Mistral explained (7B)

Prerequisites

- Structure of the Transformer model and how the attention mechanism works.

Topics

- Architectural differences between the vanilla Transformer and Mistral
- Rotary Positional Encoding
- Grouped Query Attention
- KV-Cache
  - Review of self-attention
  - Receptive field
- Sliding Window Attention
  - Motivation
  - How it works
  - Rolling Buffer Cache
  - Pre-fill and chunking

Mistral 7B

- **Mistral 7B is a dense decoder-only Transformer model that introduces several optimizations over standard architectures.**
- **It improves efficiency, inference speed, and memory usage while maintaining high performance.**

- Mistral 7B outperforms the best open 13B model (Llama 2) across all evaluated benchmarks, and the best released 34B model (Llama 1) in reasoning, mathematics, and code generation.

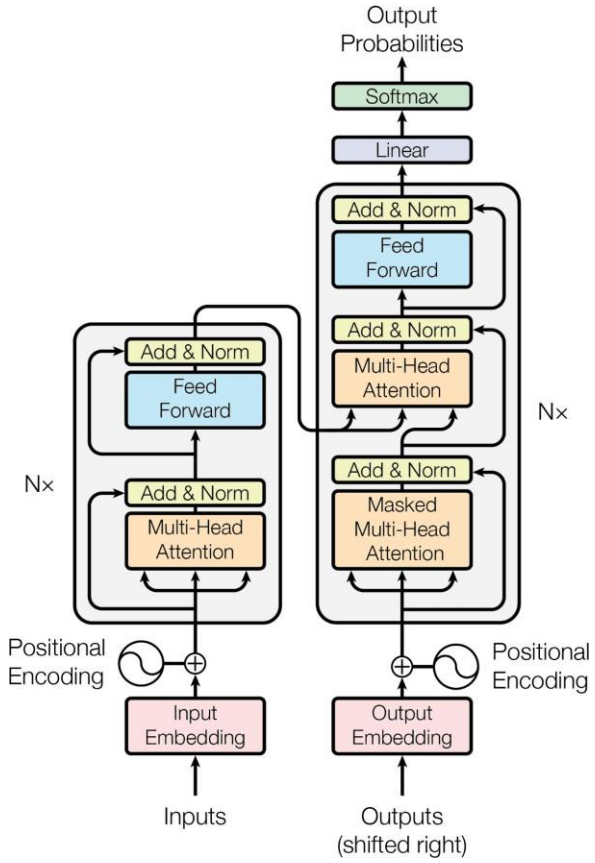
Prerequisites

- Structure of the Transformer model and how the attention mechanism works.

