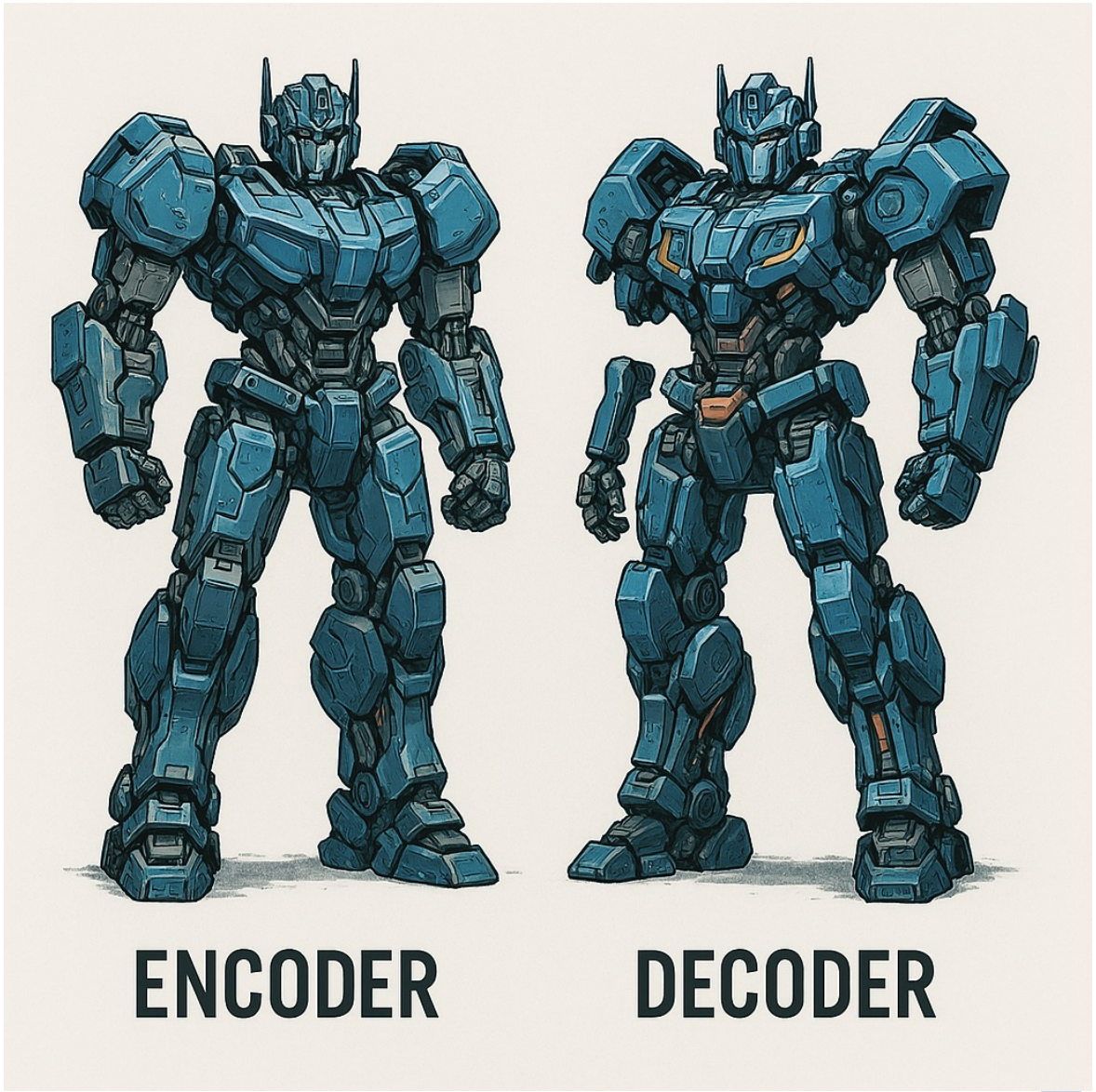
EE-508: Hardware Foundations for Machine Learning

Transformers – Part 2

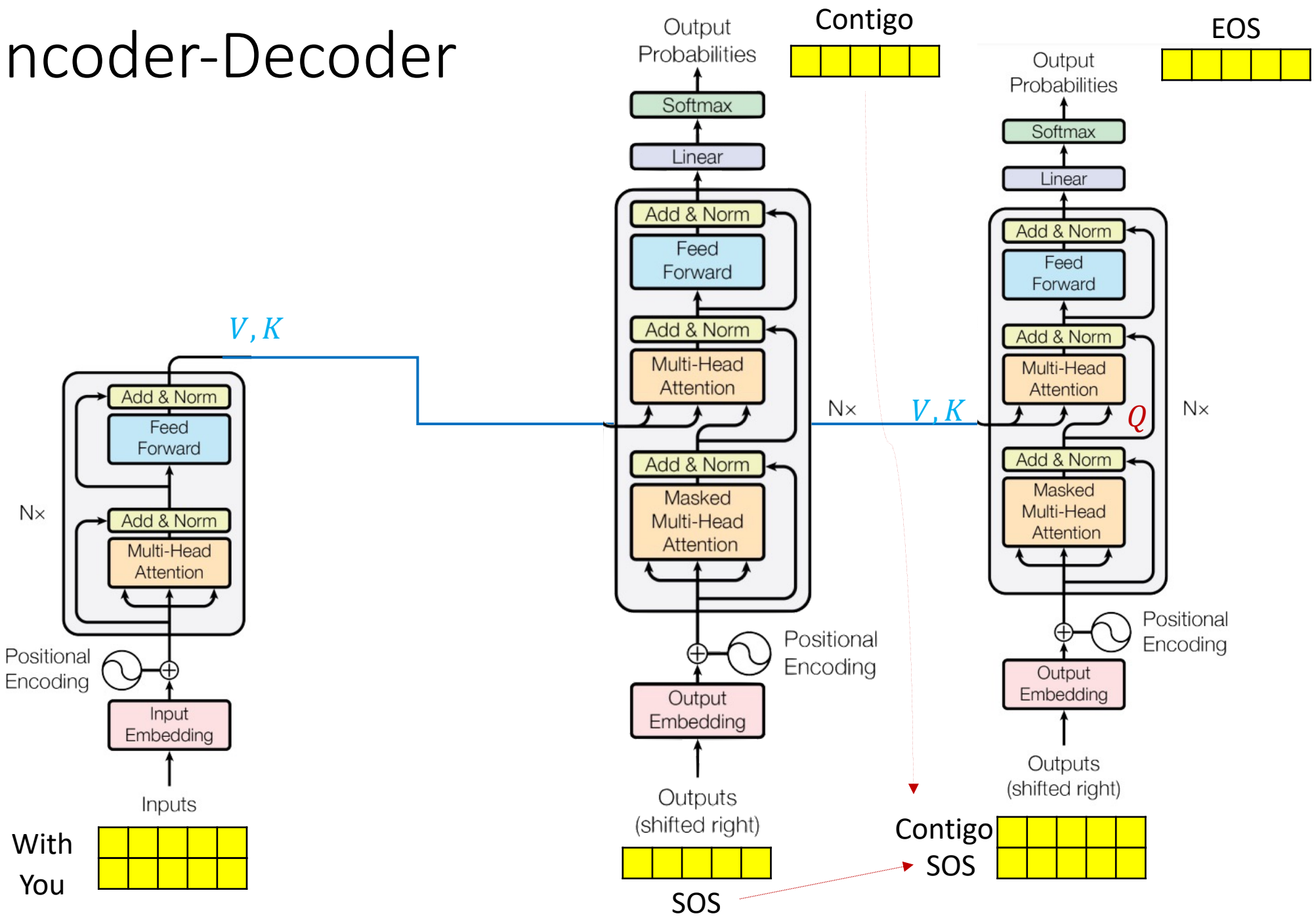
University of Southern California

Ming Hsieh Department of Electrical and Computer Engineering

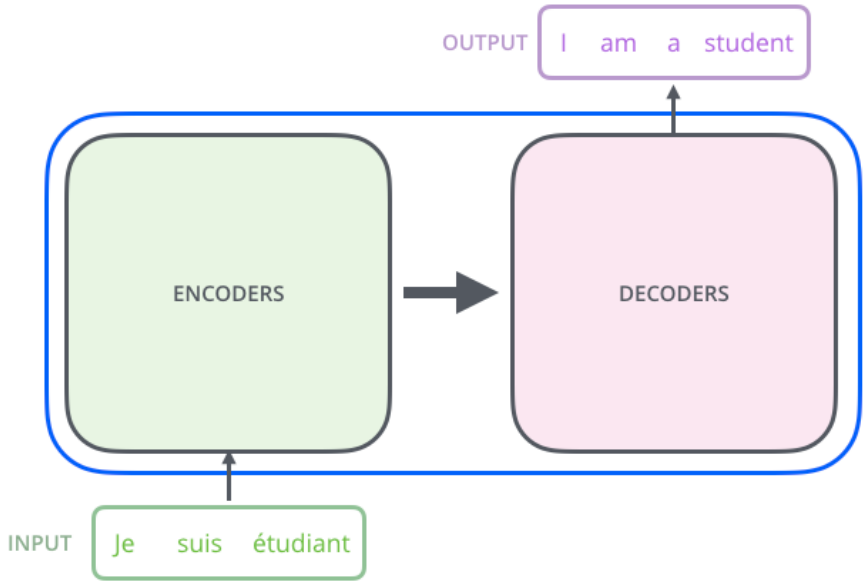
Instructors:
Arash Saifhashemi


ENCODER
DECODER

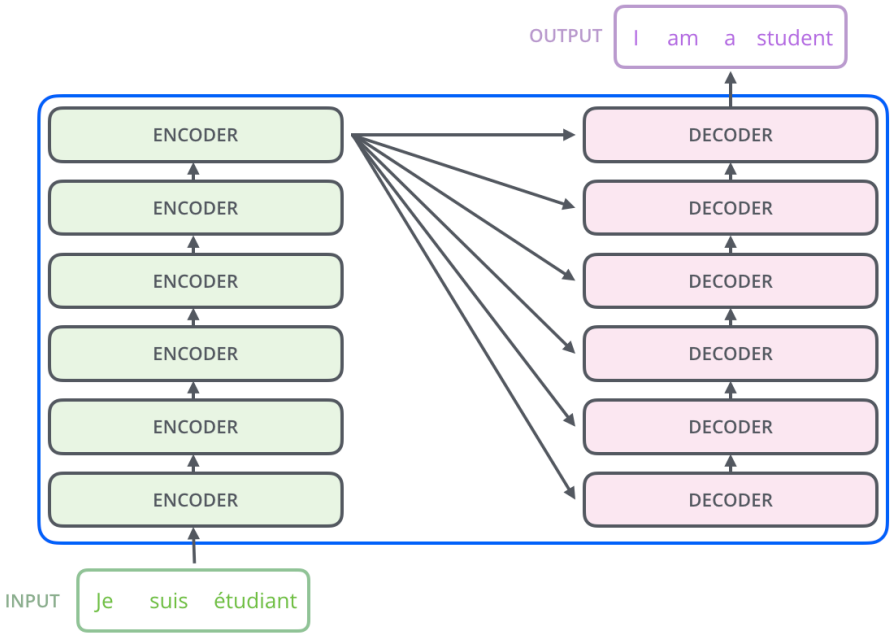
Encoder-Decoder



Encoder-Decoder Summary



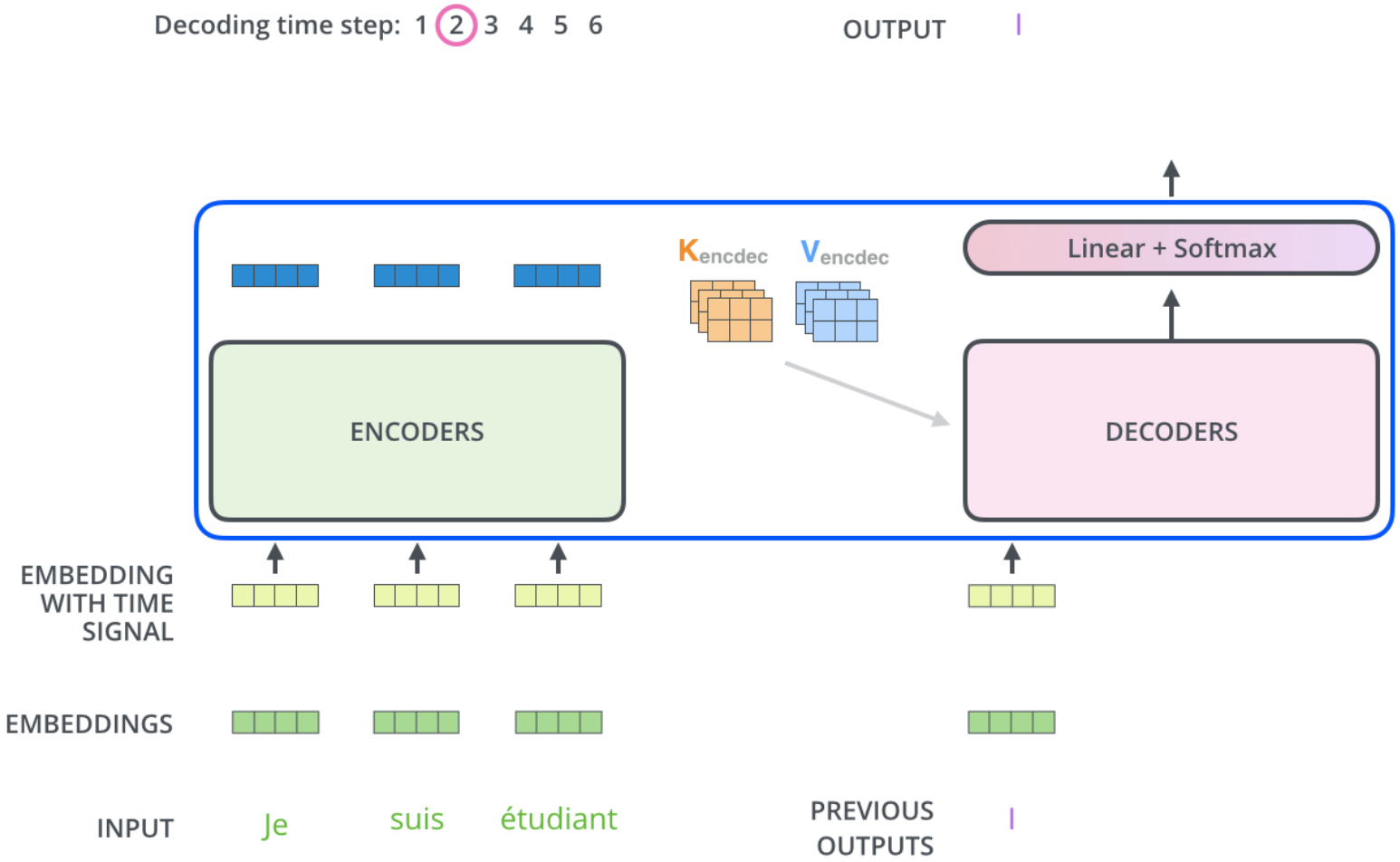
Encoder-decoder structure for translation



Stacked encoders and decoders and attention connections

Source: https://jalammar.github.io/illustrated-transformer/

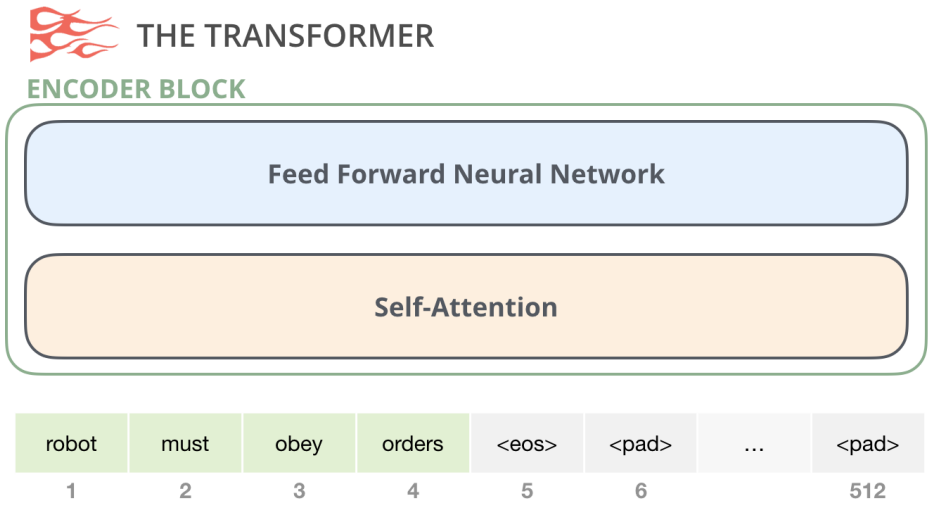
Encoder-Decoder Summary



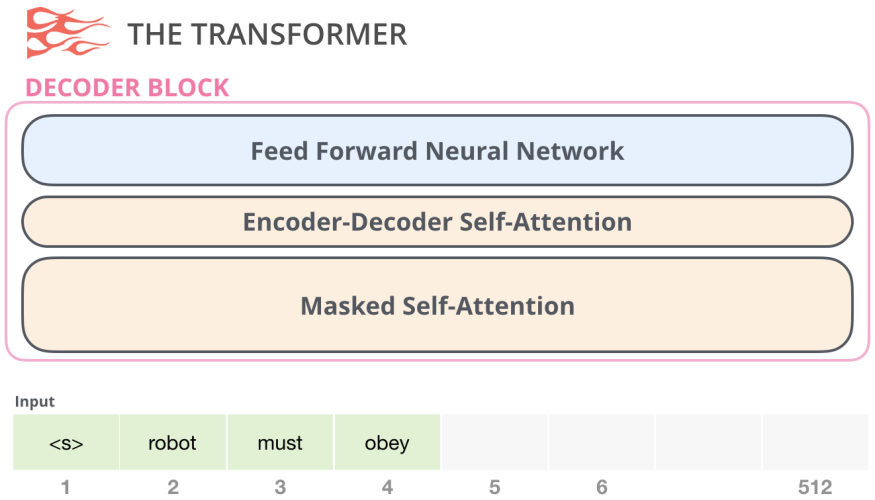
Source: https://jalammar.github.io/illustrated-transformer/

The Evolution of the Transformer Block

"...robot must obey orders given by human beings, except where such orders would conflict with the First Law"



An encoder block from the original transformer paper can take inputs up until a certain max sequence length (e.g. 512 tokens). It's okay if an input sequence is shorter than this limit, we can just pad the rest of the sequence.

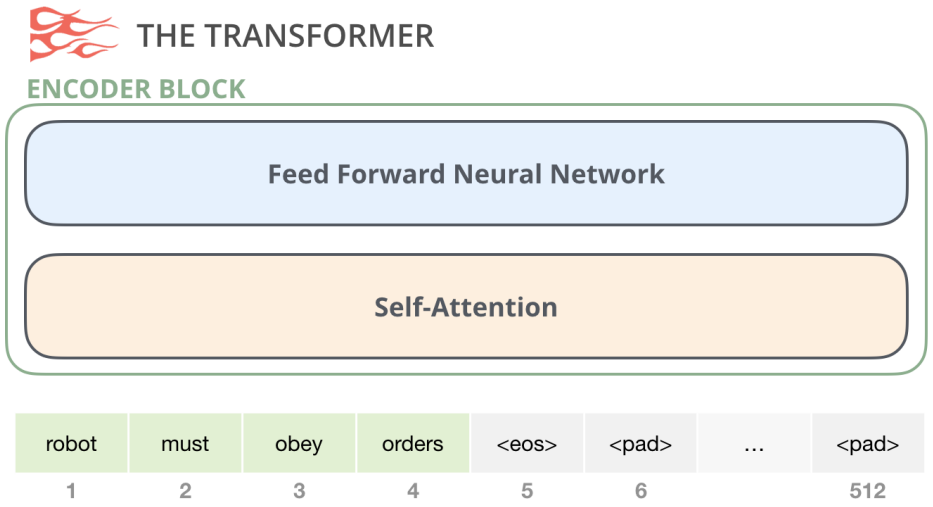


In the decoder, future tokens are masked. During training If it sees future tokens, it "cheats".

