, shared projection matrix ($d_{model} * d_k$)
- K_i: key in the same dk space as queries, so dot products Q.K are comparable

Why: Queries need something to match against. Keys are the comparators—learned directions that make the "**right" tokens align with the current token's intent** (e.g., a noun's query aligns with adjective keys). Without keys, the model couldn't score who to attend to.

Scope

- We compute a key Ki for every token using the same WK
- Different heads learn different "advertisements" (adjective-ness, subject-verb links, coreference, etc.). This step is independent of masking; masking affects scoring later, not key creation.





S2: Key vector

Conceptually, Key vector is **answer** of query vector when both are in the same direction.

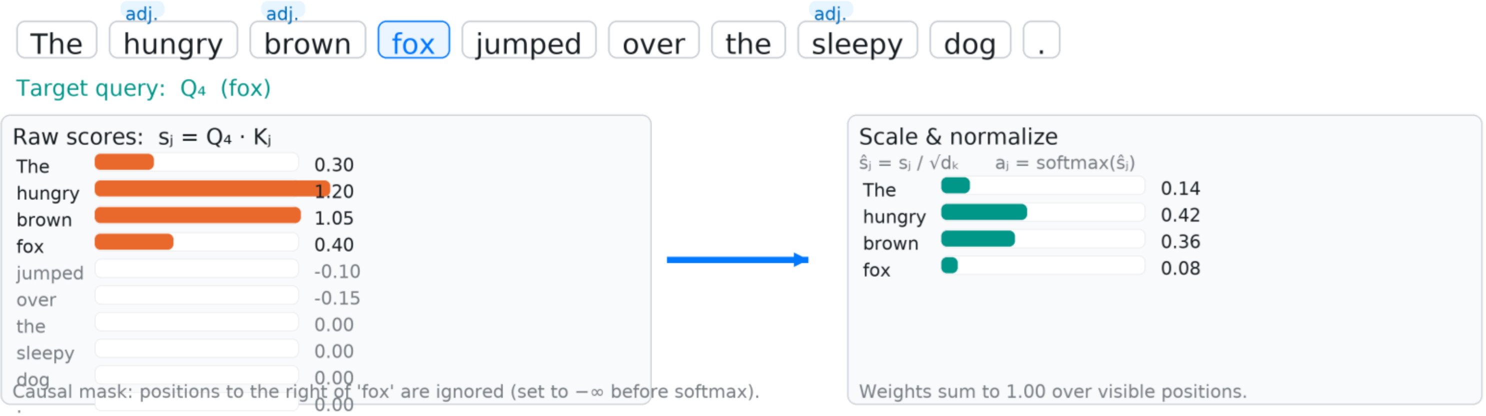
S3: Scores -> Mask -> Weights (queries vs keys)

- **Idea**: Turn "what I'm looking for" (Q) and "what I offer" (K) into match scores, then mask future positions, and finally apply softmax to get attention weights.
- **Operation**:
  - i is the current position (here we illustrate i=4 the word "fox").
  - Mask all positions j>i (future tokens) before softmax in a decoder block.