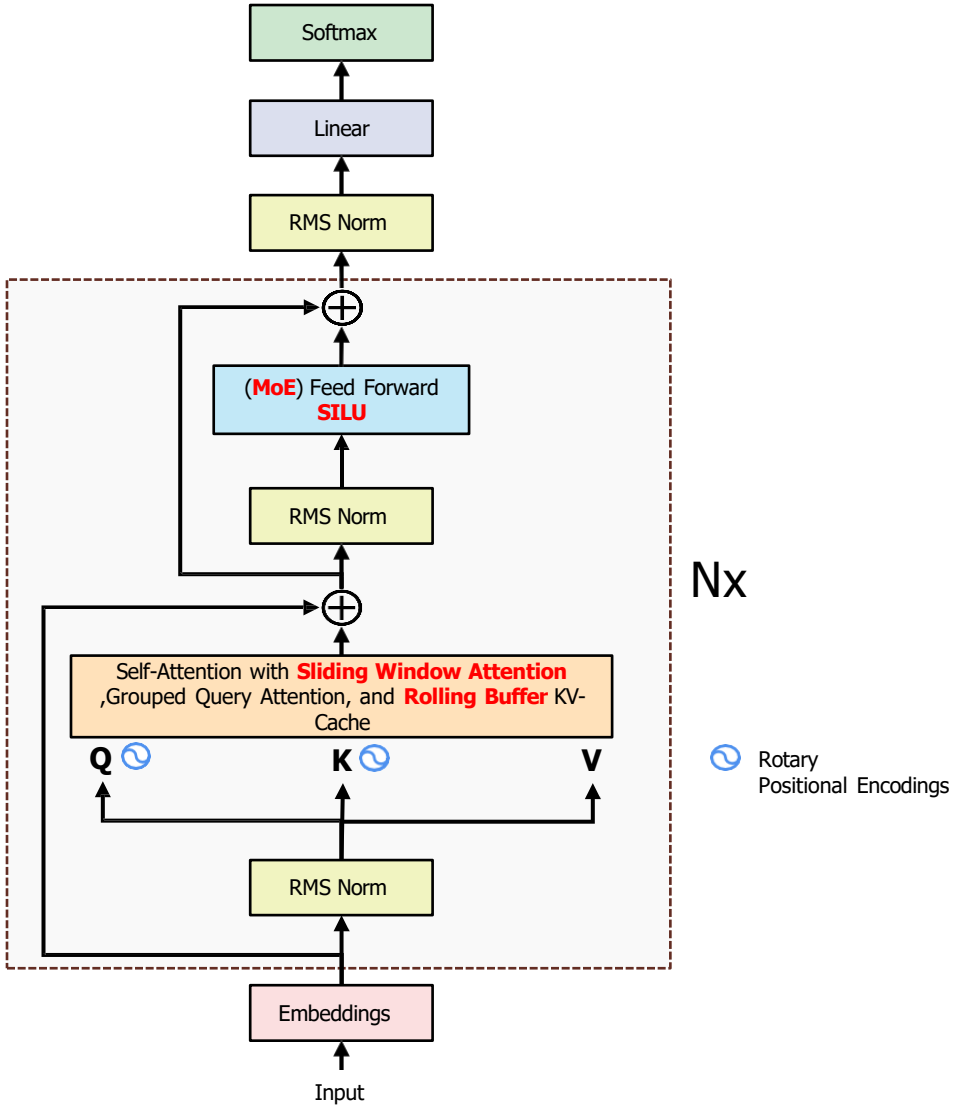
Topics

- Architectural differences between the vanilla Transformer and Mistral
- Rotary Positional Encoding
- Grouped Query Attention
- KV-Cache
  - Review of self-attention
  - Receptive field
- Sliding Window Attention
  - Motivation
  - How it works
  - Rolling Buffer Cache
  - Pre-fill and chunking

Transformer vs Mistral



Transformer
("Attention is all you need")



Mistral

Umar Jamil – https://github.com/hkproj/mistral-llm-notes

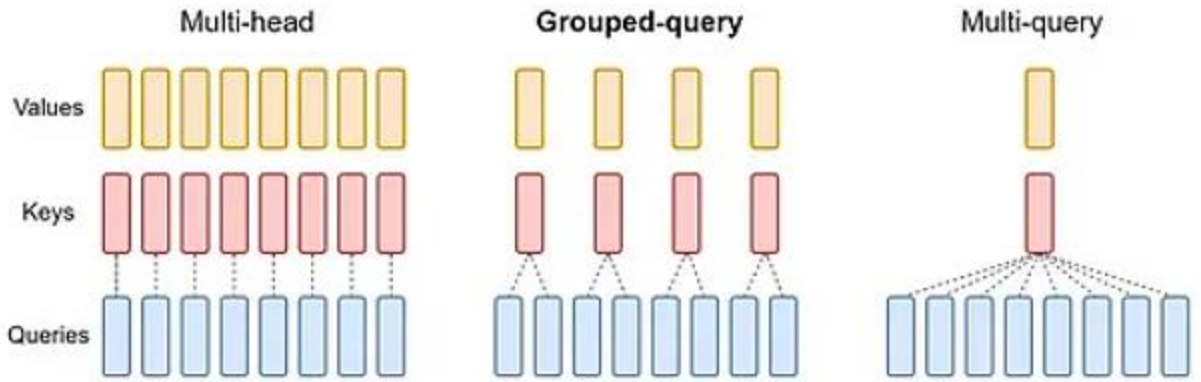
Models

Parameter	Description	Value (7B)	Value (8x7B)	Notes
dim	Size of the embedding vector	4096	4096	
n_layers	Number of encoder layers	32	32	
head_dim	dim / n_heads	128	128	
hidden_dim	Hidden dimension in the feedforward layer	14336	14336	
n_heads	Number of attention heads (Q)	32	32	Different numbers because of Grouped Query Attention
n_kv_heads	Number of attention heads (K, V)	8	8	
windows_size	Size of the sliding window for attention calculation and Rolling Cache	4096	N / A	window_size is not set in the params.json of the 8x7B model
context_len	Context on which model was trained upon	8192	32000	For 8x7B, values are from the official announcement page
vocab_size	Number of tokens in the vocabulary	32000	32000	The tokenizer is the SentencePiece tokenizer
num_experts_per_tok	Number of expert models for each token	N / A	2	Sparse Mixture of Experts only available for 8x7B
num_experts	Number of exper models	N / A	8	

Grouped-Query Attention (GQA)

- Standard multi-head attention (MHA) operates by creating multiple independent query-key-value heads. However, this is computationally expensive.
- Mistral 7B replaces MHA with Grouped-Query Attention (GQA) to improve efficiency.

- If we have 8 query heads but only 2 key-value heads, then each key-value head is shared among 4 query heads.



Grouped-Query Attention (GQA)

- Instead of each query attending to its own set of keys and values, multiple queries share a smaller number of key-value groups.
- This reduces memory overhead while keeping quality close to full multi-head attention.
- Faster inference and more efficient caching.

- Each token in a sequence is different, meaning each one should have a unique query to fetch relevant information.
- If we shared queries, we would lose the ability to focus on different parts of the sequence dynamically.

Why Does Sharing K/V Work?

Reduces computation

- Instead of computing a separate K/V for every head, we compute a smaller number of K/V groups.
- This results in faster inference and reduced memory usage.