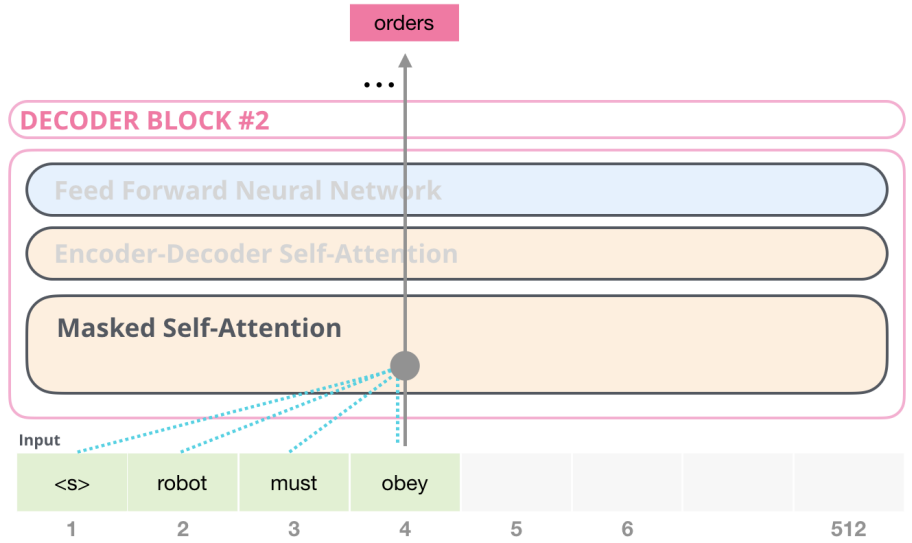
Source: https://jalammar.github.io/illustrated-transformer/

The Evolution of the Transformer Block

"...robot must obey orders given by human beings, except where such orders would conflict with the First Law"



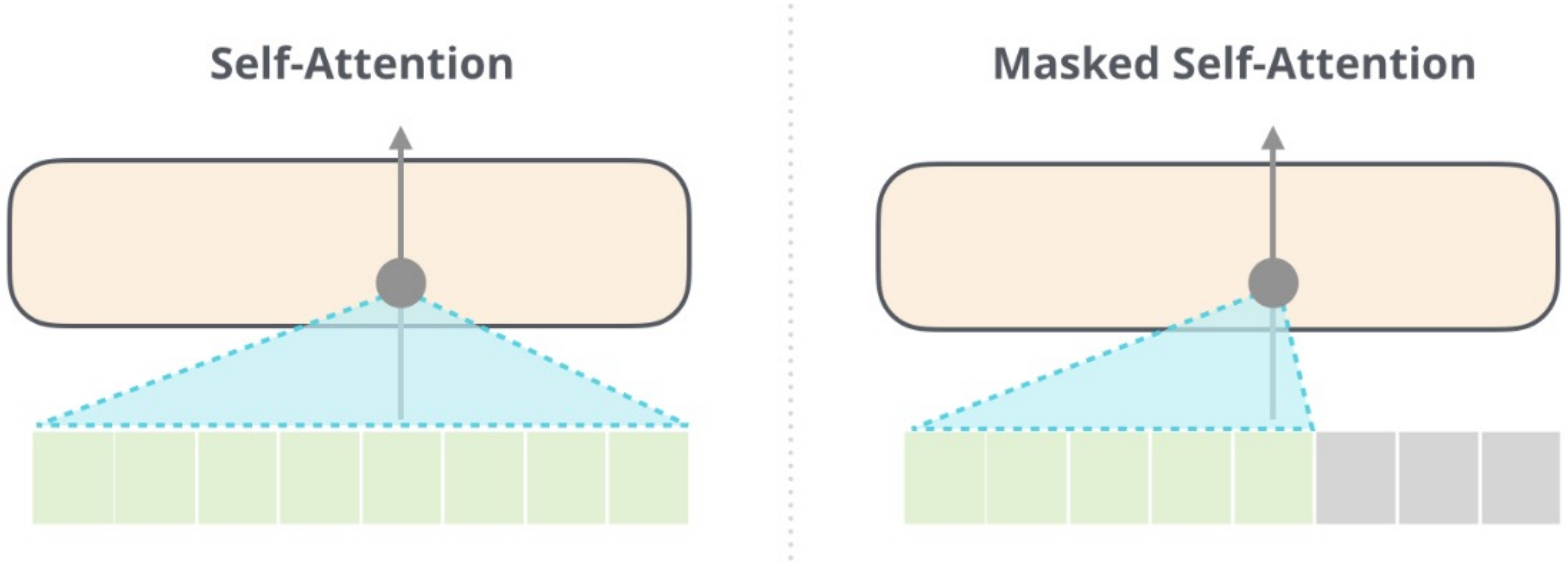
An encoder block from the original transformer paper can take inputs up until a certain max sequence length (e.g. 512 tokens). It's okay if an input sequence is shorter than this limit, we can just pad the rest of the sequence.



In the decoder, future tokens are masked. During training If it sees future tokens, it "cheats".

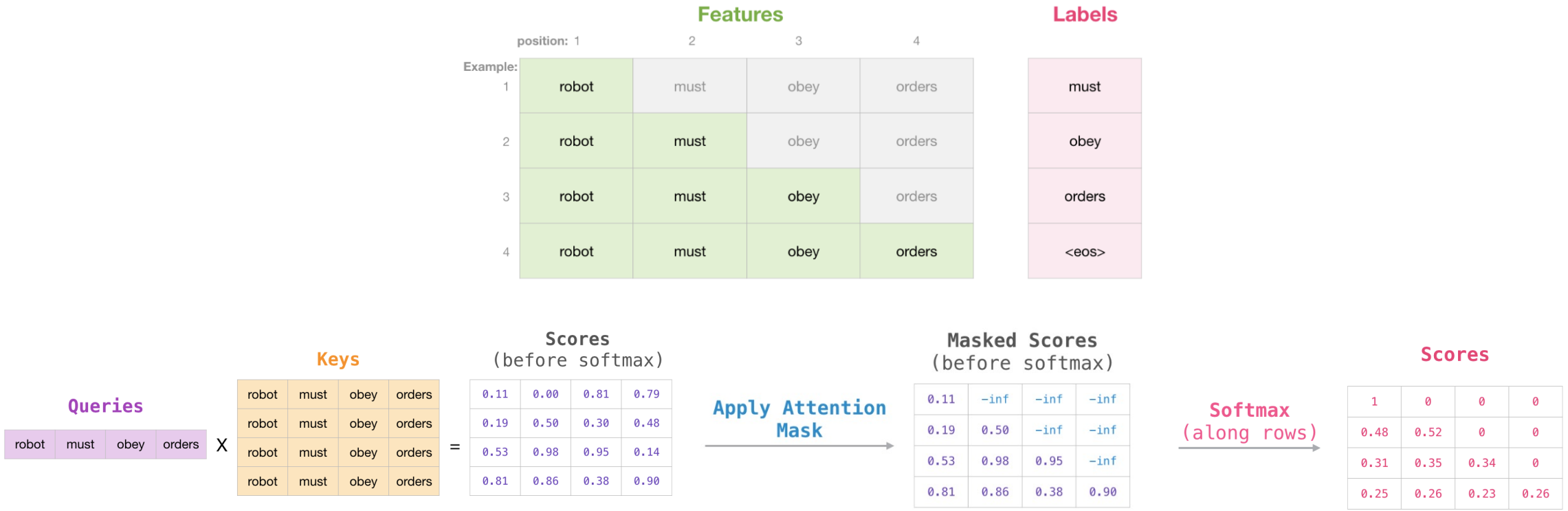
Source: https://jalammar.github.io/illustrated-transformer/

The Evolution of the Transformer Block



Source: https://jalammar.github.io/illustrated-gpt2/

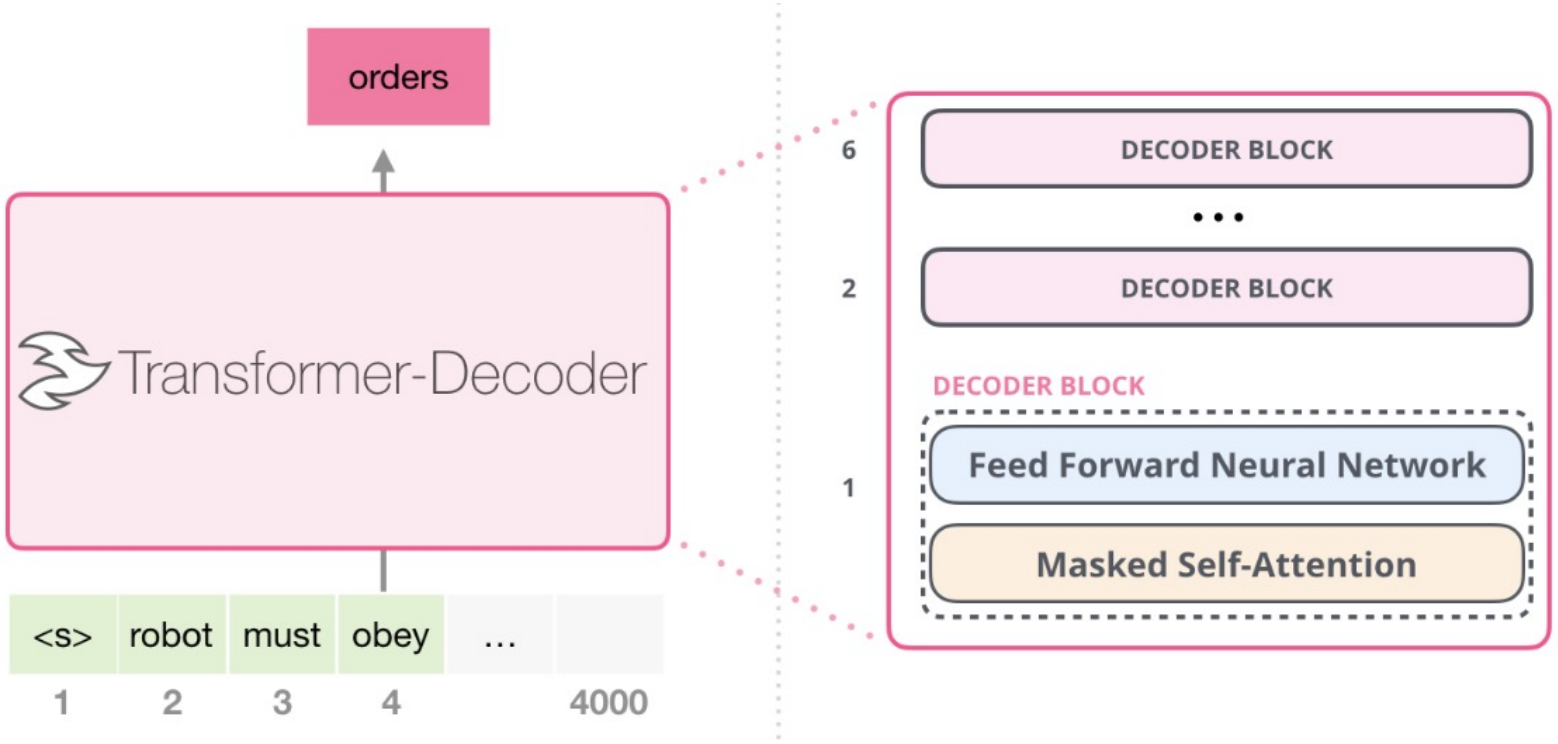
The Evolution of the Transformer Block



"...robot must obey orders given by human beings, except where such orders would conflict with the First Law"

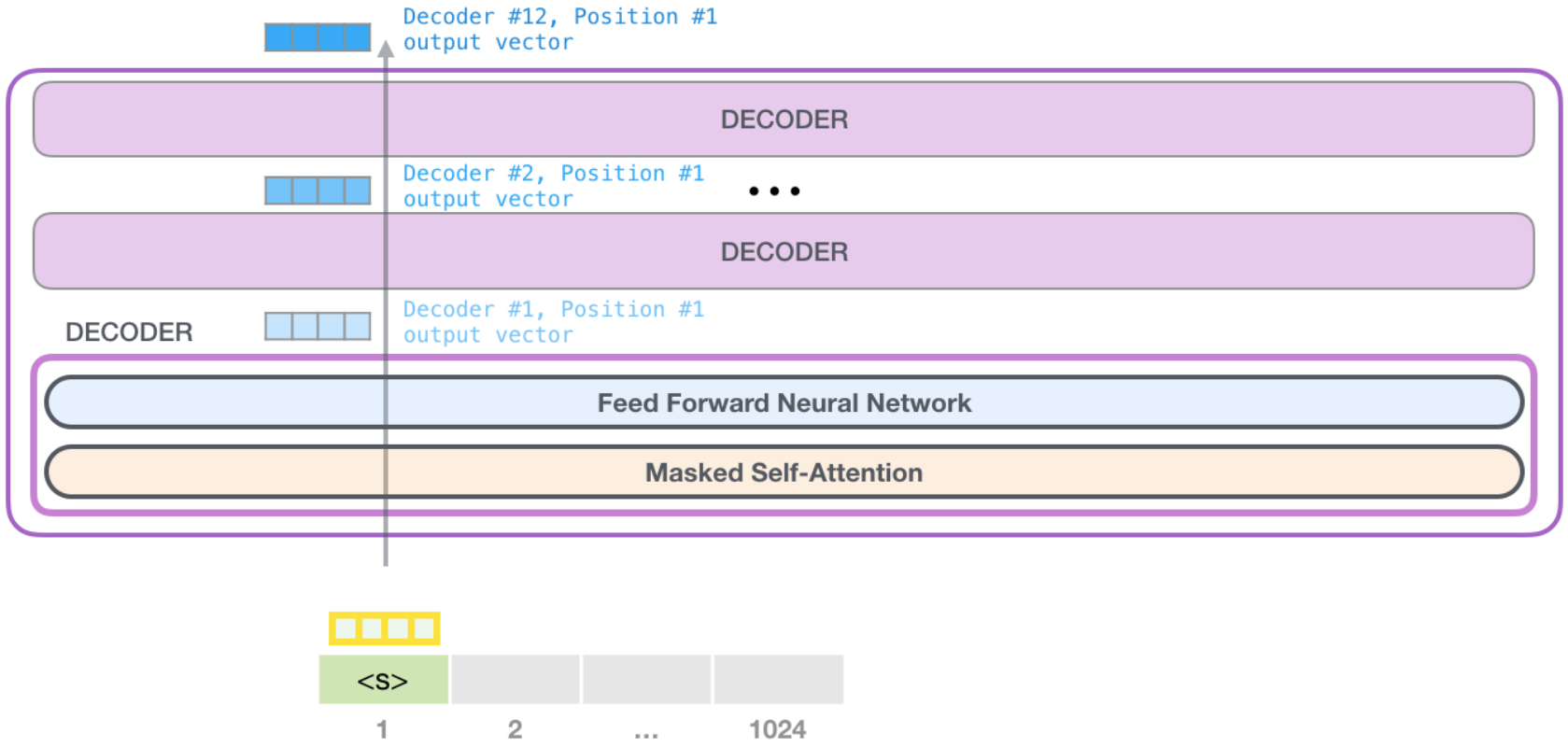
Source: https://jalammar.github.io/illustrated-gpt2/

The Decoder Only Block



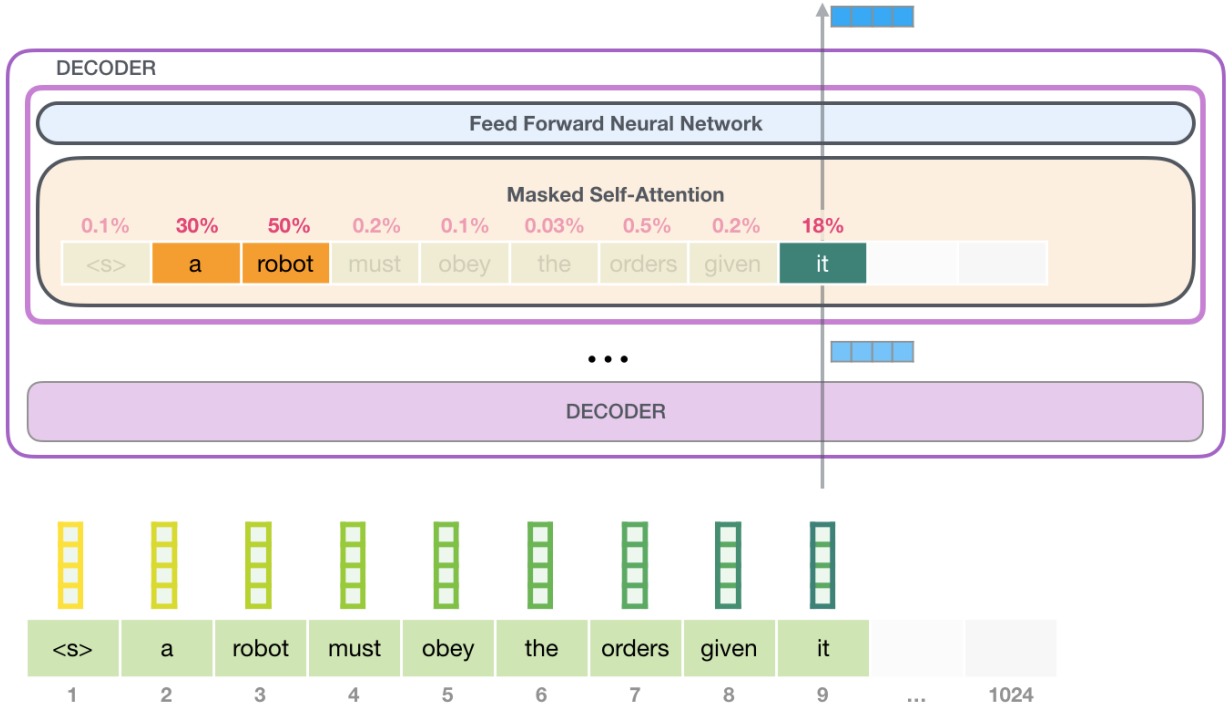
Source: https://jalammar.github.io/illustrated-gpt2/

The Output is Still Embedding



Source: https://jalammar.github.io/illustrated-gpt2/

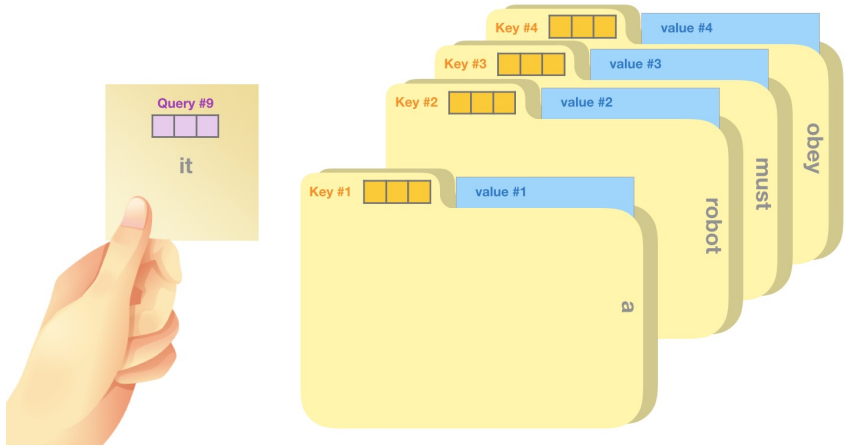
Masked Self Attention

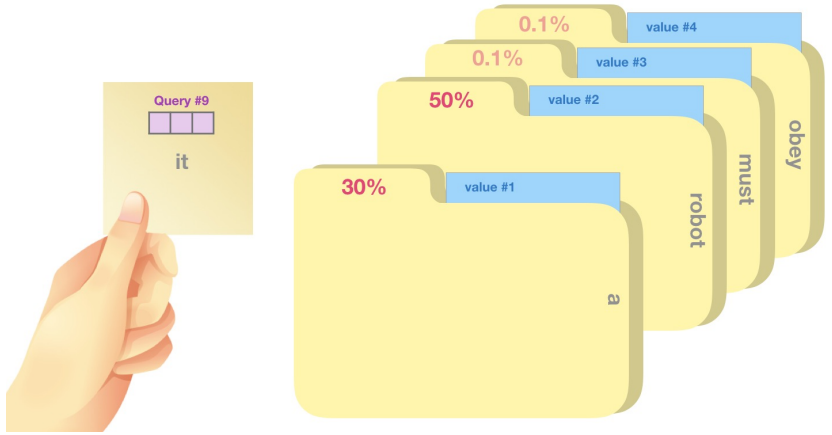


The self-attention layer in the top block is paying attention to “a robot” when it processes the word “it”. The vector it will pass to its neural network is a sum of the vectors for each of the three words multiplied by their scores.