$$s_j = \frac{Q_i \cdot K_j}{\sqrt{d_k}}$$

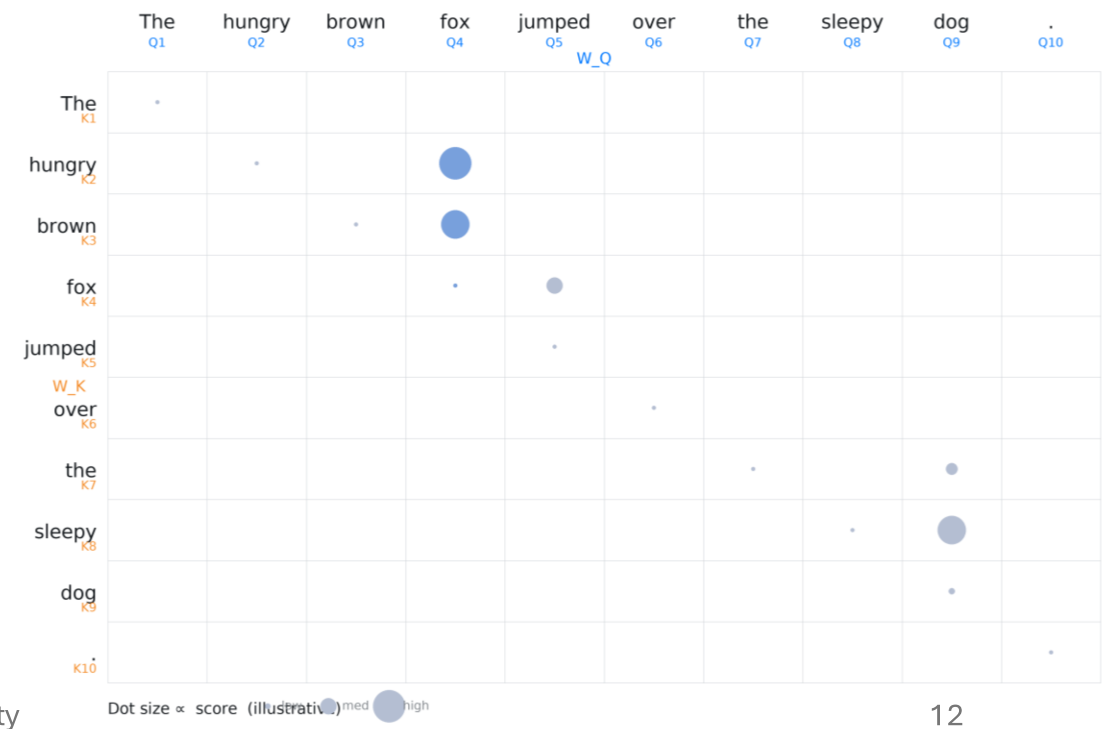
$$a_j = \mathrm{softmax}(s)_j$$

	The Q1	hungry Q2	brown Q3	fox Q4	jumped Q5	over Q6	the Q7	sleepy Q8	dog Q9	. Q10
The K1	0.00	0.00	0.00	0.30	0.00	0.00	0.00	0.00	0.00	0.00
hungry K2	0.00	0.00	0.00	1.20	0.00	0.00	0.00	0.00	0.00	0.00
brown K3	0.00	0.00	0.00	1.05	0.00	0.00	0.00	0.00	0.00	0.00
fox K4	0.00	0.00	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
jumped K5	0.00	0.00	0.00	-0.10	0.00	0.00	0.00	0.00	0.00	0.00
over K6	0.00	0.00	0.00	-0.15	0.00	0.00	0.00	0.00	0.00	0.00
the K7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
sleepy K8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
dog K9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
. K10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00



Step 3: scores → mask → softmax. Example for Q_4 ('fox'): focus lands on adjectives 'hungry' and 'brown'.

ATTENTION PATTERN



Why

- The dot product measures **alignment** between the current token's intent and other tokens' signals.
- Scaling by sqrt(d_k) stabilizes training.
- Softmax produces a **probability distribution** (weights) over allowed positions that sum to 1.

Scope

- Do this for **every position** i.
- Masking policy depends on architecture: **causal** (decoder-only) vs **bidirectional** (encoders don't mask).

- By Sai Teja Polisetty

** Context Size

- size of attention pattern is same as that of context size.
- Hence context is not scalable, but to work on it different mechanics came into picture like
  - Sparse attention mechanism
  - Blockwise attention
  - Linformer
  - Reformer
  - Ring attention
  - Long former
  - Adaptive attention span

S4 : Values -> ΔE -> Updated Embedding

- Idea: Use the attention **weights** to mix **value vectors** V_j . The result is a **context vector** C_i that captures what the current token needs from others. Project it (optional W_O) and **add** to the original embedding.

- Operation:

$$
\begin{aligned}
V_j &= E_j W_V \\
C_i &= \sum_j a_{ij} V_j \\
\Delta E_i &= C_i W_O \quad \text{(optional head output proj)} \\
E_i' &= E_i + \Delta E_i \quad \text{(then LayerNorm in the block)}
\end{aligned}
$$