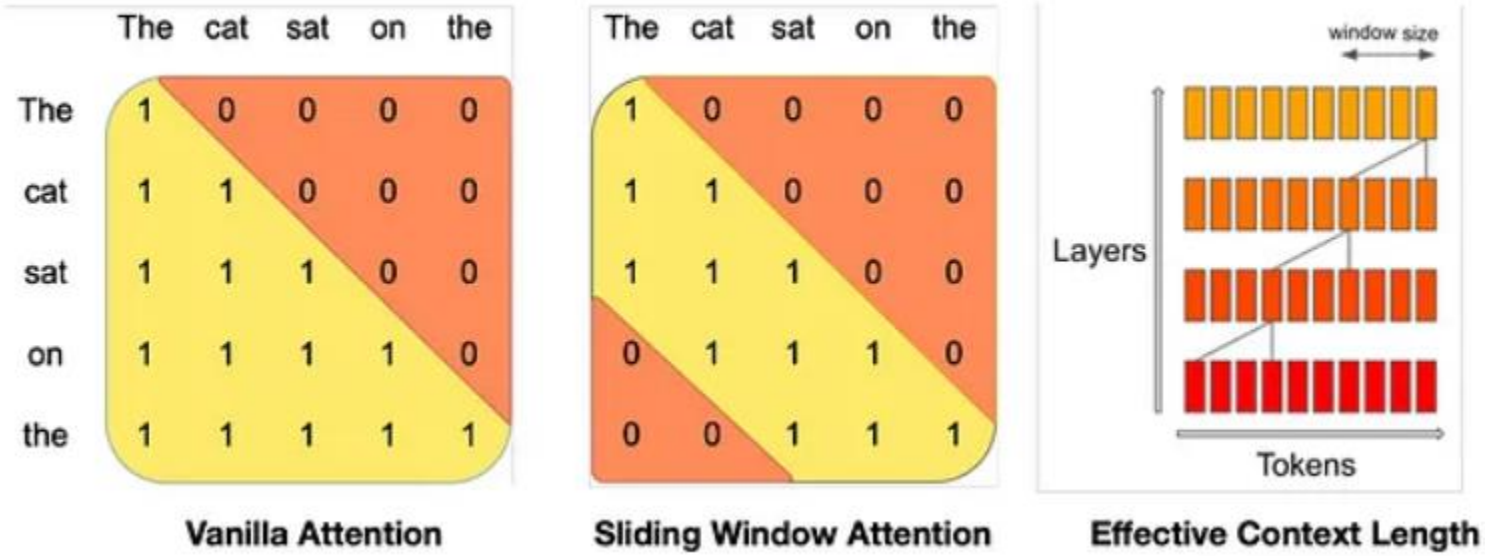
Does not harm model performance

- Studies show that attention patterns are often highly correlated, meaning many heads attend to similar information.
- Instead of computing redundant K/V, we reuse them across multiple queries.

Attention Type	# of Queries (Q)	# of Keys (K)	# of Values (V)	How Keys & Values are Shared?	Computational Cost
Multi-Head Attention (MHA)	8	8	8	Each head has its own Q, K, and V	**High** (full computation for all heads)
Grouped-Query Attention (GQA) (4 groups)	8	4	4	Multiple heads share the same K/V (4 groups, each serving 2 heads)	**Moderate** (faster than MHA but still retains diversity)
Multi-Query Attention (MQA)	8	1	1	All heads share the **same** K/V	**Lowest** (fastest but less diverse attention)

Sliding Window Attention

- (Handles long-context more efficiently)
- Traditional self-attention has a quadratic complexity ($O(n^2)$) due to each token attending to all others. This becomes infeasible for long-context LLMs.
- Mistral 7B applies Sliding Window Attention (SWA) to process longer inputs efficiently.



How SWA Works:

Each token only attends to a fixed-size window (W) of preceding tokens instead of the entire sequence

- This significantly reduces computational complexity while preserving local coherence.
- Works well for tasks like summarization or dialogue generation where nearby words are most relevant.
- Reduces the computational complexity from $O(n^2)$ to $O(nW)$.
- The model automatically retains past information through the hidden states that flow from one layer to the next.

What is Self-Attention?

Self-Attention allows the model to relate words to each other. Imagine we have the following sentence: "**The cat is on a chair**"

Here I show the product of the Q and the K matrix **before** we apply the softmax.

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

Q (6, 4096) X **K^T** (4096, 6) = $\frac{Q \times K^T}{\sqrt{4096}}$

	THE	CAT	IS	ON	A	CHAIR
THE	0.268	0.119	0.134	0.148	0.179	0.152
CAT	0.124	0.278	0.201	0.128	0.154	0.115
IS	0.147	0.132	0.262	0.097	0.218	0.145
ON	0.210	0.128	0.206	0.212	0.119	0.125
A	0.146	0.158	0.152	0.143	0.227	0.174
CHAIR	0.195	0.114	0.203	0.103	0.157	0.229

(6, 6)

* all values are random.

Let's apply a causal mask

After applying the causal mask we apply the softmax, which makes the remaining values on the row in such a way that the row sums up to 1.

Now, let's look at the **sliding window attention**.

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

Q

(6, 4096)

X

K^T

(4096, 6)

$$\frac{}{\sqrt{4096}} =$$

	THE	CAT	IS	ON	A	CHAIR
THE	0.268	−∞	−∞	−∞	−∞	−∞
CAT	0.124	0.278	−∞	−∞	−∞	−∞
IS	0.147	0.132	0.262	−∞	−∞	−∞
ON	0.210	0.128	0.206	0.212	−∞	−∞
A	0.146	0.158	0.152	0.143	0.227	−∞
CHAIR	0.195	0.114	0.203	0.103	0.157	0.229

(6, 6)

* all values are random.