Source: https://jalammar.github.io/illustrated-gpt2/

Remember K,V,Q from Database Analogy





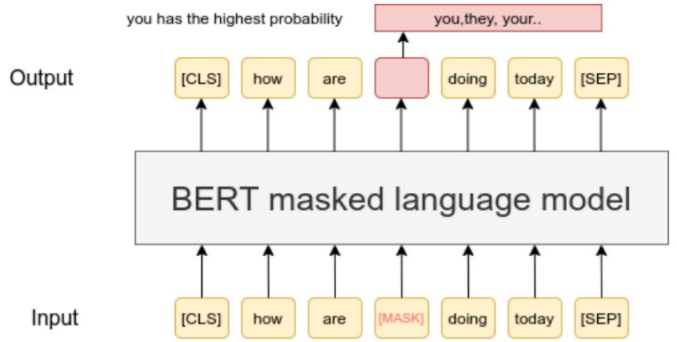
Second Law of Robotics

A robot must obey the orders given **it** by human beings except where **such orders** would conflict with the **First Law**.

Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
it		0.19	
		Sum:	

Understanding vs Generation

- **BERT = encoder-only (understanding)**
  - **Input:**
    "A robot may not [MASK] a human being."
  - **Output (prediction):**
    "*injure*"
  - BERT is trained to fill in masked words based on full sentence context. It sees both left and right of [MASK].
- **GPT = decoder-only (generation)**
  - **Input:**
    "A robot may not"
  - **Generated Output (token-by-token):**
    "injure" → "a" → "human" → "being" ...
  - GPT-2 generates the next token using only the previous ones.



Applications of BERT

Text Classification

Sentiment analysis, topic classification

The movie was [great]

↓

Positive

Question Answering (QA)

Input: passage + question

Einstein was born in Germany.

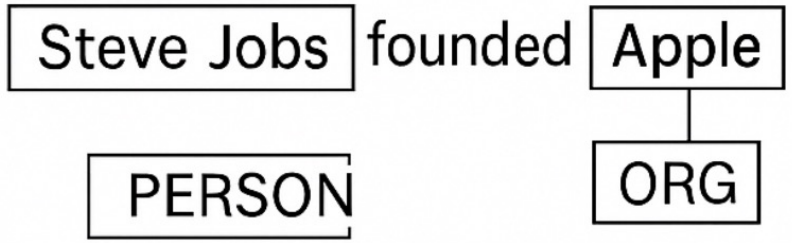
Q: Where was Einstein born?

↓

Germany

Named Entity Recognition

Label words as people, places, organizations



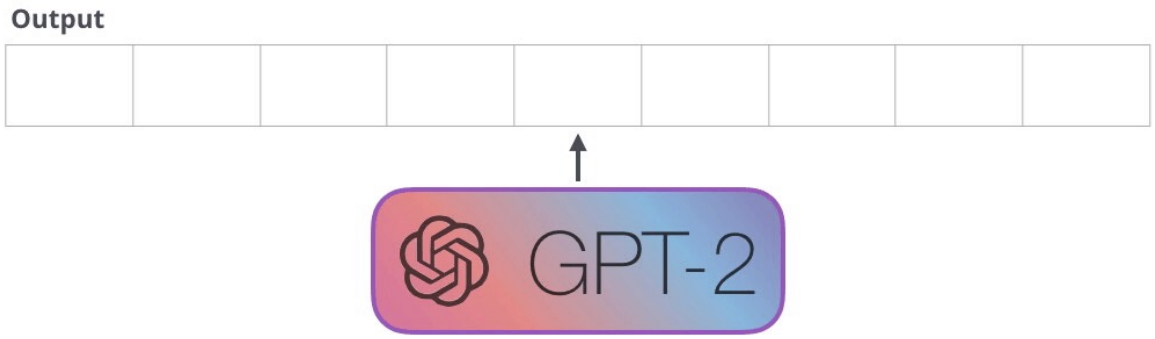
Coreference Resolution

Figuring out what pronouns refer to

John helped Tom because he was late.

he = John?

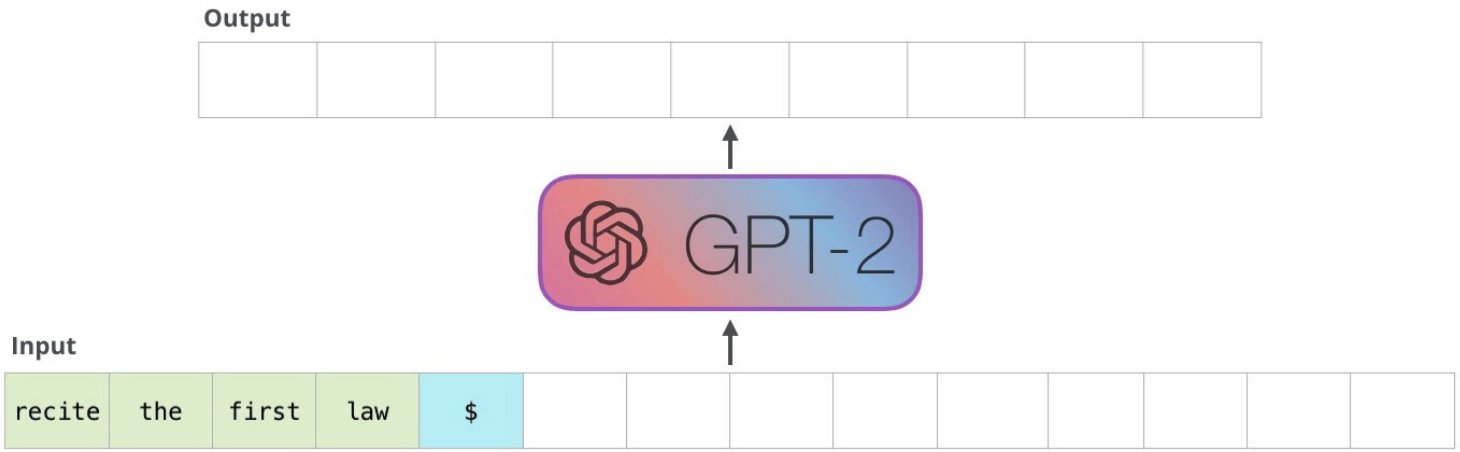
How Does the Decoder Work?



In the auto-regressive generation of the decoder, given an input the model predicts the next token, and then taking the combined input in the next step the next prediction is made.

(Image source: https://jalammar.github.io/illustrated-gpt2/)

How Does the Decoder Work?

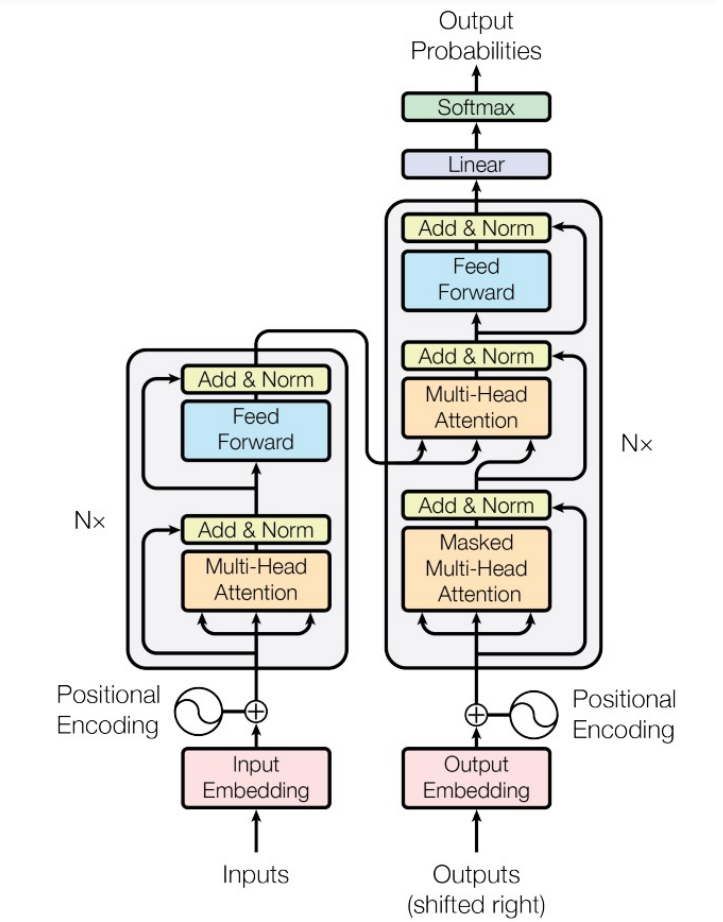


In the auto-regressive generation of the decoder, given an input the model predicts the next token, and then taking the combined input in the next step the next prediction is made.

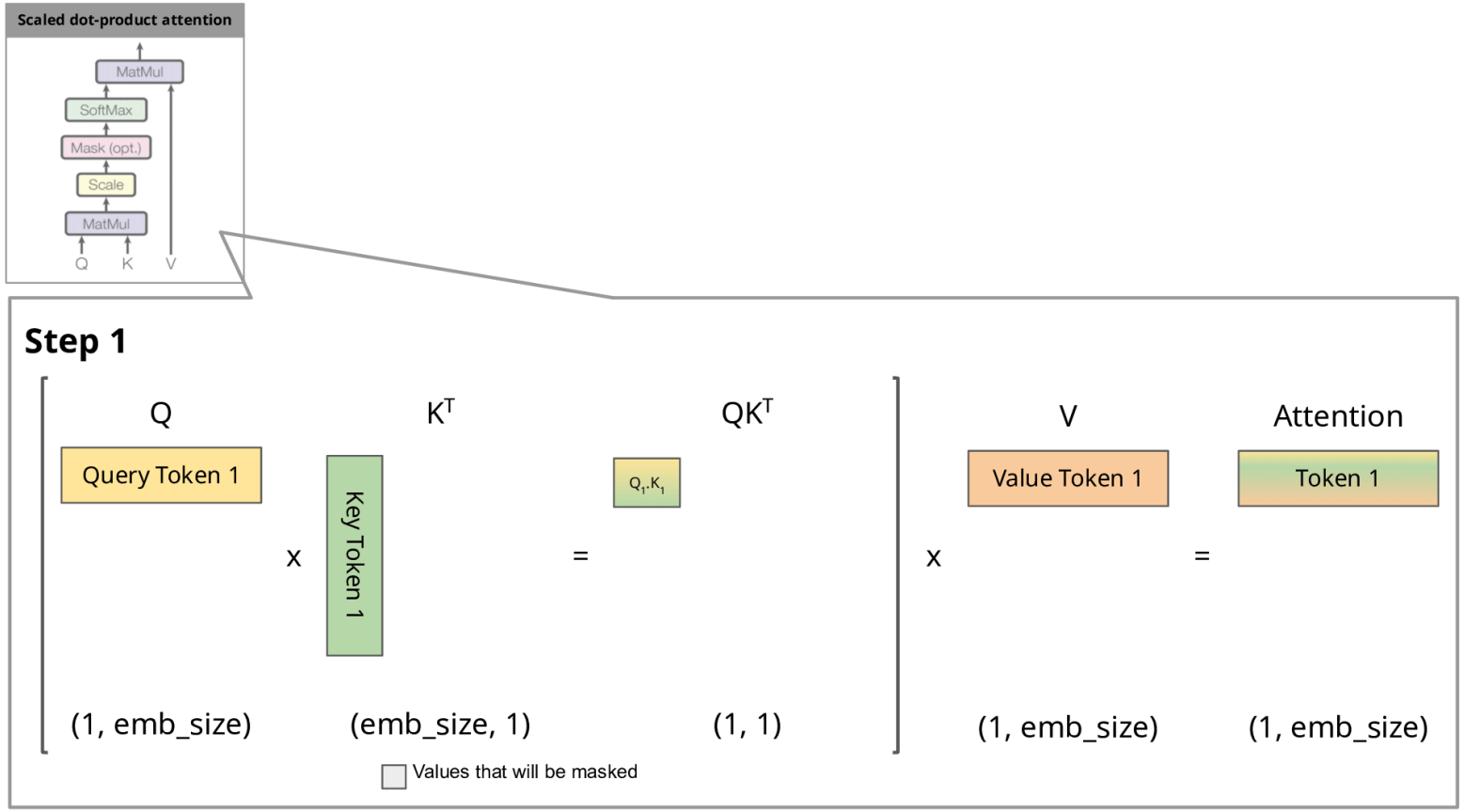
(Image source: https://jalammar.github.io/illustrated-gpt2/)

Understanding KV Caching in Transformers

- **Transformer Inference Basics**
  - **Autoregressive Generation**
    - Transformers generate text step-by-step (autoregressively).
    - To predict the *next* token, the model uses *all* previous tokens as input.
  - **Example:**
    1. Input: *[CLS] The cat sat* -> Predict: **on**
    2. Input: *[CLS] The cat sat **on*** -> Predict: **the**
    3. Input: *[CLS] The cat sat on **the*** -> Predict: **mat**
  - Each prediction step involves a full pass through the Transformer layers.



How Does the Decoder Work?

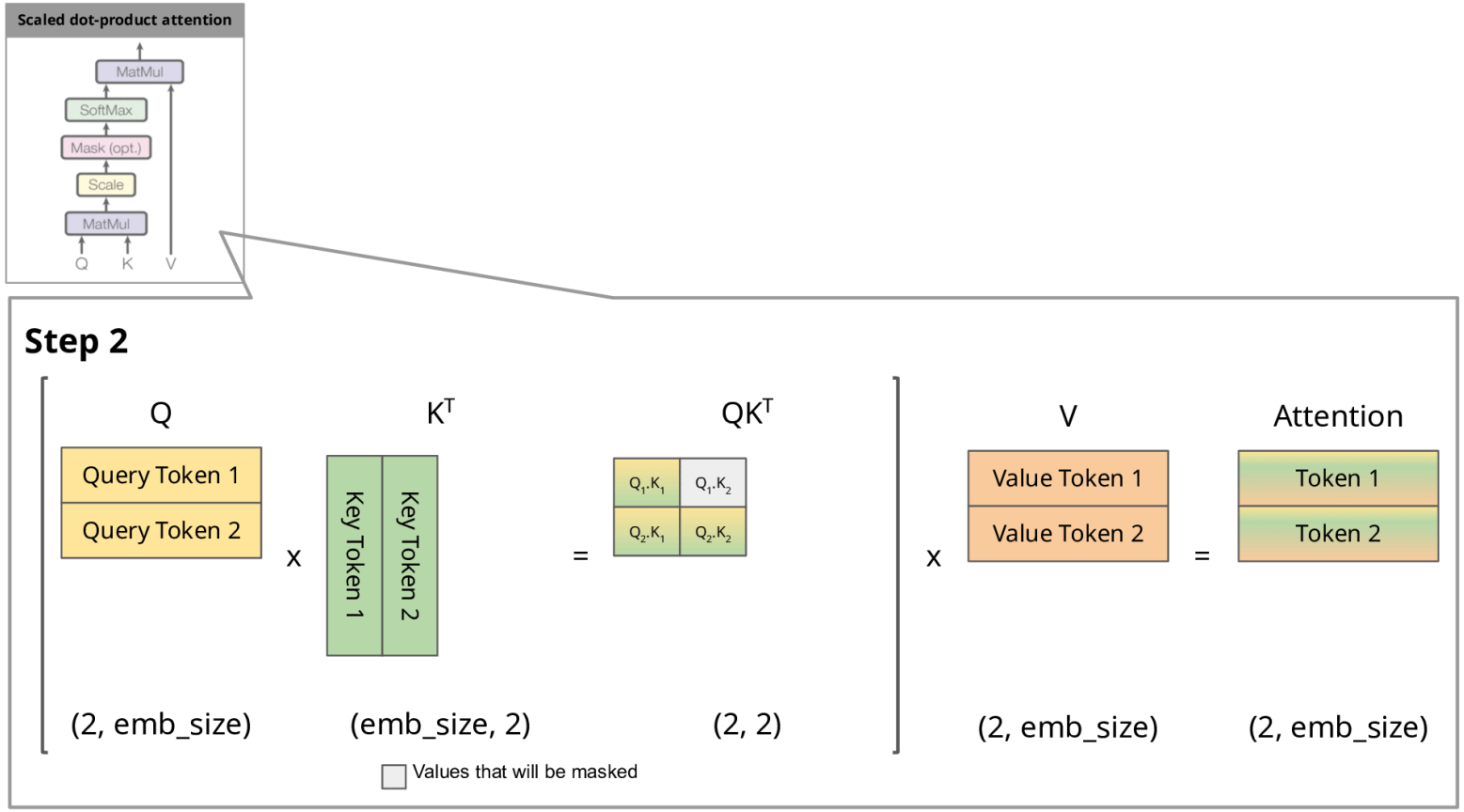


Zoom-in! (simplified without Scale and Softmax)

Step-by-step visualization of the scaled dot-product attention in the decoder. emb_size means embedding size.

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

How Does the Decoder Work?