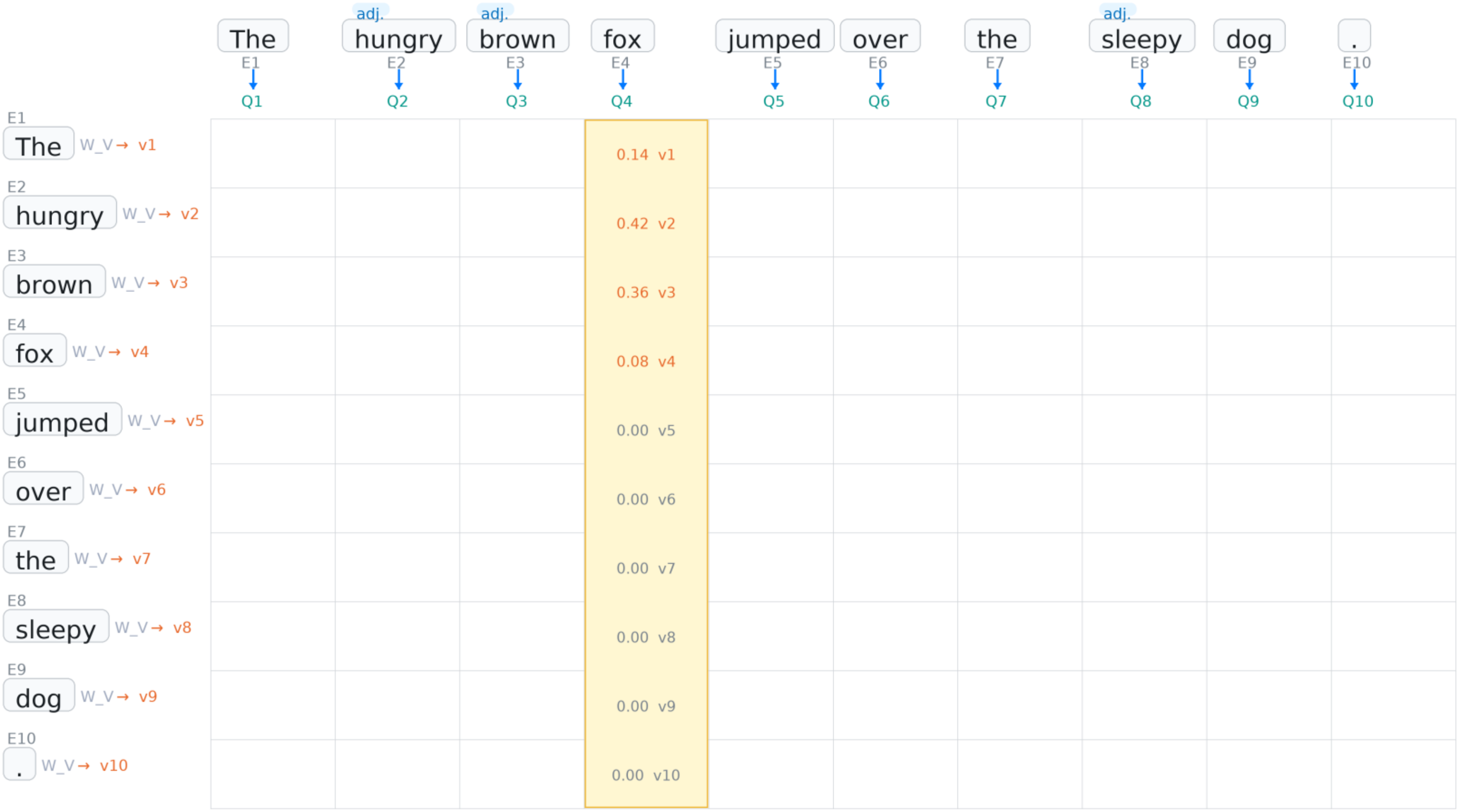
Why

Values carry the information to transfer (e.g., the adjectives' descriptors). The weights decide how much of each value flows to the current position. Adding ΔEi steers the embedding in a context-aware direction.

Scope

This happens for every token and for every head in the layer. Multi-head outputs are concatenated and projected before the residual add & LayerNorm.