Let's apply sliding window attention

The sliding window size is 3

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

Q

(6, 4096)

X

K^T

(4096, 6)

$$\frac{}{\sqrt{4096}}$$

=

	THE	CAT	IS	ON	A	CHAIR
THE	0.268	−∞	−∞	−∞	−∞	−∞
CAT	0.124	0.278	−∞	−∞	−∞	−∞
IS	0.147	0.132	0.262	−∞	−∞	−∞
ON	−∞	0.128	0.206	0.212	−∞	−∞
A	−∞	−∞	0.152	0.143	0.227	−∞
CHAIR	−∞	−∞	−∞	0.103	0.157	0.229

(6, 6)

* all values are random.

Sliding Window Attention: details

- Reduces the number of dot-products to perform, and thus, performance during training and inference.
- Sliding window attention may lead to degradation in the performance of the model, as some "interactions" between tokens will not be captured.
  The model mostly focuses on the **local context**, which depending on the size of the window, is enough for most cases.
  This makes sense if you think about a book: the words in a paragraph on chapter number 5 depend on the paragraphs in the same chapter but may be totally unrelated to the words used in chapter 1.

- Sliding window attention can still allow one token to watch tokens outside the window, using a reasoning similar to the **receptive field** in convolutional neural networks.

Receptive field in CNNs

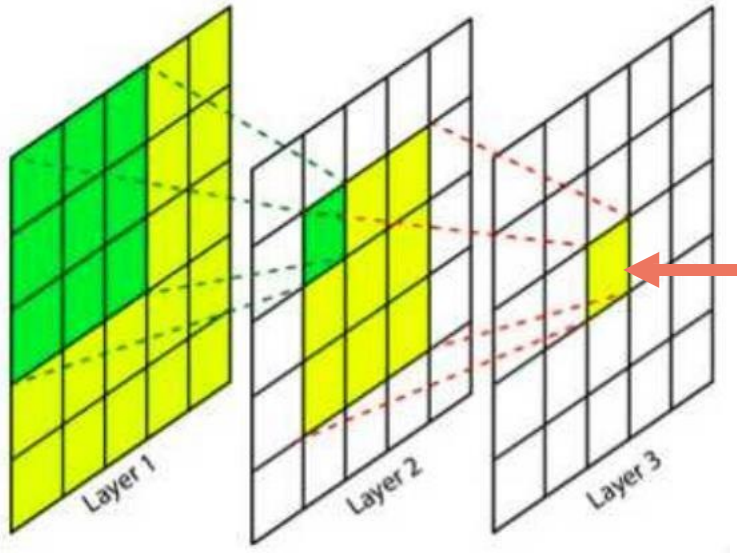


Image source: https://theaisummer.com/receptive-field/

This feature depends **directly** on 9 features of the previous layers, but **indirectly** on all the features of the initial layers, since each feature of the intermediate layer depends on 9 features of the previous layer. This means that a change in any of the features of the layer 1 will influence this features as well.

Sliding Window Attention: information flow (1)

After applying the softmax, all the –Infinity have become 0, while the other values in the row are changed in such a way that they sum up to one. The output of the softmax can be thought of as a probability distribution.

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

softmax $\left(\frac{Q \text{ (6, 4096)} \times K^T \text{ (4096, 6)}}{\sqrt{4096}}\right) =$

	THE	CAT	IS	ON	A	CHAIR
THE	1.0	0	0	0	0	0
CAT	0.461	0.538	0	0	0	0
IS	0.3219	0.317	0.361	0	0	0
ON	0	0.316	0.341	0.343	0	0
A	0	0	0.326	0.323	0.351	0
CHAIR	0	0	0	0.313	0.331	0.356

Sliding Window Attention: information flow (2)

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

	THE	CAT	IS	ON	A	CHAIR
THE	1.0	0	0	0	0	0
CAT	0.461	0.538	0	0	0	0
IS	0.3219	0.317	0.361	0	0	0
ON	0	0.316	0.341	0.343	0	0
A	0	0	0.326	0.323	0.351	0
CHAIR	0	0	0	0.313	0.331	0.356

(6, 6)

X

V

(6, 4096)

=

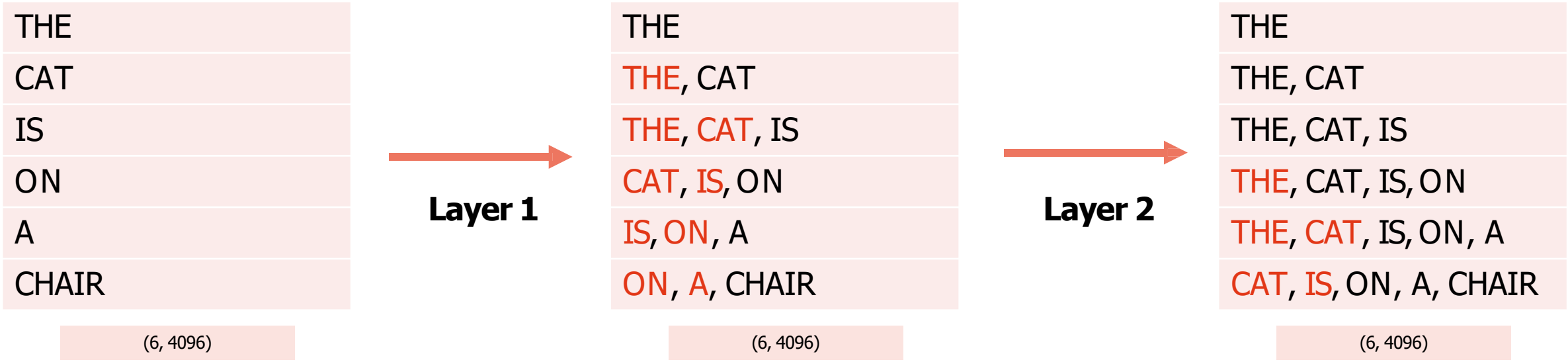
Output

(6, 4096)

The output of the Self-Attention is a matrix of the same shape as the input sequence, but where each token now captures information about other tokens according to the mask applied.