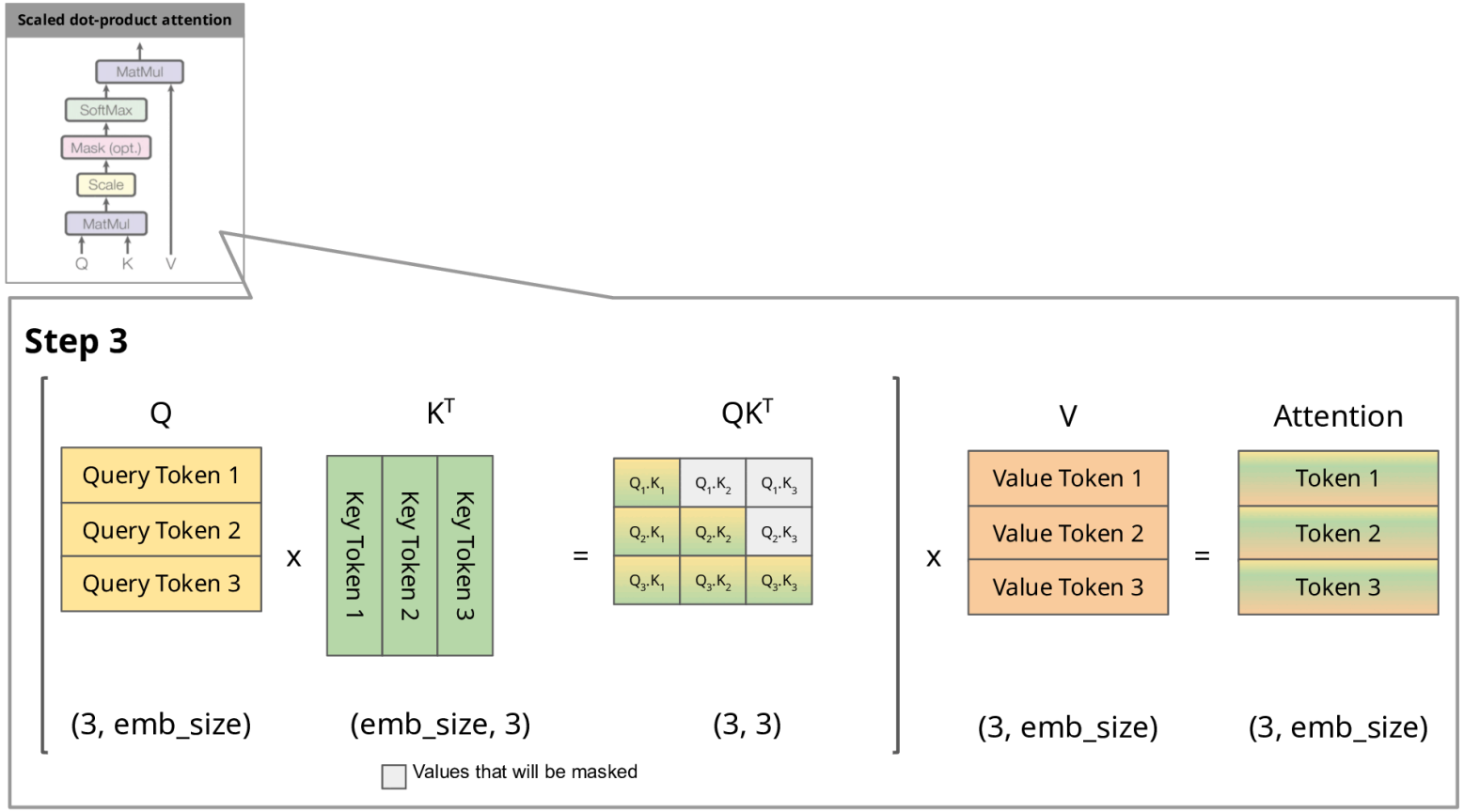


Zoom-in! (simplified without Scale and Softmax)

Step-by-step visualization of the scaled dot-product attention in the decoder. emb_size means embedding size.

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

How Does the Decoder Work?

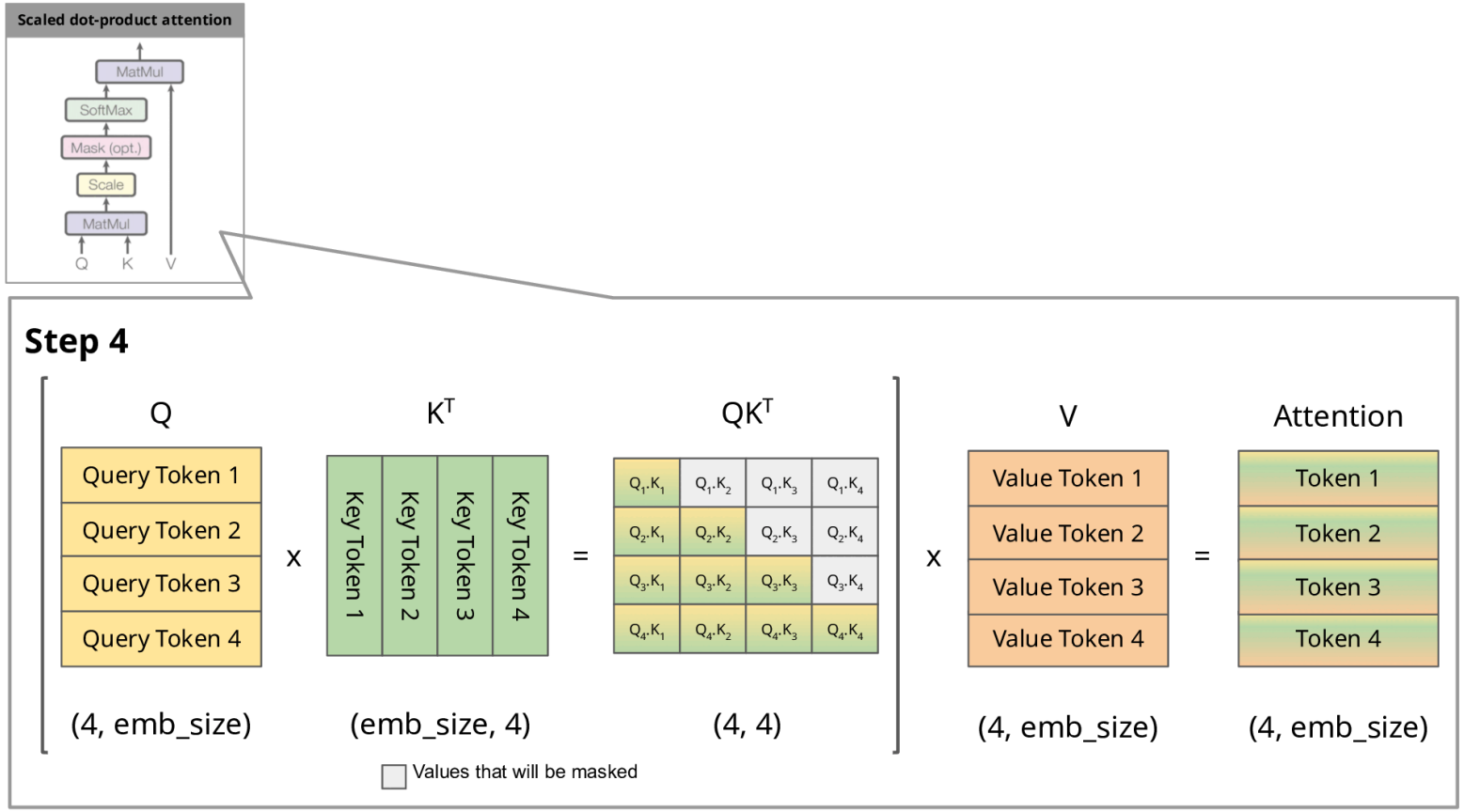


Zoom-in! (simplified without Scale and Softmax)

Step-by-step visualization of the scaled dot-product attention in the decoder. emb_size means embedding size.

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

How Does the Decoder Work?



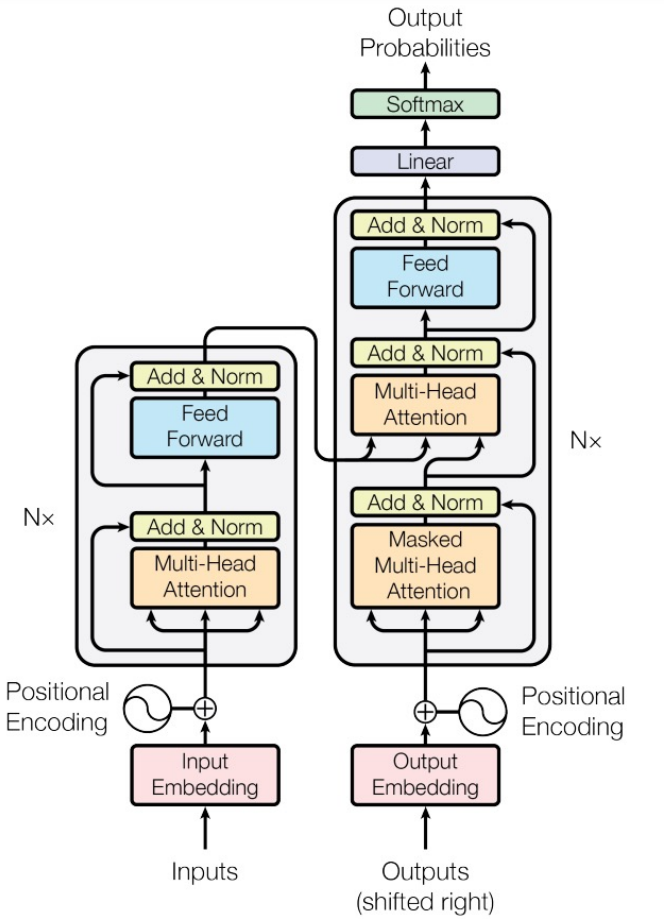
Zoom-in! (simplified without Scale and Softmax)

Step-by-step visualization of the scaled dot-product attention in the decoder. emb_size means embedding size.

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

The Challenge: No Cache

- **What happens without caching?**
  - Every calculation is repeated for the *entire growing sequence* at each step.
  - Embeddings, Positional Encodings, Q/K/V projections, Attention Scores, FFNs... everything!
- **Example (Predicting Token 4: "mat"):**
  - Input: `The cat sat on`
  - Calculations involve interactions of all token pairs
- **Example (Predicting Token 5):**
  - Input: `The cat sat on` **`the`**
  - *All* previous interactions are recalculated, PLUS new ones:
  - **Highly inefficient!**



The cat sat on

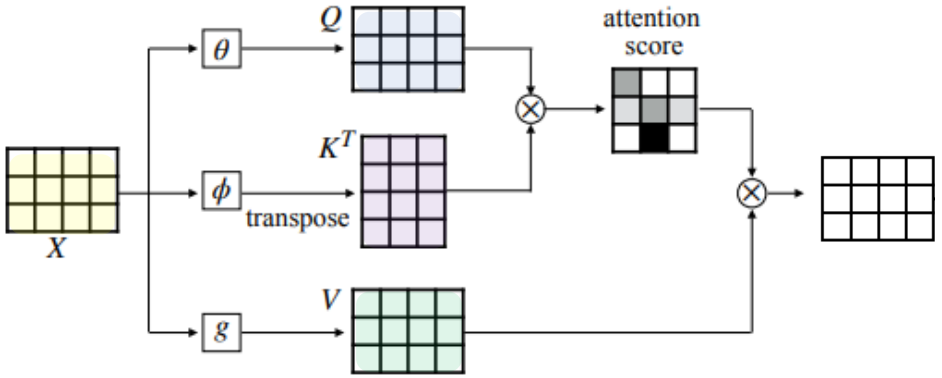
The cat sat on the

Solution: KV Cache

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

- **What Can We Reuse?**
  - In the attention mechanism, the **Key (K)** and **Value (V)** vectors for a token depend *only* on that token's input representation (at that layer).
  - They don't change based on which token is *querying* them.
- **The KV Cache:**
  - Store (cache) the calculated **K** and **V** vectors for all previous tokens in each layer.

The cat, the cute little white one, sat on the mat and it



Role of Cache

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

- **Optimized Calculation Flow**
  1. Calculate Q, K, V only for the *newest* input token.
  2. Retrieve the cached K and V vectors for *all previous* tokens.
  3. Combine the new K/V with the cached K/V.
  4. Calculate attention scores *only* for the new token's Q against *all* K's (cached + new).
  5. Compute the weighted sum using *all* V's (cached + new).
  6. Update the cache by adding the new K and V.

Role of Cache

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

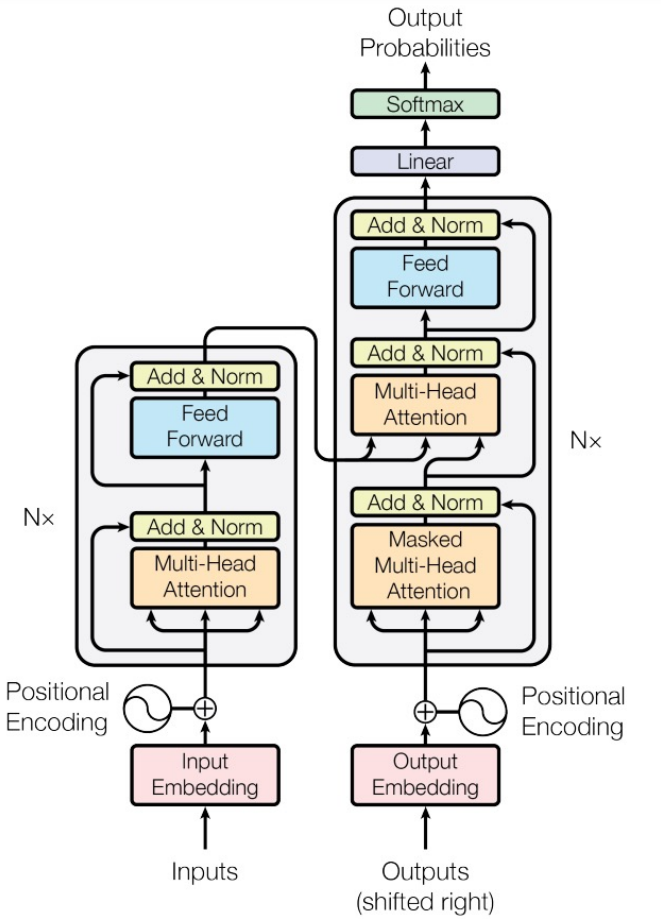
- **Example (Predicting Token 5 with Cache):**
  - `The cat sat on` **`the`**
  - Input: Only process token 4 (the). Calculate: $Q_4 = W_Q \cdot x_4, \quad K_4 = W_K \cdot x_4, \quad V_4 = W_V \cdot x_4$
  - Retrieve Cache $[K_0, K_1, K_2, K_3]$, $[V_0, V_1, V_2, V_3]$:
  - Attention: Calculate Q_4 interacting with $[K_0, K_1, K_2, K_3]$
  - Output: Weighted sum using $[V_0, V_1, V_2, V_3]$:
  - Update Cache: Store K_4, V_4.

$$\text{Scores: } \alpha_i = \text{softmax}\left(\frac{Q_4 \cdot K_i^T}{\sqrt{d_k}}\right), \quad i \in \{1, 2, 3, 4\}$$

- **Much faster:** Only 1 Q vector calculation and 1 x N attention instead of N x N.

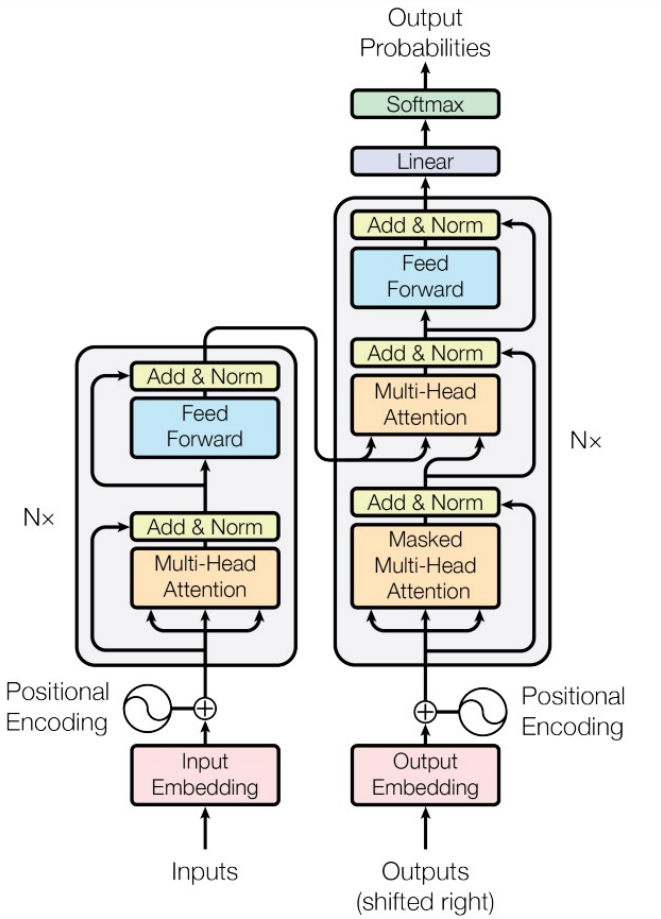
How KV Cache Works

- **Encoding Position Information**
  - Transformers need to know the order of tokens.
- **Traditional Absolute Encodings:** (e.g., Sinusoidal, Learned Embeddings)
  - Add a unique vector based on the token's *absolute position* (1st, 2nd, 3rd...) directly to the token embedding.
  - $Input = TokenEmbedding + PositionalEncoding(absolute\_position)$

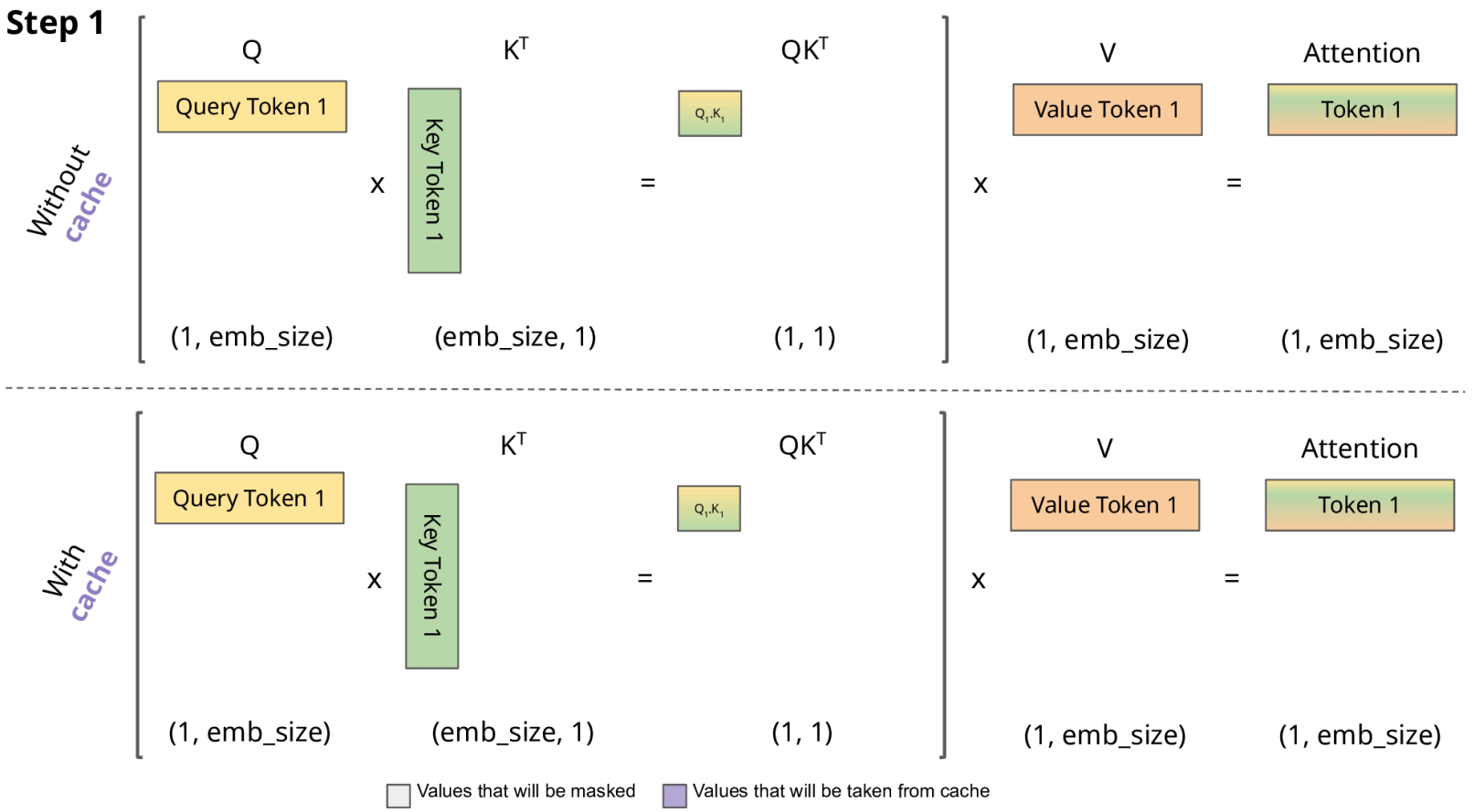


Caching with Absolute Encodings

- **How it Works:**
  - The K and V vectors are calculated from embeddings that *already include* absolute positional information.
  - KV Caching **still works** perfectly fine. You cache these position-infused K and V vectors.
- **Subtlety:**
  - The cached K and V vectors have the *absolute position* information "baked in".
  - Attention scores implicitly depend on these absolute positions.
  - The model must learn to interpret **relative positioning** (m-n) from vectors that have **absolute positions** (m and n) baked into them.