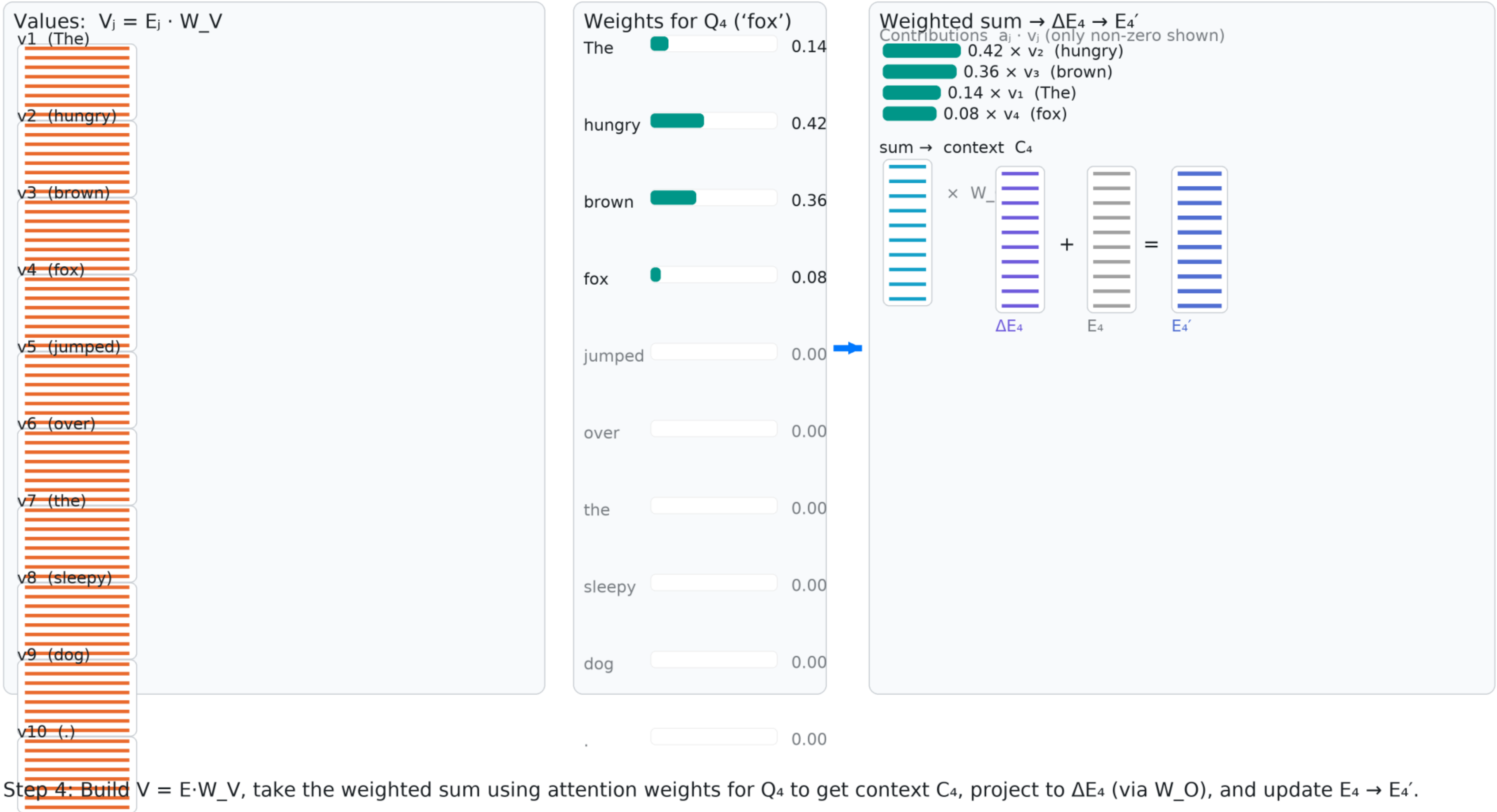
Step 4 (clarified): sum the value vectors with the attention weights to form C_4, project to ΔE_4, and add to $E_4 \rightarrow E_4'$.



Value matrix (white): each row forms $v_i = E_i \cdot W\_V$ (left). The highlighted ‘fox’ column shows its attention weights × value labels.