In our case, the last token of the output captures information about itself and the two preceding tokens.

Sliding Window Attention: information flow (3)



With a sliding window size $W = 3$, every layer adds information about $(W - 1) = 2$ tokens.
This means that after N layers, we will have an information flow in the order of $W \times N$.
You can test all the future configurations using the Python Notebook provided in the GitHub repository.

Sliding Window Attention: information flow (4)

	THE	THE/CAT	THE/CAT /IS	CAT/IS/ ON	IS/ON/A	ON/A/C HAIR
THE	1.0	0	0	0	0	0
THE/CA T	0.461	0.538	0	0	0	0
THE/CA T/IS	0.3219	0.317	0.361	0	0	0
CAT/IS/ ON	0	0.316	0.341	0.343	0	0
IS/ON/ A	0	0	0.326	0.323	0.351	0
ON/A/C HAIR	0	0	0	0.313	0.331	0.356

(6, 6)

X

v

(6, 4096)

=

Output

(6, 4096)

Sliding Window Attention: information flow (5)

As you can see, the information flow is very similar to the receptive field of a CNN.

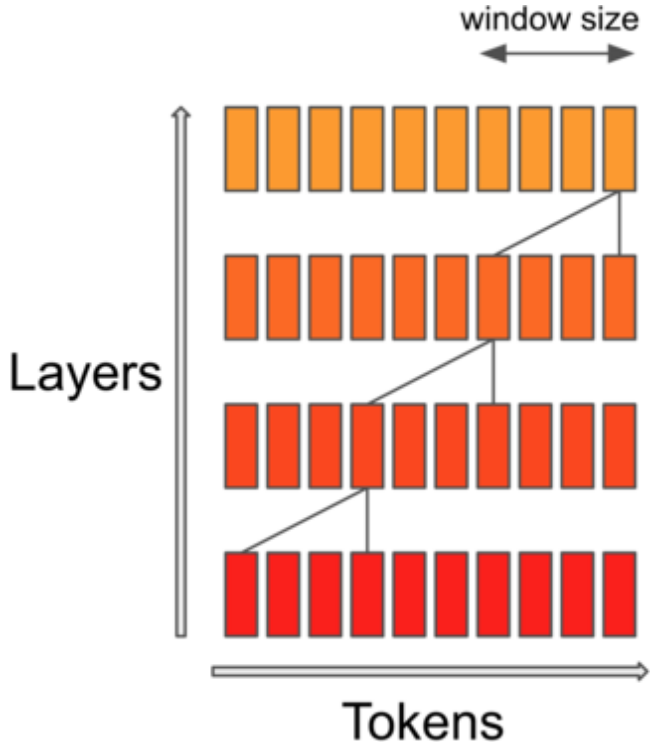


Image source: Mistral 7B paper

Topics

- Architectural differences between the vanilla Transformer and Mistral
- Sliding Window Attention
  - Review of self-attention
  - Receptive field
- KV-Cache
  - Motivation
  - How it works
  - Rolling Buffer Cache
  - Pre-fill and chunking
- Sparse Mixture of Experts
- Model Sharding
  - Pipeline Parallelism
- Understanding the Mistral model's code
  - Block attention in xformers

Next Token Prediction Task

- Imagine we want to train a model to write Dante Alighieri's Divine Comedy's 5th Canto from the Inferno.

Amor, ch'al cor gentil ratto s'apprende,
prese costui de la bella persona
che mi fu tolta; e 'l modo ancor m'offende.

Amor, ch'a nullo amato amar perdona,
mi prese del costui piacer sì forte,
che, come vedi, ancor non m'abbandona.