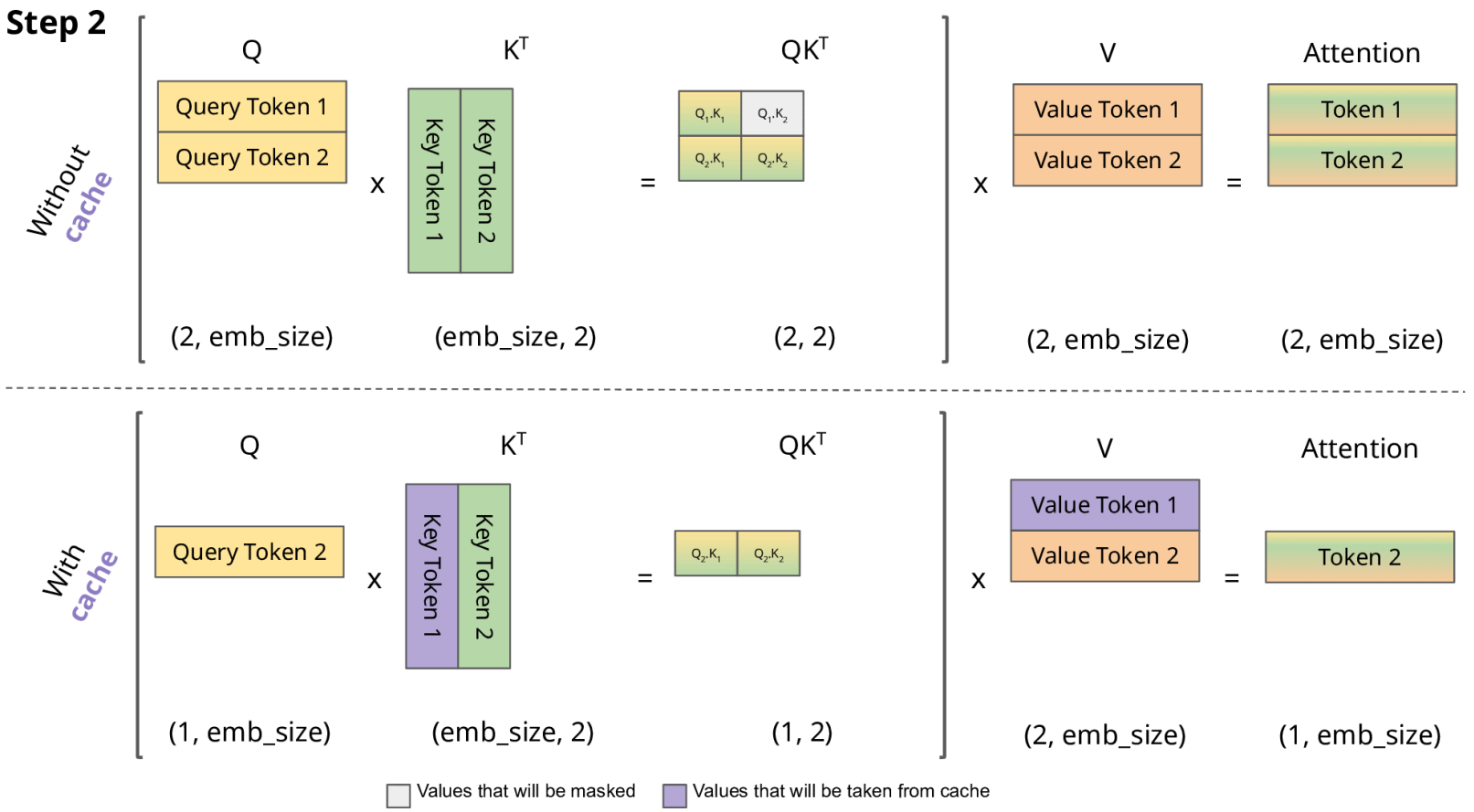
Caching Key and Value



- Pro: Matrix multiplication would be simpler (smaller size)
- Con: We need extra memory per token!

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

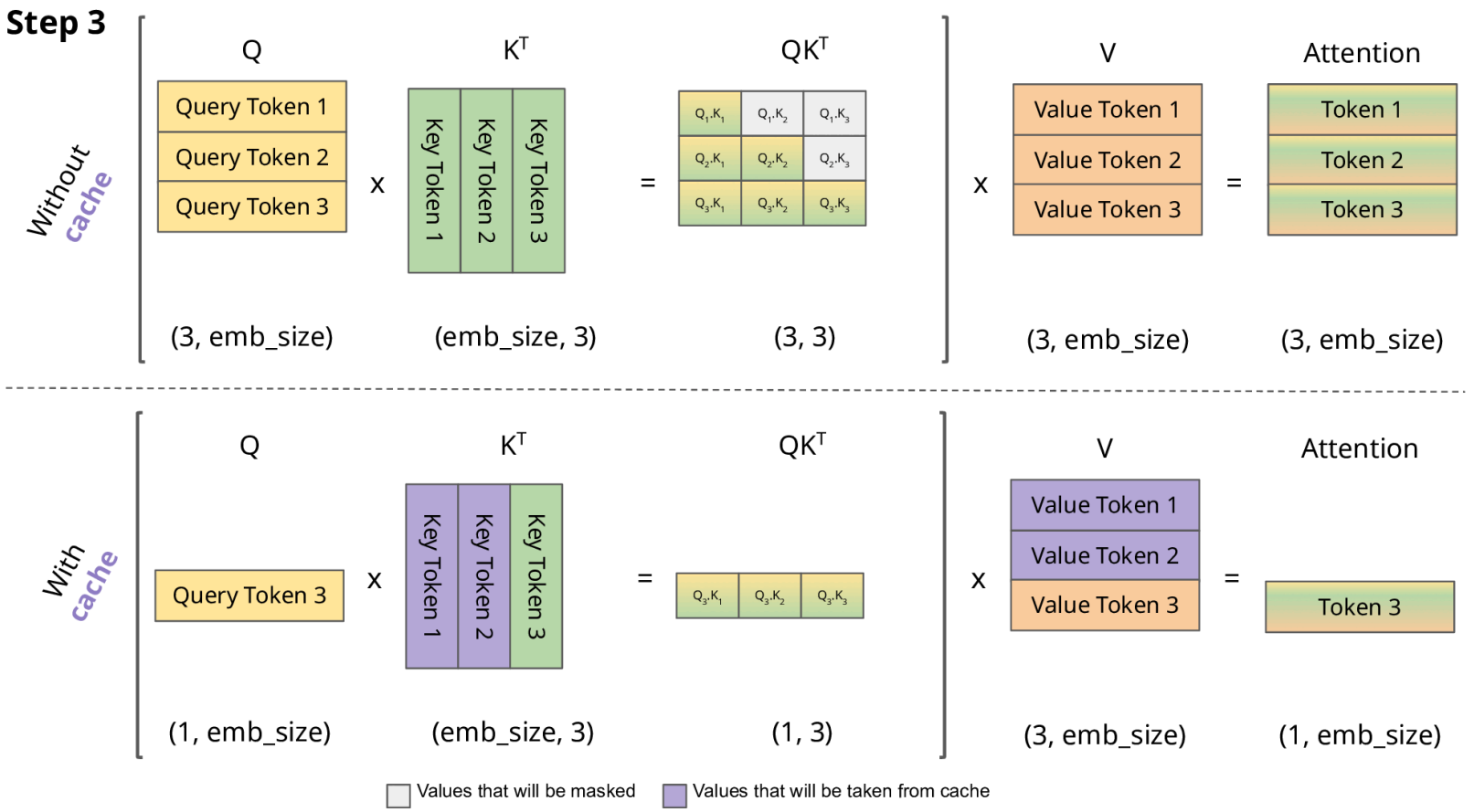
Caching Key and Value



- Pro: Matrix multiplication would be simpler (smaller size)
- Con: We need extra memory per token!

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

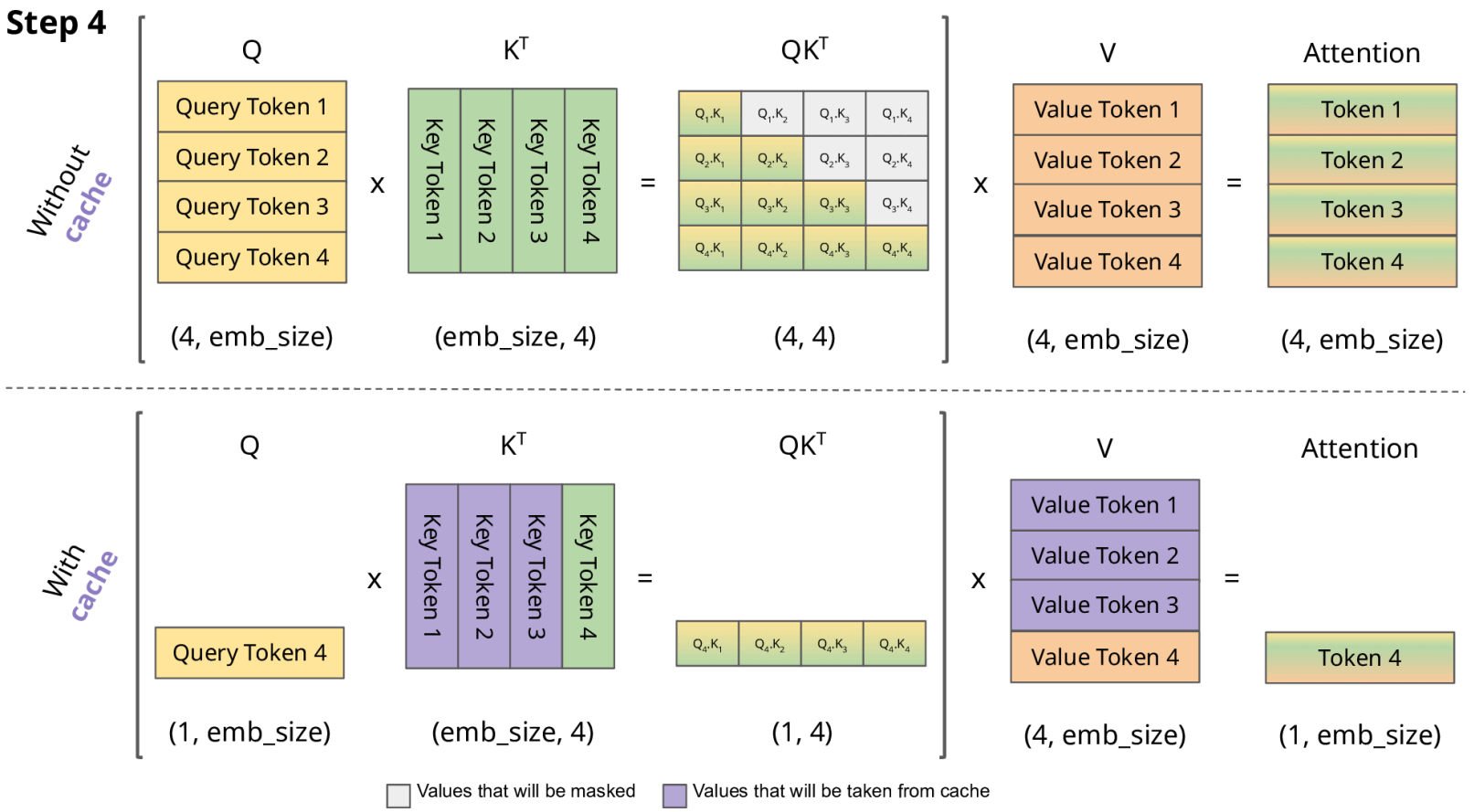
Caching Key and Value



- Pro: Matrix multiplication would be simpler (smaller size)
- Con: We need extra memory per token!

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

Caching Key and Value

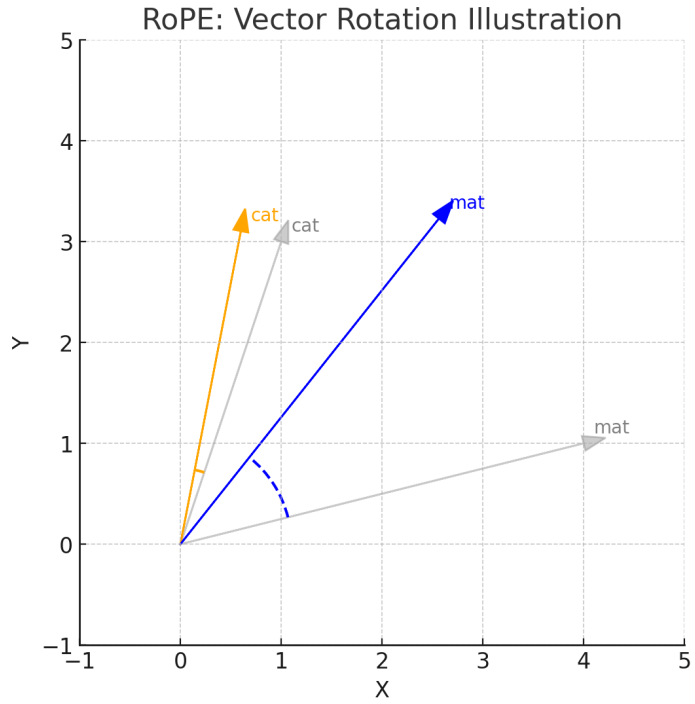


- Pro: Matrix multiplication would be simpler (smaller size)
- Con: We need extra memory per token!

(Image source: https://medium.com/@joaolages/kv-caching-explained-276520203249)

Introducing RoPE

- **Rotary Positional Embedding (RoPE)**
  - A different way to encode position.
  - Doesn't *add* positional vectors to embeddings.
  - Instead, it *rotates* the Q and K vectors based on their position *after* projection, just before the attention dot-product.
- **Key Property:**
  - The attention score for tokens at positions m,n
    - $(RotatedQ(m).\ RotatedK(n)^T)$
      - (T stands for transpose)
    - depends mathematically only on the original Q/K content and the *relative distance* (m - n), not the absolute positions m and n.

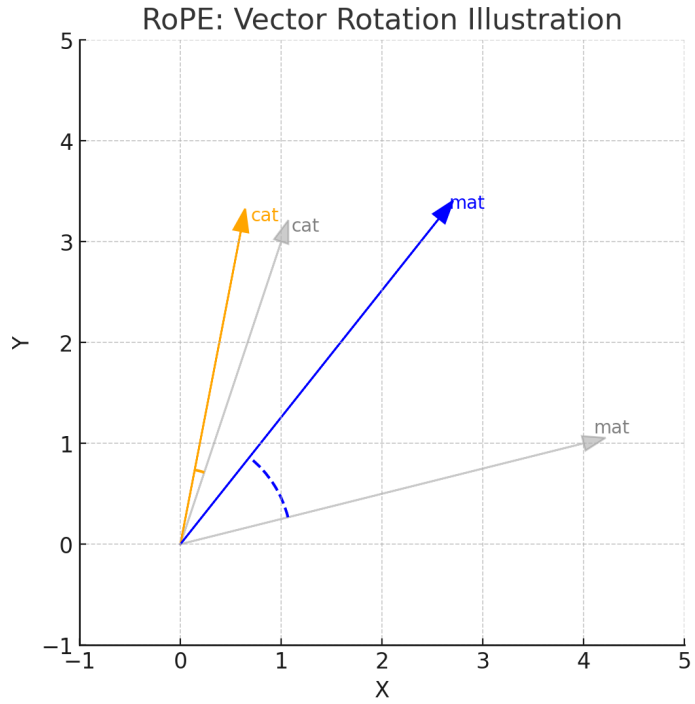


The **cat** sat on the **mat**

Each word is rotated to encode position

RoPE & Caching Advantage

- **Simpler, More Elegant Caching**
  - Calculate base Q, K, V from token embeddings (no positional info added).
  - Cache these **"pure content" K and V vectors**.
- During attention calculation for a new token at position m:
  - Retrieve cached K's (from positions n).
  - *Dynamically* apply RoPE rotation to the new Q using position m.
  - *Dynamically* apply RoPE rotation to the cached K's using their original positions n.
  - Calculate dot products.
- Cache the *un-rotated*, pure K and V for the new token.
- **Advantage:**
  - Cached state (K, V) is position-agnostic.
  - Relative position information is applied "on-the-fly" during attention.
  - Conceptually cleaner, directly models relative positions.

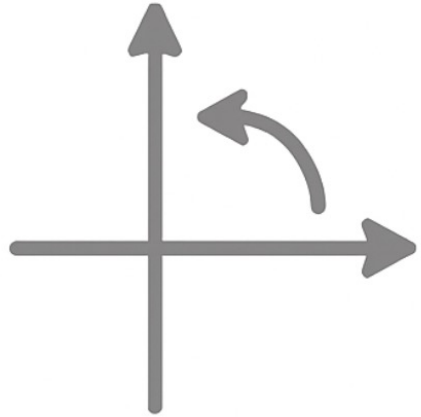


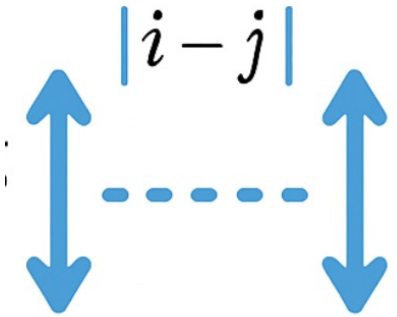
The **cat** sat on the **mat**

Each word is rotated to encode position

RoPE: Rotary Positional Embedding

- **Efficient & Scalable:**
  - RoPE encodes position via rotation, which avoids adding embeddings and works natively within attention.
- **Better Generalization:**
  - Encodes **relative** positions, preserving meaning across contexts and longer sequences.
- **Modern Default:**
  - Used in LLaMA, GPT-NeoX, and other advanced models for high-performance language modeling.





Multi-Head Latent Attention (MLA) in DeepSeek-V3

- MLA (Multi-Head Latent Attention) is a **low-rank compressed version of standard attention**.
- It is designed to:
  - **Reduce KV cache memory** during inference.
  - **Reduce query activation memory** during training.
  - **Match performance** of standard Multi-Head Attention (MHA), but more efficiently.

Remember in Standard Transformer...

- You have access to the **entire input sequence** at once (e.g., 1024 tokens), so you compute attention like this:
  - One big matrix multiplication.
  - Parallel computation across all tokens.
  - Training is fast and fully batched.