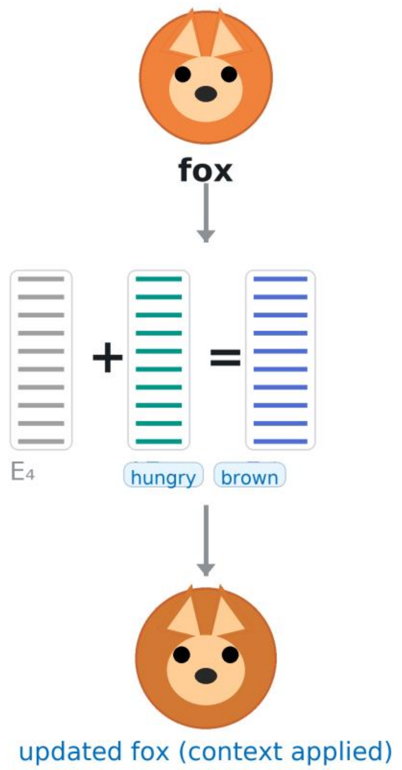
When we multiply **VALUE MATRIX** with the embedding, we can think of it as saying, if this word is relevant to adjusting the meaning of something else, what exactly should be added to embedding of something else.

**Keys and queries are not needed once u get the attention pattern.



Step 4: Build V = E·W_V, take the weighted sum using attention weights for Q_4 to get context C_4, project to ΔE_4 (via W_O), and update $E_4 \rightarrow E_4'$.

E4 = E4 + ΔE4



Attention adds ΔE₄ (from hungry/brown), giving E₄′ — a co

	The E1	hungry E2	brown E3	fox E4	jumped E5	over E6	the E7	sleepy E8	dog E9	. E10
The W_V	0.00 v1	0.00 v1	0.00 v1	0.14 v1	0.00 v1	0.00 v1	0.00 v1	0.00 v1	0.06 v1	0.00 v1
hungry W_		0.00 v2	0.00 v2	0.42 v2	0.00 v2	0.00 v2	0.00 v2	0.00 v2		0.00 v2
brown W_			0.00 v3	0.36 v3	0.00 v3	0.00 v3	0.00 v3	0.00 v3		0.00 v3
fox W_				0.08 v4	0.00 v4	0.00 v4	0.00 v4	0.00 v4		0.00 v4
jumped W_					0.00 v5	0.00 v5	0.00 v5	0.00 v5		0.00 v5
over W_						0.00 v6	0.00 v6	0.00 v6		0.00 v6
the W_							0.00 v7	0.00 v7	0.24 v7	0.00 v7
sleepy W_								0.00 v8	0.60 v8	0.00 v8
dog W_									0.10 v9	0.00 v9
. W_										0.00 v10
	ΔE1	ΔE2	ΔE3	ΔE4	ΔE5	ΔE6	ΔE7	ΔE8	ΔE9	ΔE10

Apply updates at every position

E1 +ΔE1 =E1E2 +ΔE2 =E2E3 +ΔE3 =E3E4 +ΔE4 =E4E5 +ΔE5 =E5E6 +ΔE6 =E6E7 +ΔE7 =E7E8

E1′ E2′ E3′ E4′ E5′ E6′ E7′ E8′

This complete process is called = **single head of attention.**

Connect to the "big model" numbers cont ....

- Total weights = 175,181,291,520
- Organized into 27,938 matrices

Matrix name	Matrix dimension
Embedding	D_embed * n_vocab 12,288 * 50,257 = 617558016
Key	12,288 * 128 = 1,572,864
Query	12,288 * 128 = 1,572,864
Value	12,288 * 12,288 (wrong) 128 * 12,288 = 1,572,864 And 12,288 * 128 = 1,572,864
Output	
Up-projection	
Down-projection	
UnEmbedding	n_vocab * D_embed 50,257 * 12288 = 617558016

Multi-Head Attention - Big Idea

- Idea
  - Single head attention * 10,000 = parallel ops
  - Each head with its own key/query/value matrices, so it can learn a **different relation** (e.g., adjectives →nouns, subject → verb, long-range links).
  - GPT-3 has 96 attention heads inside each block = 96 attention patterns

Operation:

For each head h:

$$Q^{(h)} = E\,W_Q^{(h)}, \quad K^{(h)} = E\,W_K^{(h)}, \quad V^{(h)} = E\,W_V^{(h)}$$

$$A^{(h)} = \mathrm{softmax}\left(\frac{Q^{(h)} {K^{(h)}}^\top}{\sqrt{d_k}}\right), \quad C^{(h)} = A^{(h)} V^{(h)}$$