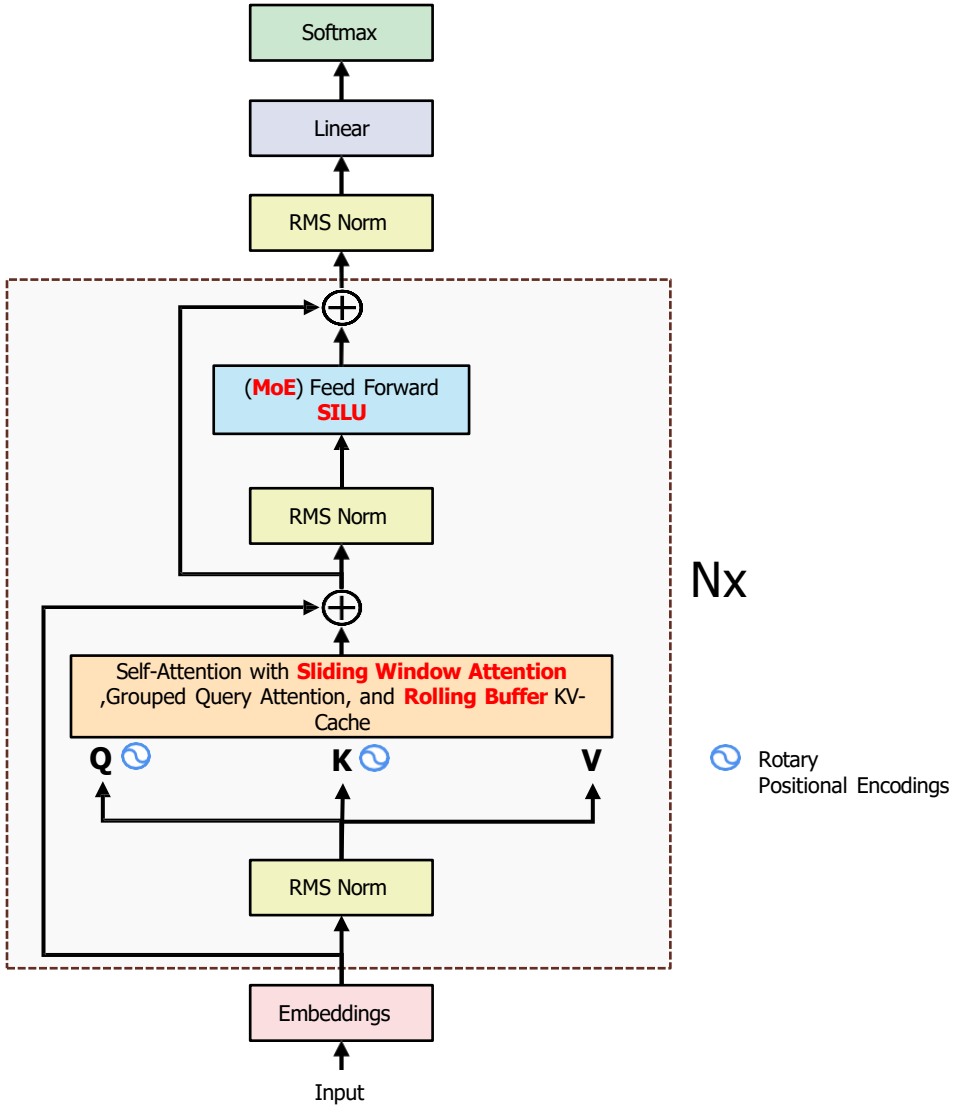
Amor condusse noi ad una morte.
Caina attende chi a vita ci spense.

Love, that can quickly seize the gentle heart,
took hold of him because of the fair body
taken from me—how that was done still wounds me.

Love, that releases no beloved from loving,
took hold of me so strongly through his beauty
that, as you see, it has not left me yet.

Love led the two of us unto one death.
Caina waits for him who took our life."

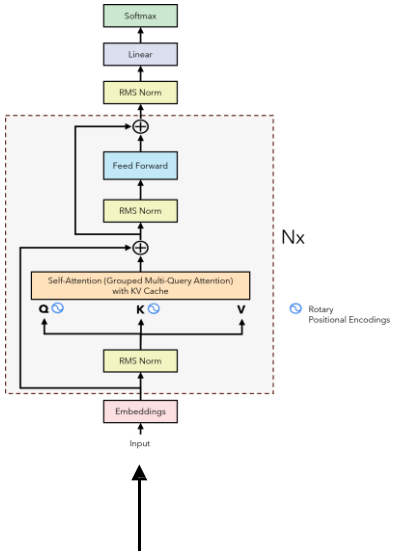
Source: https://digitaldante.columbia.edu/dante/divine-comedy/inferno/inferno-5/



Next Token Prediction Task

Target Love that can quickly seize the gentle heart **[EOS]**

Training

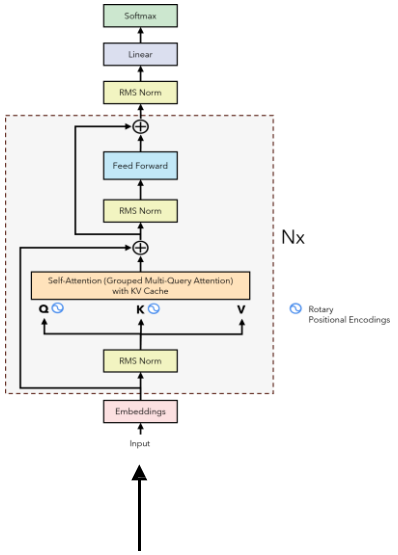


Input [SOS] Love that can quickly seize the gentle heart

Next Token Prediction Task: Inference

Output **Love**

Inference
T = 1

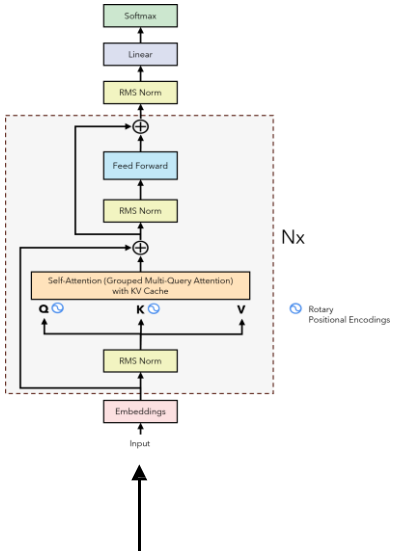


Input [SOS]