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

Remember in Standard Transformer...

- You have access to the **entire input sequence** at once (e.g., 1024 tokens), so you compute attention like this:
  - One big matrix multiplication.
  - Parallel computation across all tokens.
  - Training is fast and fully batched.

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

- But during inference (Generation):
  - Now you're generating **one token at a time**, e.g., for text completion or chat.
- **But recomputing all $K_1 ... K_t$ every time would be slow.**
  - So instead, we:
  - **Cache** $K_1, K_2, ..., K_{t-1}$ and $V_1, V_2, ..., V_{t-1}$
  - Only compute K_t and V_t once
  - Append to the cache
  - Use the cache to compute:

$$\text{Attention}(Q_t, [K_1, ..., K_t], [V_1, ..., V_t])$$

What is KV Caching? Why Optimize It?

- In autoregressive generation:
  - You compute attention one token at a time.
  - You must **cache all previous K and V vectors** for fast computation.
- **Standard Transformer:**
  - K/V stored in full dimension → **expensive**.
- **MLA:**
  - Only cache:
    - c_t^{KV} (compressed key-value vector)
    - $k_t^{(R)}$ (positional key)
  - Reconstruct actual keys/values **on-the-fly** using these small vectors.
- **Result:** Much smaller KV cache → faster inference, lower memory.

Compute Base content

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$

$\mathbf{h}_t$: hidden state at time t

W^{DKV}: projection matrix

$\mathbf{c}_t^{KV}$: base content vector for key/value processing

Compute Multi-head Content Keys

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$

- Project content vector into keys for each attention head
- Split into n_h heads

Compute Rotary Key Vector

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$

$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$

- Apply RoPE to a different projection of the hidden state
- This gives a **position-aware** key embedding (shared or per-head)

Final Key Vector per Head

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$

$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R],$$

For each attention head i, the key is a **concatenation** of:

- Head-specific content key $\mathbf{k}_{t,i}^C$
- Shared rotary positional component $\mathbf{k}_t^R$

Compute Multi-head Values

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$
$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$
$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$
$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R],$$
$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \ldots; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV},$$

- Values are derived from the same $\mathbf{c}_t^{KV}$
- One projection matrix per head

Multi-Head Latent Attention

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$

$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R],$$

$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \ldots; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV},$$

Compressing K,Q,V

- **Why Compress?**
  - Original vector is 4096D → storing this for every token in long sequences is costly.
  - Compressed vector is only 128D → **less memory** and **faster inference**.
  - In DeepSeek-V3, **only c_t^{KV} and $k_t^{(R)}$ are cached** (not the full key/value), saving even more space.

$$c_t^{KV} = W^{DKV} h_t, \quad W^{DKV} \in \mathbb{R}^{128 \times 4096}$$

Then:

- Up-project to **keys**:

$$\mathbf{k}_t^C = W^{UK} \mathbf{c}_t^{KV}$$

- Up-project to **values**:

$$\mathbf{v}_t^C = W^{UV} \mathbf{c}_t^{KV}$$

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV} \mathbf{h}_t,$$

$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK} \mathbf{c}_t^{KV},$$

$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR} \mathbf{h}_t),$$

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R],$$

$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \ldots; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV} \mathbf{c}_t^{KV},$$

What is Low-Rank Compression?

- **In standard attention:**
  - Queries (Q), Keys (K), and Values (V) are **direct linear projections** of the hidden state $h_t \in \mathbb{R}^d$ (where d is large, e.g., 4096).
- **In MLA:**
  - Instead of projecting full-size Q, K, V, we **first reduce the dimensionality**:
  - The compressed version of the hidden state vector, $h_t = c_t \in \mathbb{R}^{d^c}$, where $d^c \ll d$ (e.g., 128 instead of 4096)
  - This is called **low-rank compression**.
- **Why?**
  - It saves:
    - Memory for storing K/V during inference.
    - Activation memory for Q during training.

$$\boxed{\mathbf{c}_t^{KV}} = W^{DKV}\mathbf{h}_t,$$
$$[\mathbf{k}_{t,1}^C; \mathbf{k}_{t,2}^C; \ldots; \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV},$$
$$\boxed{\mathbf{k}_t^R} = \mathrm{RoPE}(W^{KR}\mathbf{h}_t),$$
$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R],$$
$$[\mathbf{v}_{t,1}^C; \mathbf{v}_{t,2}^C; \ldots; \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV},$$

How Compression Works in DeepSeek:

- **Start with full hidden state** $h_t \in \mathbb{R}^{4096}$.
- **Apply a down-projection matrix**:

$$\mathbf{k}_t = W^K \mathbf{h}_t \qquad \mathbf{v}_t = W^V \mathbf{h}_t$$

Standard MHA

Requires caching **both K and V** per token

$$c_t^{KV} = W^{DKV} h_t, \quad W^{DKV} \in \mathbb{R}^{128 \times 4096}$$

Then:

- Up-project to **keys**:

$$\mathbf{k}_t^C = W^{UK} \mathbf{c}_t^{KV}$$

- Up-project to **values**:

$$\mathbf{v}_t^C = W^{UV} \mathbf{c}_t^{KV}$$

Multi Latent Attention (MLA)

Only c_t^{KV} is cached — saving memory

Multi-Head Latent Attention (MLA)

(1) Compress hidden state for keys/values

$$\mathbf{c}_t^{KV} = W^{DKV}\mathbf{h}_t$$

(2) Up-project to get per-head keys

$$[\mathbf{k}_{t,1}^C, \mathbf{k}_{t,2}^C, ..., \mathbf{k}_{t,n_h}^C] = \mathbf{k}_t^C = W^{UK}\mathbf{c}_t^{KV}$$

(3) Compute positional keys with RoPE

$$[\mathbf{k}_{t,1}^R, \mathbf{k}_{t,2}^R, ..., \mathbf{k}_{t,n_h}^R] = \mathbf{k}_t^R = \mathrm{RoPE}(W^{KR}\mathbf{h}_t)$$

(4) Final key per head = content + position

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_{t,i}^R]$$

(5) Up-project to get per-head values

$$[\mathbf{v}_{t,1}^C, \mathbf{v}_{t,2}^C, ..., \mathbf{v}_{t,n_h}^C] = \mathbf{v}_t^C = W^{UV}\mathbf{c}_t^{KV}$$

Symbol	Description
$\mathbf{h}_t \in \mathbb{R}^d$	Hidden state at time step `t`
d	Embedding dimension
n_h	Number of attention heads
d_h	Dimension per head (`d = n_h × d_h`)
$d_c \ll d$	Compressed dimension
$\mathbf{c}_t^{KV} \in \mathbb{R}^{d_c}$	Compressed latent for keys & values
$W^{DKV} \in \mathbb{R}^{d_c \times d}$	Down-projection matrix
$W^{UK}, W^{UV} \in \mathbb{R}^{d_h n_h \times d_c}$	Up-projection matrices for keys and values
$W^{KR} \in \mathbb{R}^{d_h n_h \times d}$	Projects hidden state for RoPE
$\mathrm{RoPE}(\cdot)$	Rotary Positional Embedding
$\mathbf{k}_t^C, \mathbf{k}_t^R \in \mathbb{R}^{d_h n_h}$	Content and positional keys
$\mathbf{k}_{t,i} \in \mathbb{R}^{2d_h}$	Final key for head `i` (concatenation)
$\mathbf{v}_t^C \in \mathbb{R}^{d_h n_h}$	Values for all heads

Why it's called "latent"

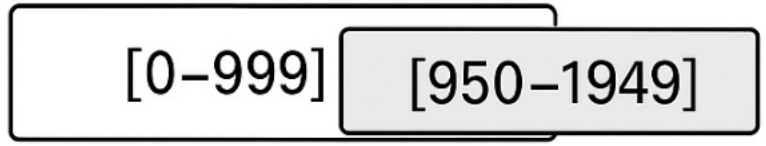
- Latent refers to **"Intermediate, hidden representations** in a neural network
  - Not directly observable, but inferred from data
  - Capture **abstract features**, patterns, or concepts
- MLA
  - h_t Hidden state — a latent vector encoding all context learned so far
  - C_t^{KV} Latent content vector — not directly observable, used internally to compute attention
  - Key, Value vectors: Latent because they represent how to relate tokens — not exposed directly
- **Other examples in ML**
  - **Latent space** in autoencoders (compressed internal representation)
  - **Latent variables** in probabilistic models (e.g. topic distributions in LDA)

Long Sequences

- Transformers have **quadratic time & memory** complexity: $O(n^2)$
  - Sequences like **books, code, or genomes** can be **10k+ tokens**
  - Solution: **Split into chunks** to fit in memory and allow parallelization
- **Naive Chunking:**
  - Input split into non-overlapping blocks : [Token 0–999], [1000–1999], etc.
  - **Problem:**
    - **No attention across chunks**
    - Tokens **lose access to global context**
    - Model can't relate distant concepts

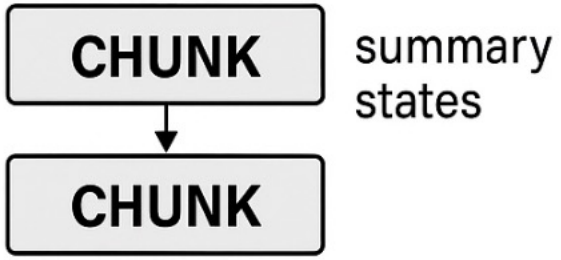
Solutions to Preserve Context

- **Overlapping Chunks**
  - Add **token overlap** between chunks
  - Example: [0–999], [950–1949]
  - Preserves partial context

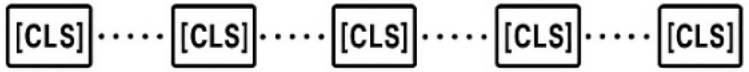


Add token overlap between chunks
Example: [0–999], [950–1949]
Preserves partial context

- **Recurrent Memory**
  - Pass **summary states** across chunks (Transformer-XL)
  - Maintains long-term dependencies

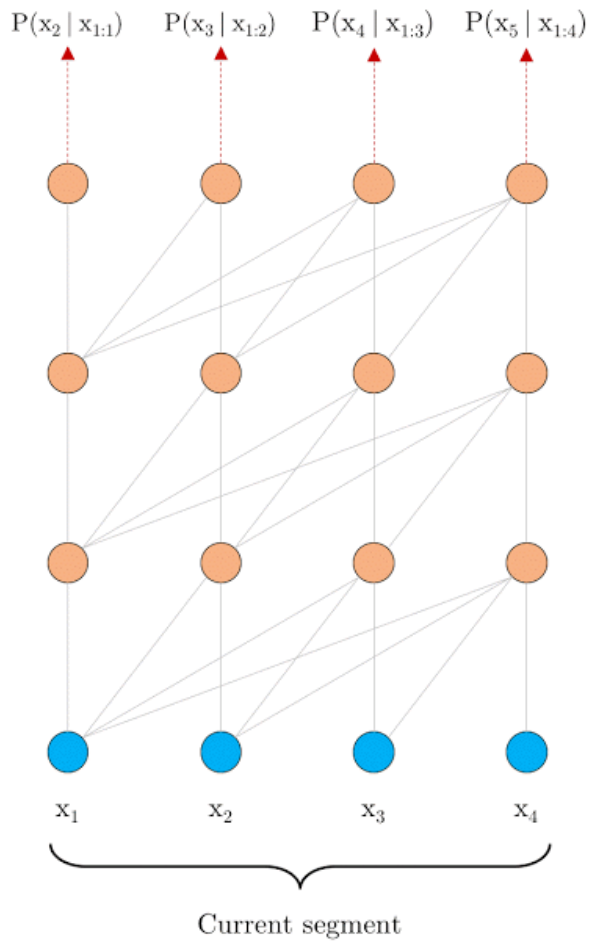


- **Sparse or Global Attention**
  - Use **special tokens** that attend across chunks (e.g., [CLS] tokens in Longformer/BigBird)
  - Global views with local efficiency

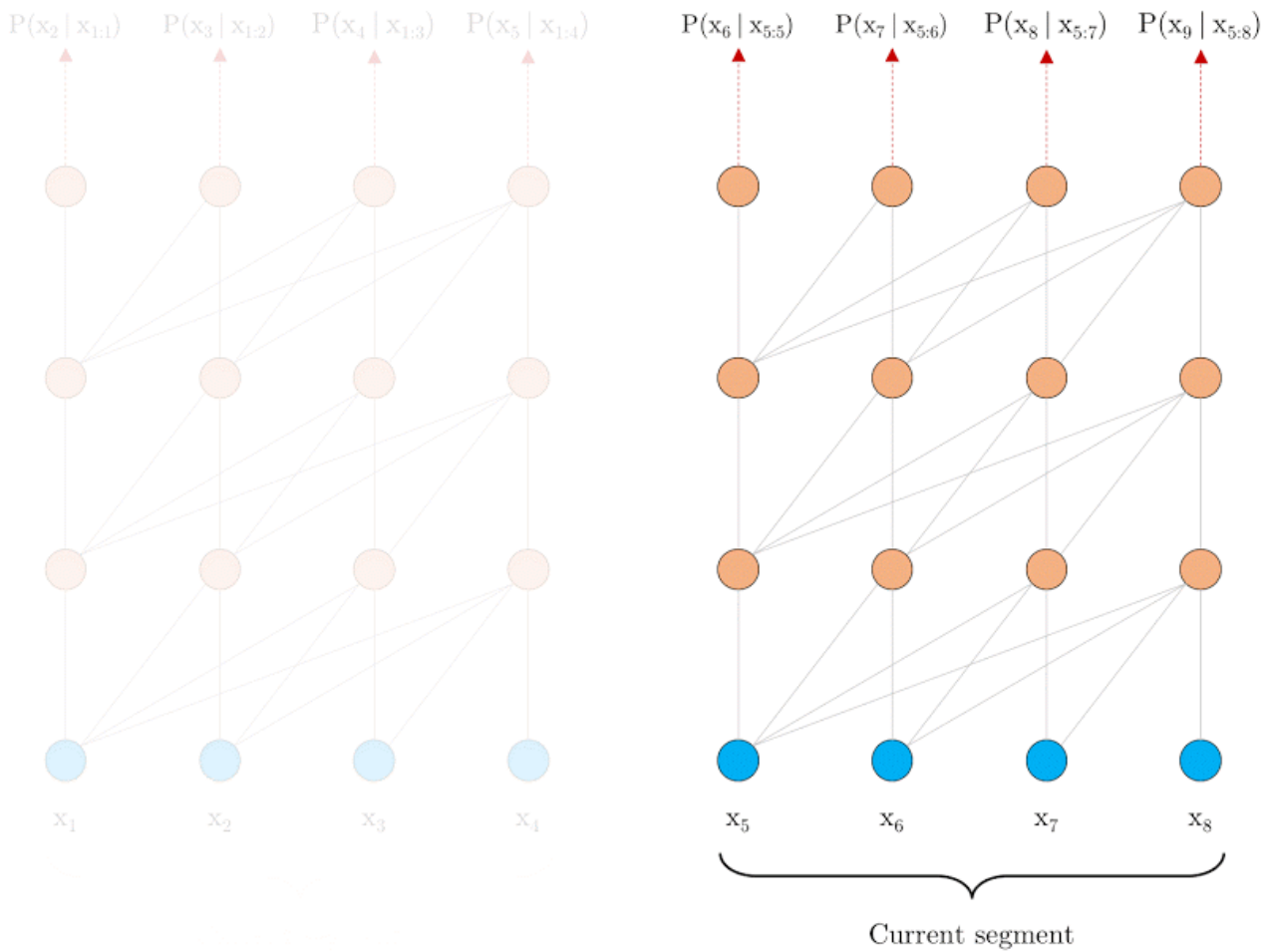


The Problem: Limited Context in Transformers

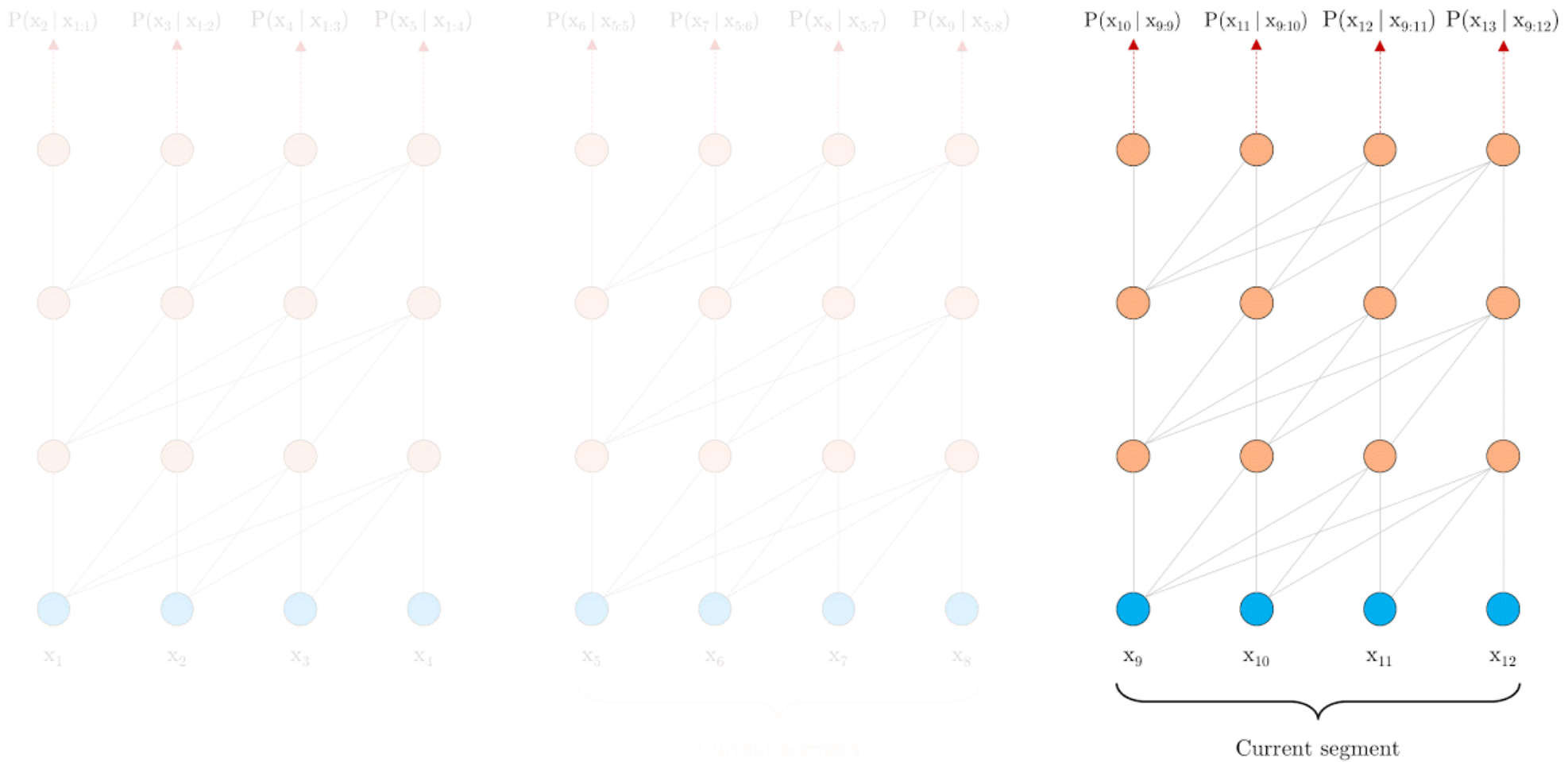
- Standard Transformers process sequences in fixed-length segments.
- No information flow between segments during training.
- Leads to:
  - Inability to capture long-range dependencies.
  - Context fragmentation during evaluation.
  - Inefficient evaluation.