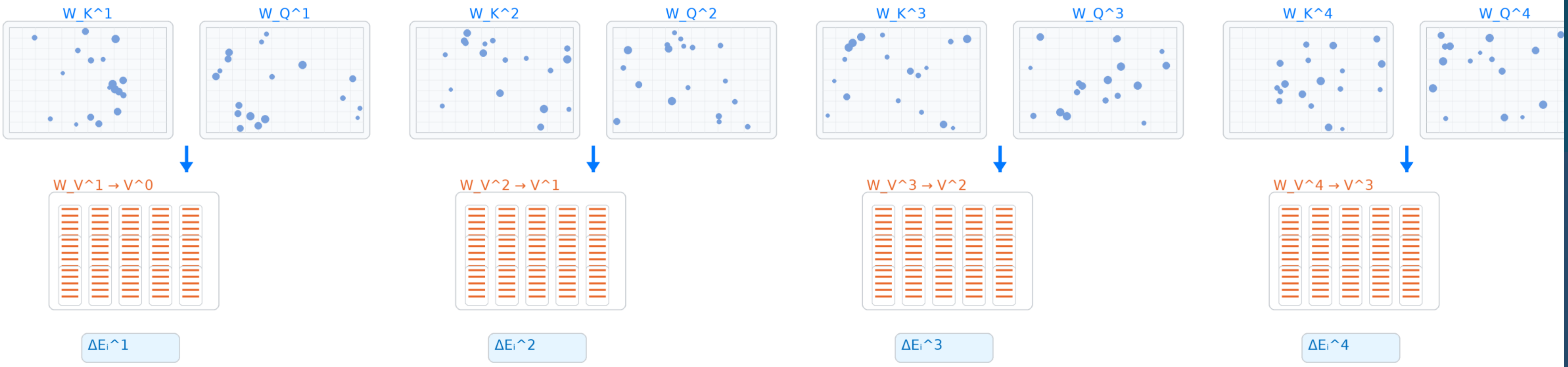
Concat all head outputs and project:

$$\mathrm{MHA}(E) = \left[\, C^{(1)} \parallel \cdots \parallel C^{(H)} \,\right] W_O$$

Residual add & LayerNorm finish the block:

$$E' = \mathrm{LayerNorm}\big(E + \mathrm{MHA}(E)\big)$$

Combine heads: per-head updates add into ΔE_i, then $E_i \leftarrow E_i + \Delta E_i$



Original embedding $E_i + \Delta E_i^{(1)} + \Delta E_i^{(2)} + \Delta E_i^{(3)} + \Delta E_i^{(4)} + \dots$

→ E_i' (concat heads → W_O, then residual add & LayerNorm)

Multi-Head Attention – cont...

Why this works

- Different heads specialize (syntax, agreement, long-distance references, etc.).
- Parallel heads = richer, disentangled signals per token.
- Scales well: GPT-3 uses **96 heads per block** (we illustrate with 4 for clarity).

Scope

- Heads run **for every token** and **at every layer**.
- The mechanism is identical to single-head; only the number of projections changes.

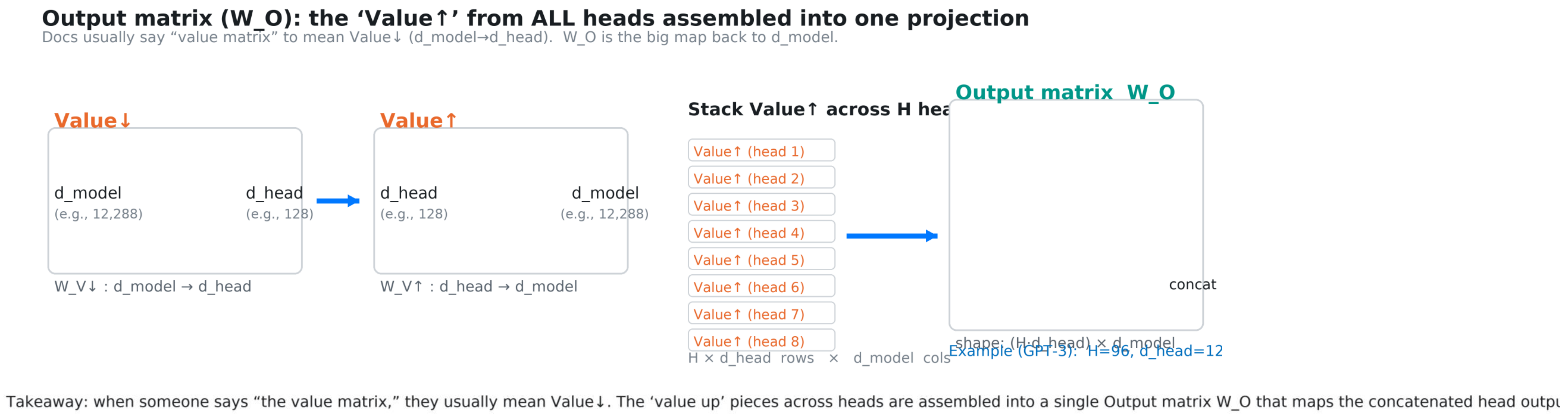
Connect to the "big model" numbers cont ….