Next Token Prediction Task: Inference

Output Love **that**

Inference
T = 2

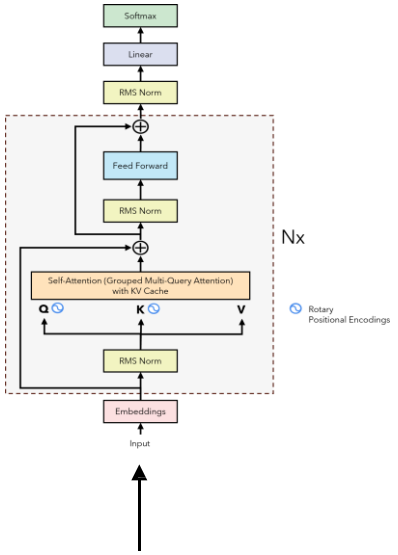


Input [SOS] Love

Next Token Prediction Task: Inference

Output Love that **can**

Inference
T = 3

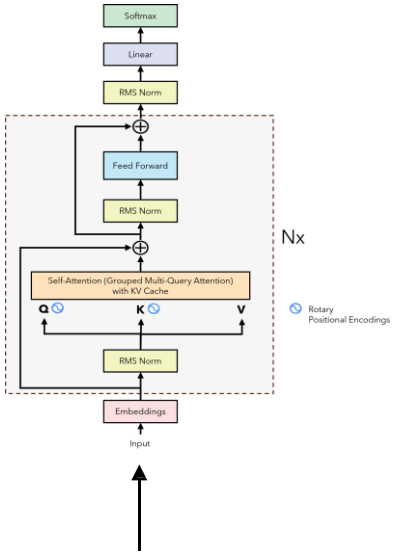


Input [SOS] Love that

Next Token Prediction Task: Inference

Output Love that can **quickly**

Inference
T = 4

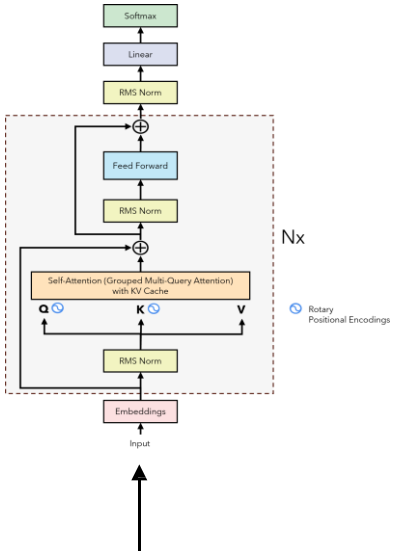


Input [SOS] Love that can

Next Token Prediction Task: Inference

Output Love that can quickly **seize**

Inference
T = 5

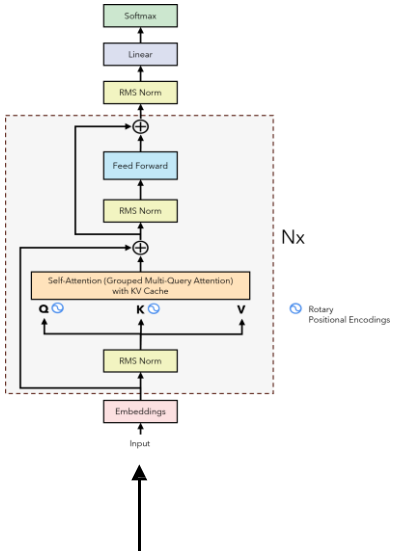


Input [SOS] Love that can quickly

Next Token Prediction Task: Inference

Output Love that can quickly seize **the**

Inference
T = 6

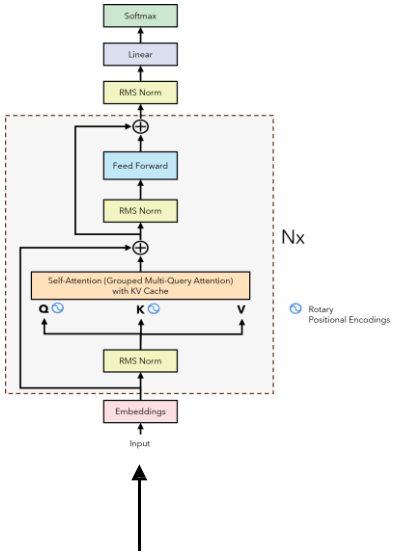