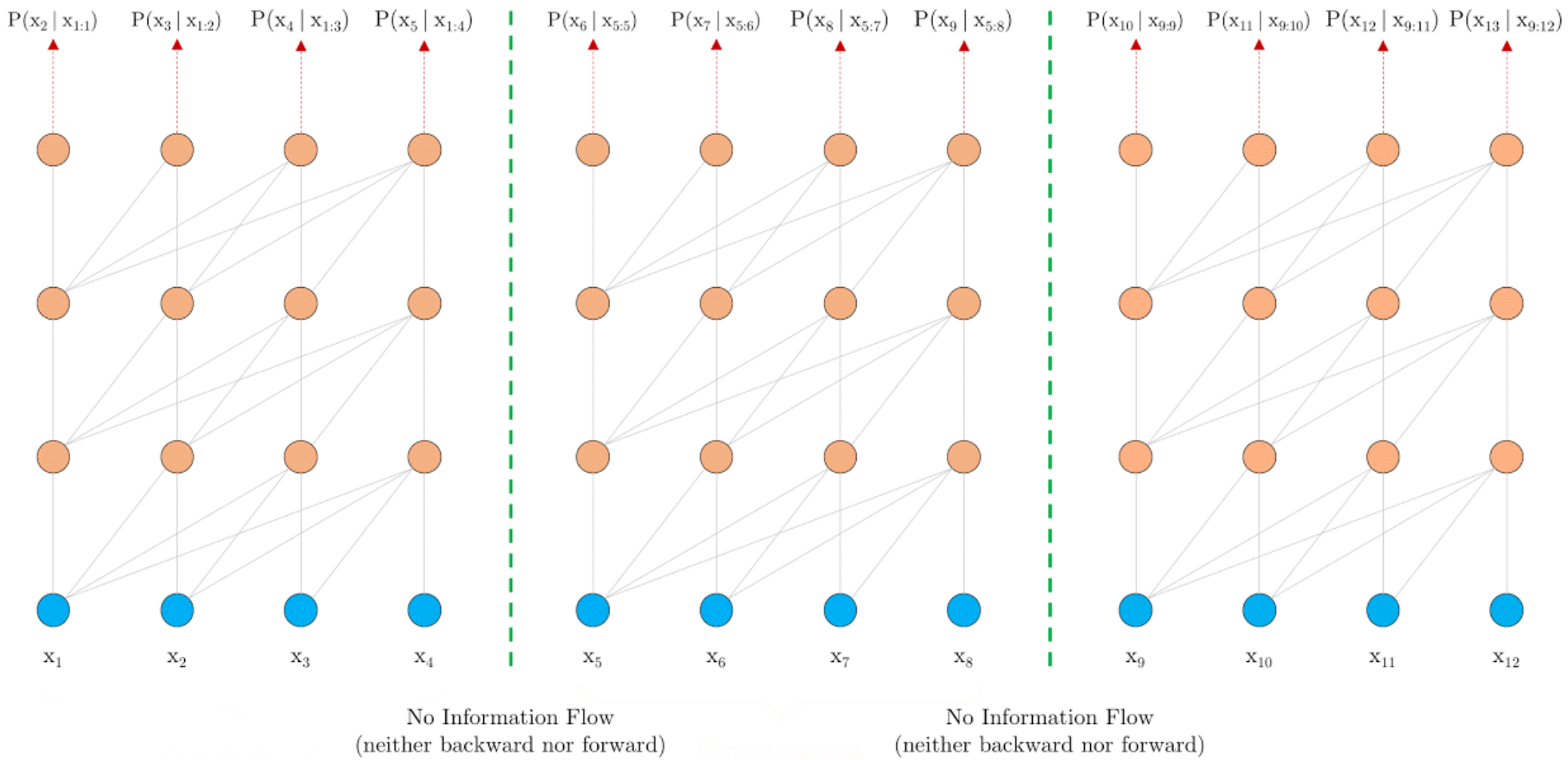
Vanilla Transformer with a fixed-length context at training time.



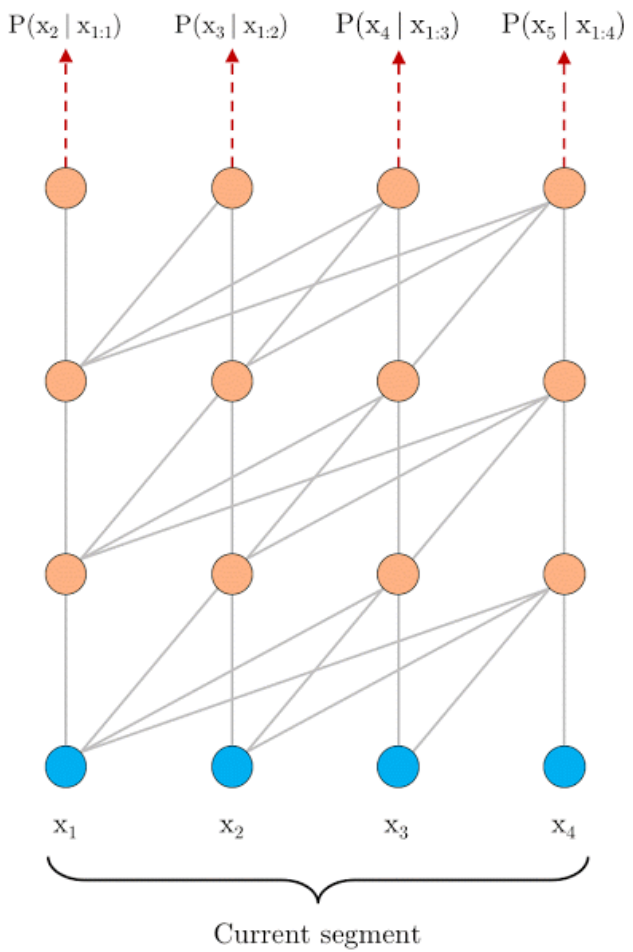
Vanilla Transformer with a fixed-length context at training time.



Vanilla Transformer with a fixed-length context at training time.

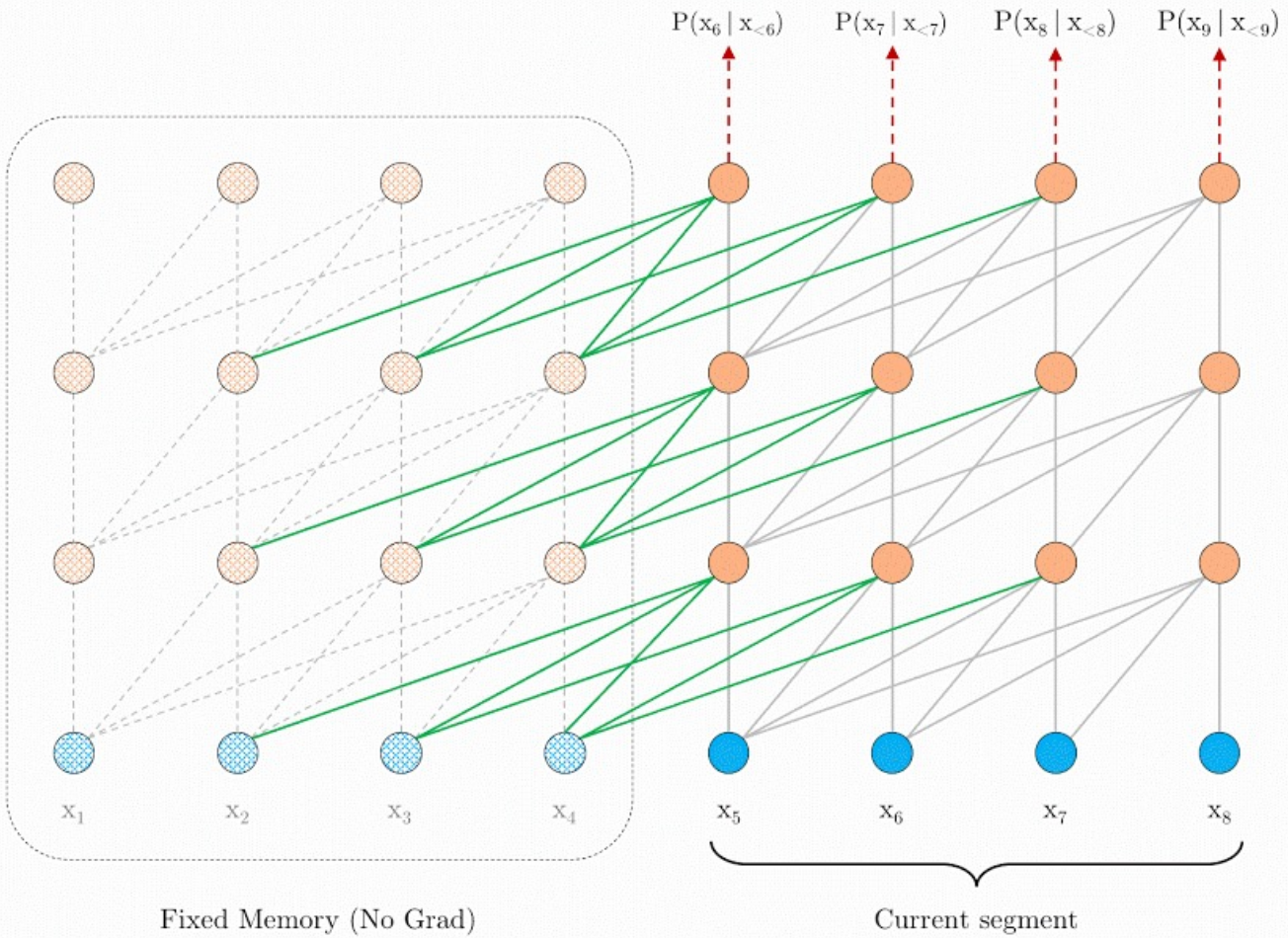


Vanilla Transformer with a fixed-length context at training time.



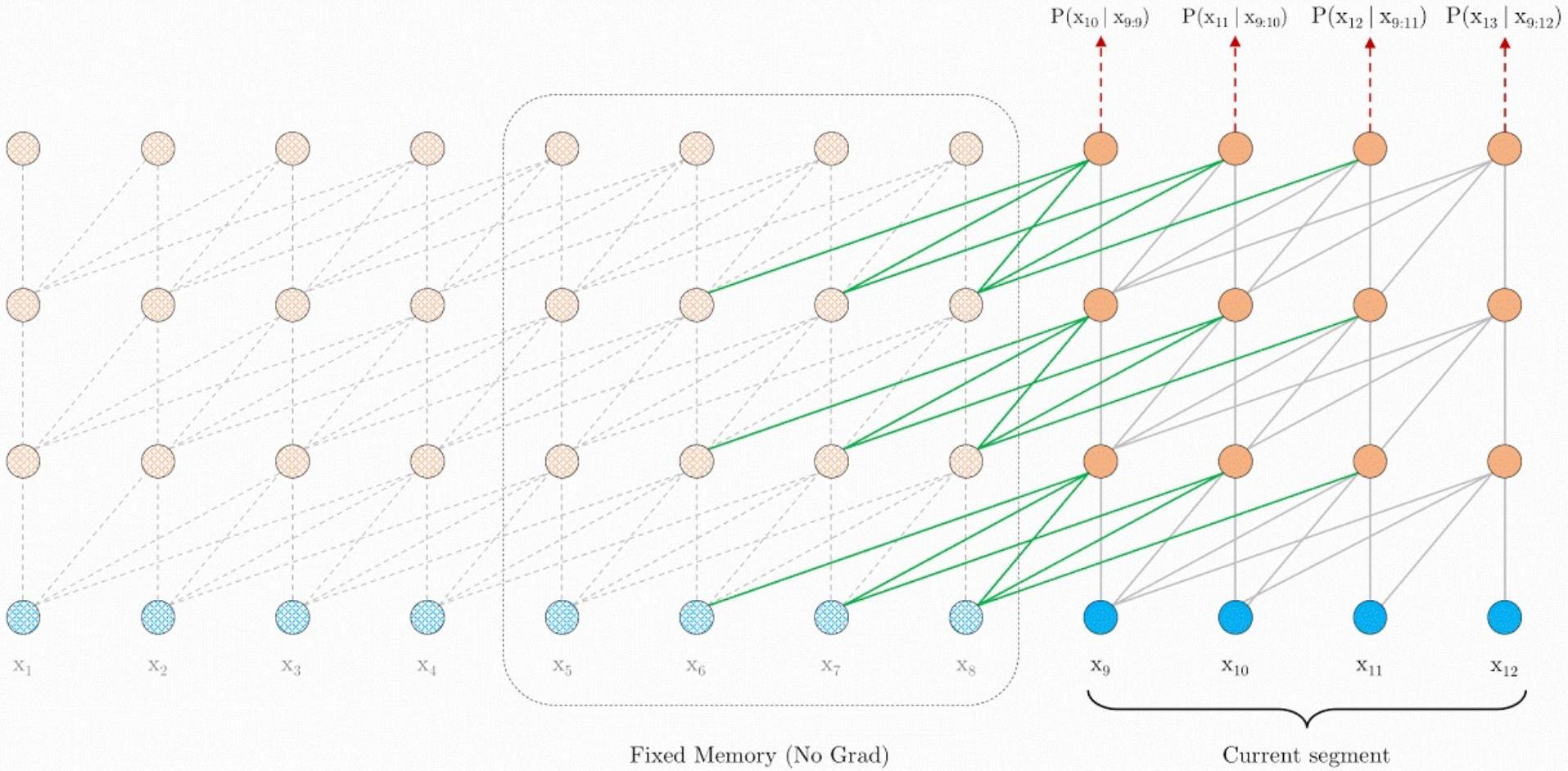
Transformer-XL with segment-level recurrence at training time.

Improve Transformer's ability to model **long-term dependencies** by **reusing hidden states from previous segments**.



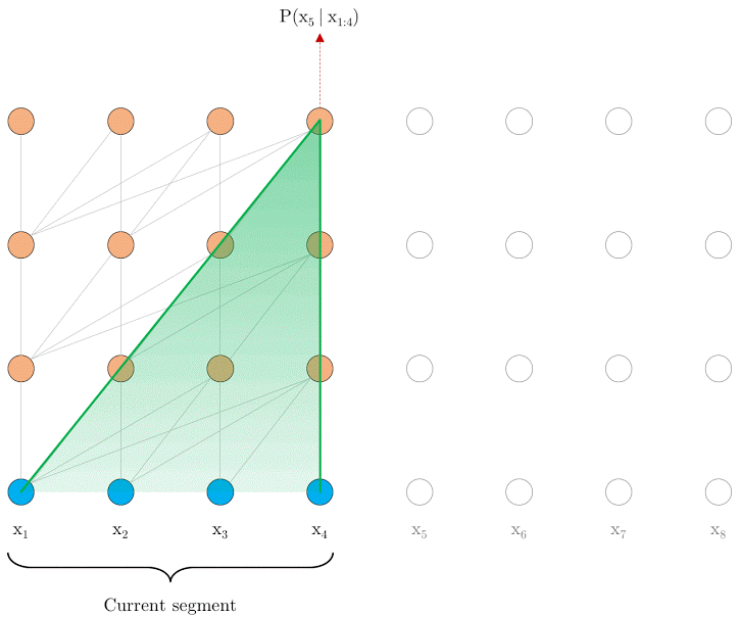
Transformer-XL with segment-level recurrence at training time.

Improve Transformer's ability to model **long-term dependencies** by **reusing hidden states from previous segments**.

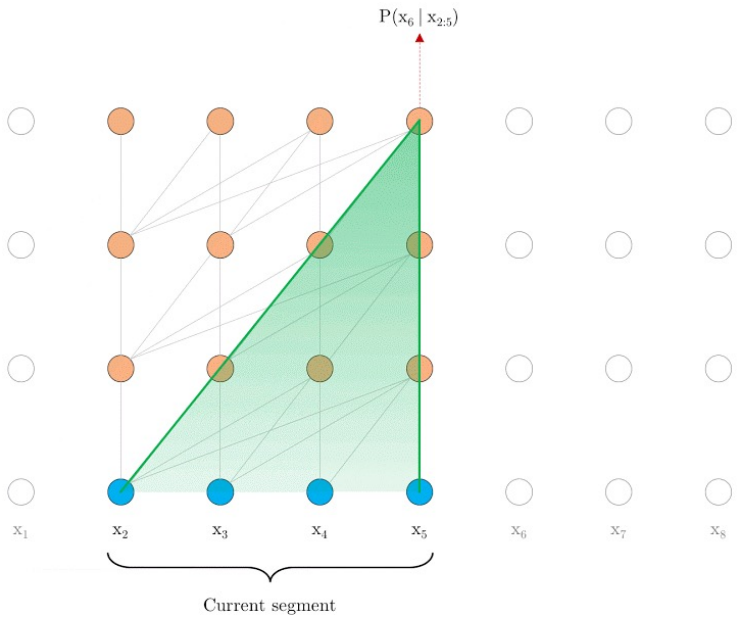


Transformer-XL with segment-level recurrence at training time.

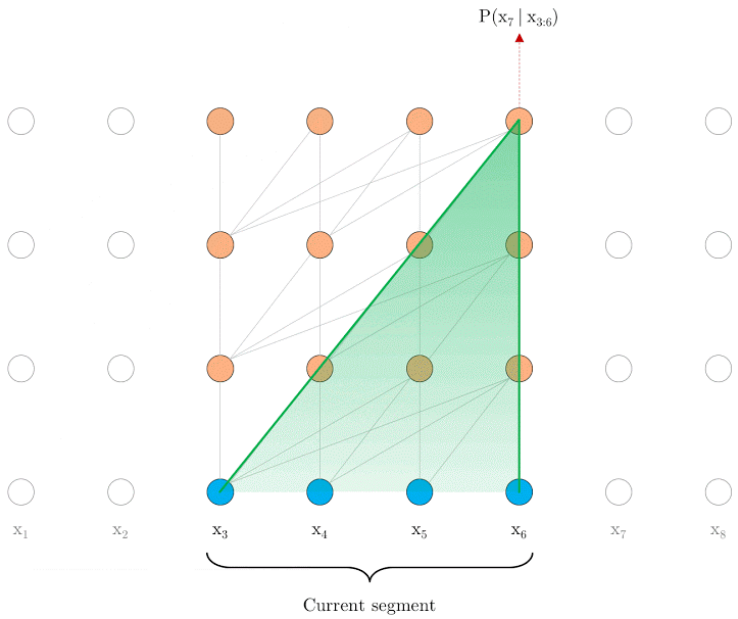
Improve Transformer's ability to model **long-term dependencies** by **reusing hidden states from previous segments**.



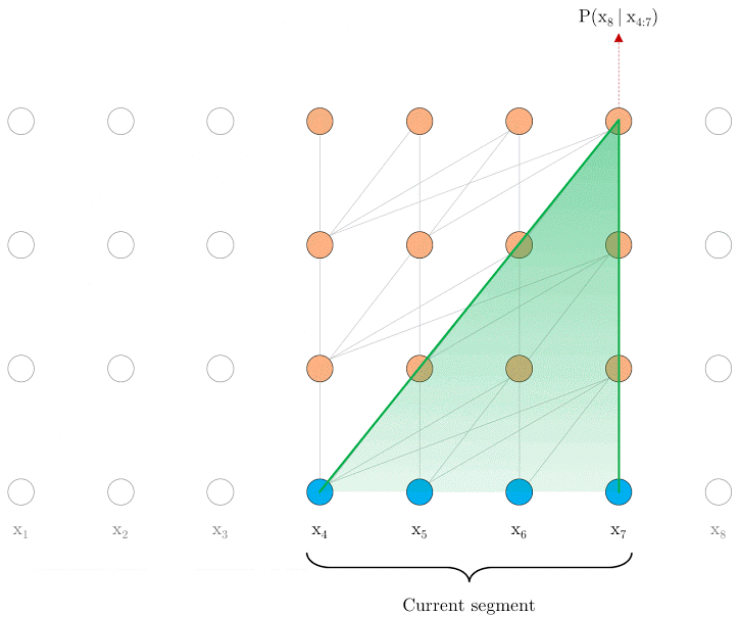
Vanilla Transformer with a fixed-length context at evaluation time.