- Total weights = 175,181,291,520
- Organized into 27,938 matrices

Matrix name	Matrix dimension
Embedding	D_embed * n_vocab 12,288 * 50,257 = 617558016
Key	12,288 * 128 = 1,572,864 *** 96**
Query	12,288 * 128 = 1,572,864 *** 96**
Value	12,288 * 12,288 (wrong) 128 * 12,288 = 1,572,864 *** 96** And 12,288 * 128 = 1,572,864 *** 96**
Output	
Up-projection	
Down-projection	
UnEmbedding	n_vocab * D_embed 50,257 * 12288 = 617558016

Output Matrix

- Inside attention, each head uses two "value" projections:
  - **Value↓ (W_V↓)**: projects the model width to the head width
    $W_{V\downarrow} \in R$ (e.g., 12,288→128)
  - After attention mixes values, each head must return to model width via **Value↑**.
    Collecting **all heads' Value↑** blocks into one big projection is the **output matrix**
    $W_O \in R^{(H \cdot dhead) \times dmodel}$.



Beyond attention: the full Transformer block

- Data doesn't stop at attention; each block has Multi-Head Attention → residual+LayerNorm → MLP → residual+LayerNorm.
- The MLP is position-wise (same weights at every token), typically widening by 2–4× then projecting back down.
- Depth builds meaning
  - Blocks are repeated many times (e.g., GPT-3 uses 96 layers).
  - As we go deeper, embeddings absorb richer context from other embeddings that are themselves improving—enabling higher-level abstractions (sentiment, tone, genre), not just grammar.

Deep Stack: repeated blocks (e.g., 96 layers in GPT-3)

As depth increases:

- lower: local grammar / short-range links
- middle: long-range dependencies, entities
- higher: topic, sentiment, tone, genre (poem vs. research)

Transformer Block: Multi-Head Attention + MLP (with residuals & LayerNorm)

E (token embeddings, all positions in parallel)

Output of block → feeds next block (same for every token).

Parallelism:

- Across tokens (matrix multiplies over the whole sequence)
- Across heads (dozens of heads per block)

→ GPUs exploit this for scale and throughput

Connect to the "big model" numbers cont ....

- Total weights = 175,181,291,520
- Organized into 27,938 matrices

Matrix name	Matrix dimension
Embedding	D_embed * n_vocab 12,288 * 50,257 = 617558016
Key	12,288 * 128 = 1,572,864 * 96 *** 96 layers**
Query	12,288 * 128 = 1,572,864 * 96 *** 96 layers**
Value	12,288 * 12,288 (wrong) 128 * 12,288 = 1,572,864 * 96 *** 96 layers** And 12,288 * 128 = 1,572,864 * 96 *** 96 layers**
Output	
Up-projection	
Down-projection	
UnEmbedding	n_vocab * D_embed 50,257 * 12288 = 617558016

Total studied params so far: ~58B parameters

Yet to study params: ~ 117B parameters

What's next? - MLP layers ...

Where the parameters live (attention vs other)

Attention params (K/Q/V + Output)
≈ 57,982,058,496 (≈58B) ≈ one-third

Other params (MLPs, embeddings, layer norms, etc.)
≈ two-thirds of total

Takeaways

- Attention is powerful for routing information, and it parallelizes well across tokens & heads.
- Most parameters come from the MLPs (big up/down projections at each layer).
- Scaling works because GPUs handle the large matrix multiplies in parallel.