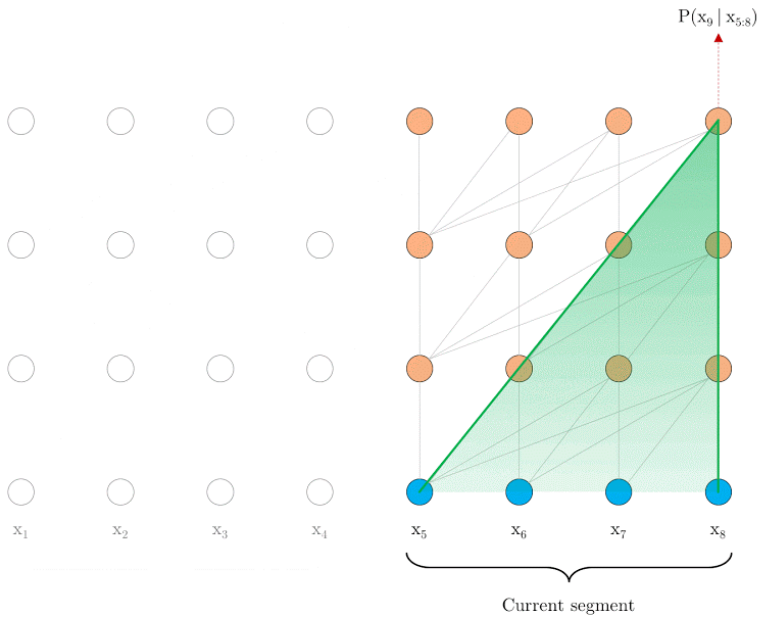
Vanilla Transformer with a fixed-length context at evaluation time.



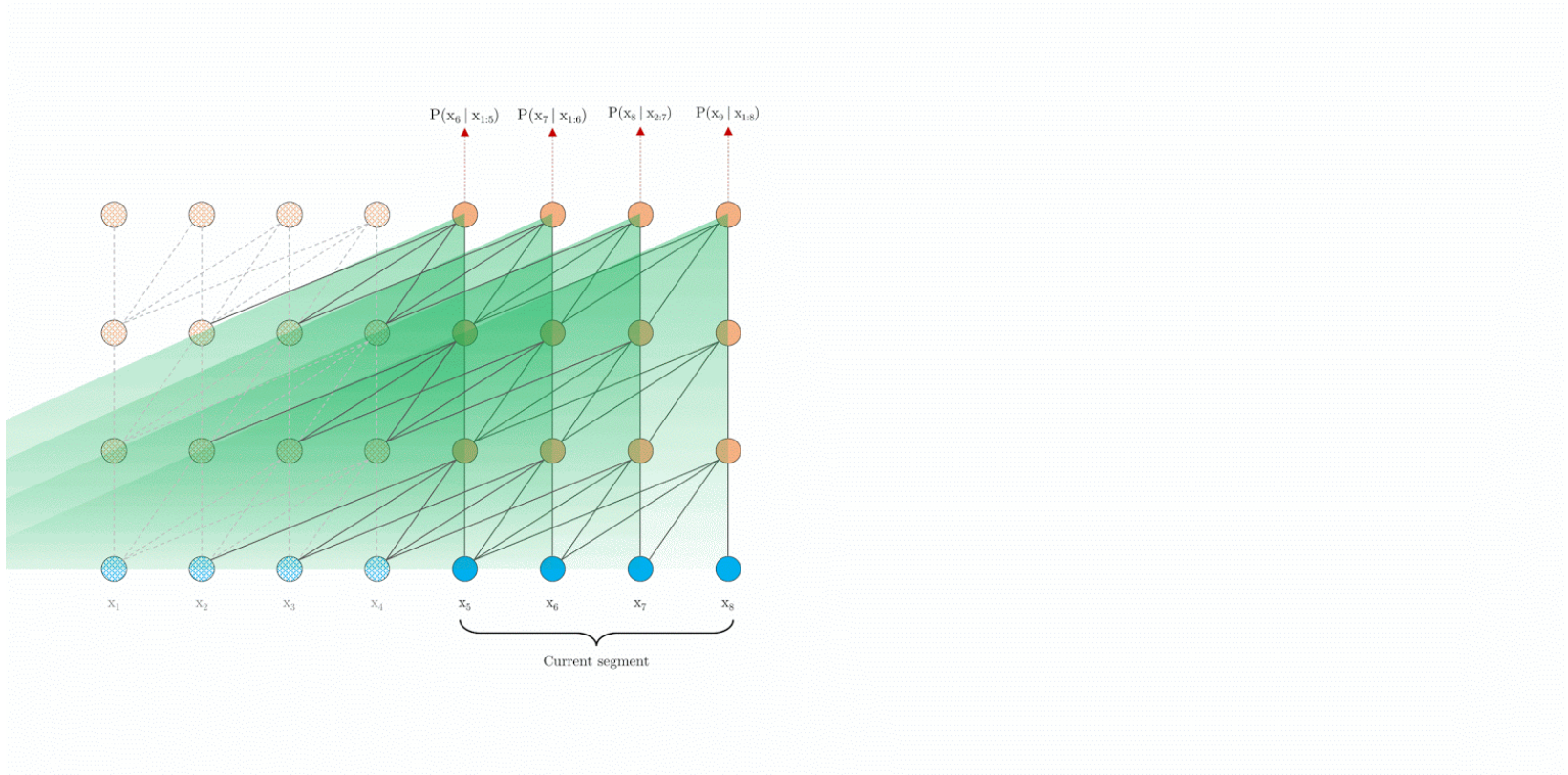
Vanilla Transformer with a fixed-length context at evaluation time.



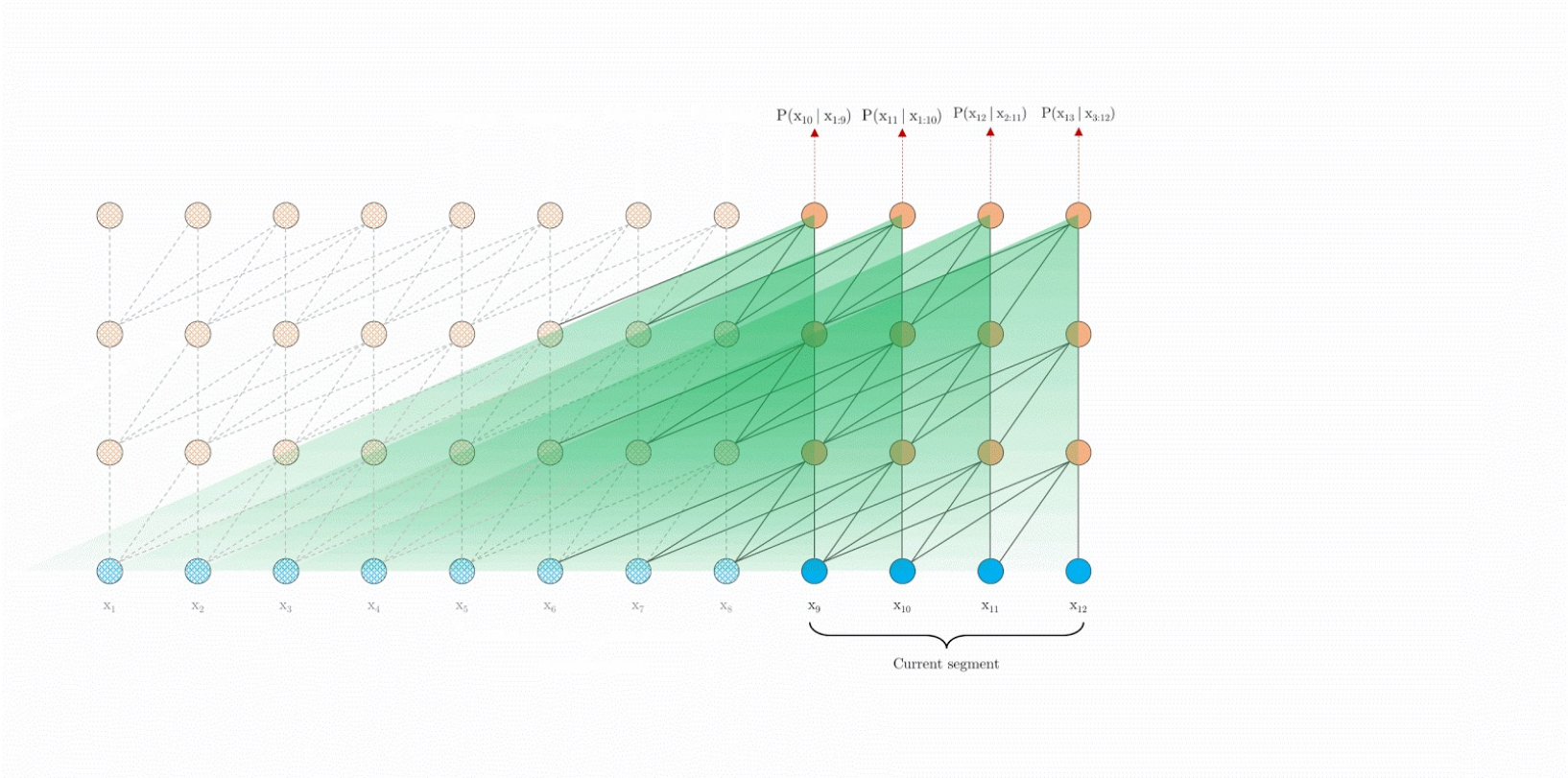
Vanilla Transformer with a fixed-length context at evaluation time.



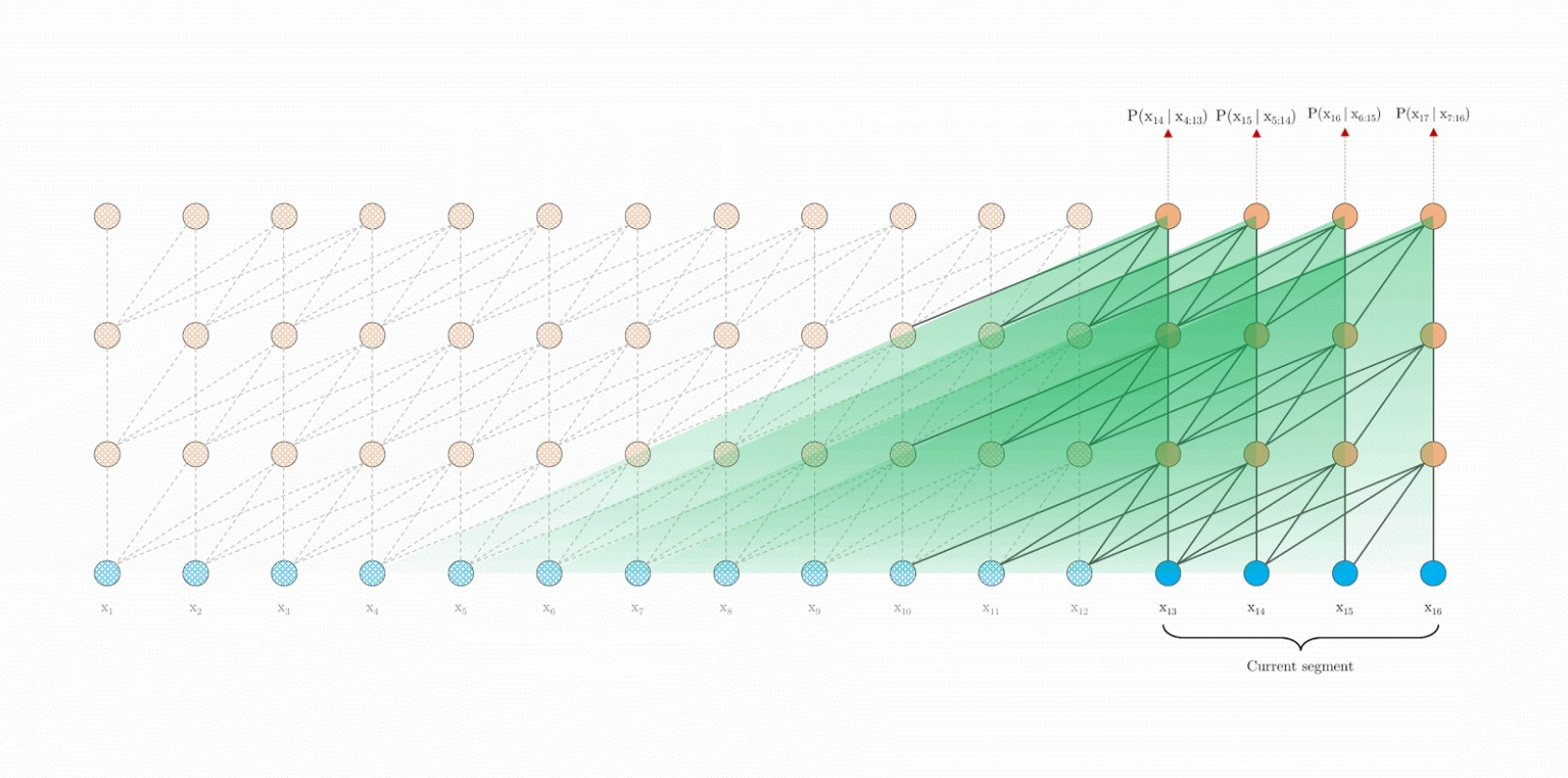
Vanilla Transformer with a fixed-length context at evaluation time.



Transformer-XL with segment-level recurrence at evaluation time.



Transformer-XL with segment-level recurrence at evaluation time.



Transformer-XL with segment-level recurrence at evaluation time.

Relative Positional Encoding

- In long sequences, we **divide data into fixed-length segments** during training/inference. For example:
  - Segment 1: tokens $x_0, x_1, \ldots, x_9$
  - Segment 2: tokens $x_{10}, x_{11}, \ldots, x_{19}$
- **Segments shift when we move to the next chunk of input**, like in streaming text or large documents.
- **Problem with absolute positions:**
  - Segment 1: token x0 has absolute position 0
  - Segment 2: token x10 has absolute position 0 **again**, if we reset every segment
  - This breaks the continuity of attention across segments

For example, consider an old segment with contextual positions [0, 1, 2, 3]. When a new segment is processed, we have positions [0, 1, 2, 3, 0, 1, 2, 3] for the two segments combined, where the semantics of each position id is incoherent through out the sequence.

Relative Positional Encoding

- **Core Idea:** Instead of encoding absolute position (i), encode the *relative distance* (i-j) between the query token (i) and the key token (j).
- **Consistency:** The distance between two tokens remains the same regardless of which segment they appear in or which states are being reused.
  - Encodes "How far away is token j from token i?"

The quick brown fox jumps over the lazy dog

Absolute 4 5 8

Relative

jump to fox: 1
jump to lazy: -3

Relative Positional Encoding Used in Transformer-XL

- x_i, x_j: input token vectors at positions i, j
- W_Q, W_K, W_R: learnable projection matrices
- r_{i-j}: **sinusoidal embedding** of relative distance $i - j$ (non-learnable)
- u, v: trainable global vectors (shared across tokens)

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

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

$$A_{i,j} = Q_i \cdot K_j^T = (x_i W_Q)(x_j W_K)^T$$

Score between query xi and key xj

$$A_{i,j} = \underbrace{x_i^T W_Q^T W_K x_j}_{\text{content-to-content}} + \underbrace{x_i^T W_Q^T W_R r_{i-j}}_{\text{content-to-position}} + \underbrace{u^T W_K x_j}_{\text{global content bias}} + \underbrace{v^T W_R r_{i-j}}_{\text{global position bias}}$$

Standard query-key interaction

Query attends to relative position embedding

Bias based on key content

Bias based on position offset

Content to Content Attention

- **What it is:**
  - The standard dot product between the query and key.
  - Captures the similarity between token xi (query) and token xj (key).
- **Role:**
  - Purely based on **token content**.
  - Found in the original Transformer.

$$A_{i,j} = \underbrace{x_i^T W_Q^T W_K x_j}_{\text{content-to-content}} + \underbrace{x_i^T W_Q^T W_R r_{i-j}}_{\text{content-to-position}} + \underbrace{u^T W_K x_j}_{\text{global content bias}} + \underbrace{v^T W_R r_{i-j}}_{\text{global position bias}}$$

Content-to-Position Attention

- **What it is:**
  - r_{i-j} represents the **relative distance** between i and j.
  - We project both the query and the relative position embedding, then take their dot product.
  - Allows the query to attend to **relative positions**, not absolute.
  - Crucial for enabling consistent attention when segments shift or memory is reused.
  - Same r would be used even if segment started at a different index.

$i-j$	dim 0 (sin)	dim 1 (cos)	dim 2 (sin)	dim 3 (cos)	dim 4 (sin)	dim 5 (cos)
-3	-0.1411	-0.9900	-0.2939	-0.9558	-0.4339	-0.9010
-2	-0.0900	-0.9959	-0.1960	-0.9807	-0.2963	-0.9551
-1	-0.0475	-0.9988	-0.1036	-0.9946	-0.1581	-0.9874
0	0.0000	1.0000	0.0000	1.0000	0.0000	1.0000
+1	0.0475	0.9988	0.1036	0.9946	0.1581	0.9874
+2	0.0900	0.9959	0.1960	0.9807	0.2963	0.9551
+3	0.1411	0.9900	0.2939	0.9558	0.4339	0.9010

$$A_{i,j} = \underbrace{x_i^T W_Q^T W_K x_j}_{\text{content-to-content}} + \underbrace{x_i^T W_Q^T W_R r_{i-j}}_{\text{content-to-position}} + \underbrace{u^T W_K x_j}_{\text{global content bias}} + \underbrace{v^T W_R r_{i-j}}_{\text{global position bias}}$$

Global Content Bias

- **What it is:**
  - u is a **learned global content bias vector**, shared across all queries.
  - Computes a bias based on the **key token's content** xj.
- **Role:**
  - Adds a fixed influence based on **what the key is**, not on the query.
  - Helps stabilize attention scores.

$$A_{i,j} = \underbrace{x_i^T W_Q^T W_K x_j}_{\text{content-to-content}} + \underbrace{x_i^T W_Q^T W_R r_{i-j}}_{\text{content-to-position}} + \underbrace{u^T W_K x_j}_{\text{global content bias}} + \underbrace{v^T W_R r_{i-j}}_{\text{global position bias}}$$

Global Position Bias

- **What it is:**
  - v is a **learned global position bias vector**.
  - Applied to the relative position embedding r_{i-j} .
- **Role:**
  - Adds a fixed bias for each **relative position** (e.g., -1, 0, +2).
  - Helps encode structural inductive bias — e.g., maybe recent tokens are more important.

$$A_{i,j} = \underbrace{x_i^T W_Q^T W_K x_j}_{\text{content-to-content}} + \underbrace{x_i^T W_Q^T W_R r_{i-j}}_{\text{content-to-position}} + \underbrace{u^T W_K x_j}_{\text{global content bias}} + \underbrace{v^T W_R r_{i-j}}_{\text{global position bias}}$$