Input [SOS] Love that can quickly seize

Next Token Prediction Task: Inference

Output Love that can quickly seize the **gentle**

Inference
T = 7

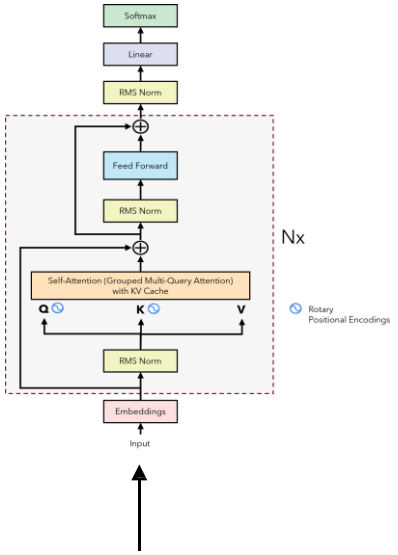


Input [SOS] Love that can quickly seize the

Next Token Prediction Task: Inference

Output Love that can quickly seize the gentle **heart**

Inference
T = 8

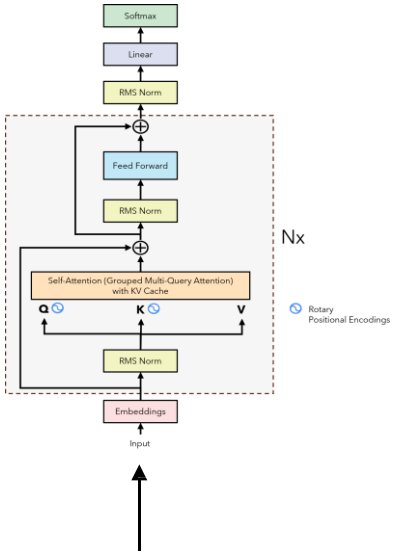


Input [SOS] Love that can quickly seize the gentle

Next Token Prediction Task: Inference

Output Love that can quickly seize the gentle heart [**EOS**]

Inference
T = 9

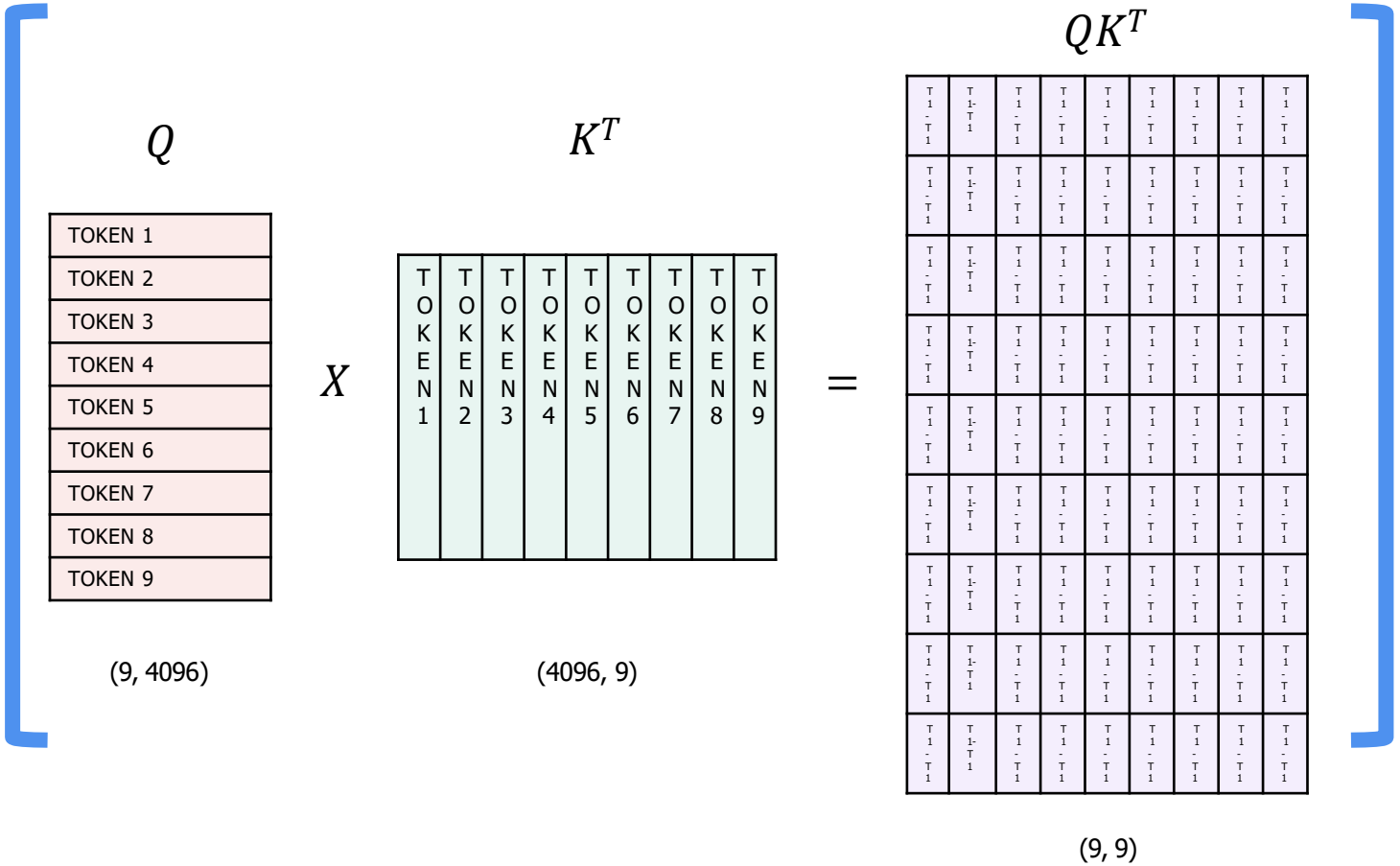


Input [SOS] Love that can quickly seize the gentle heart

Next Token Prediction Task: the motivation behind the KV cache

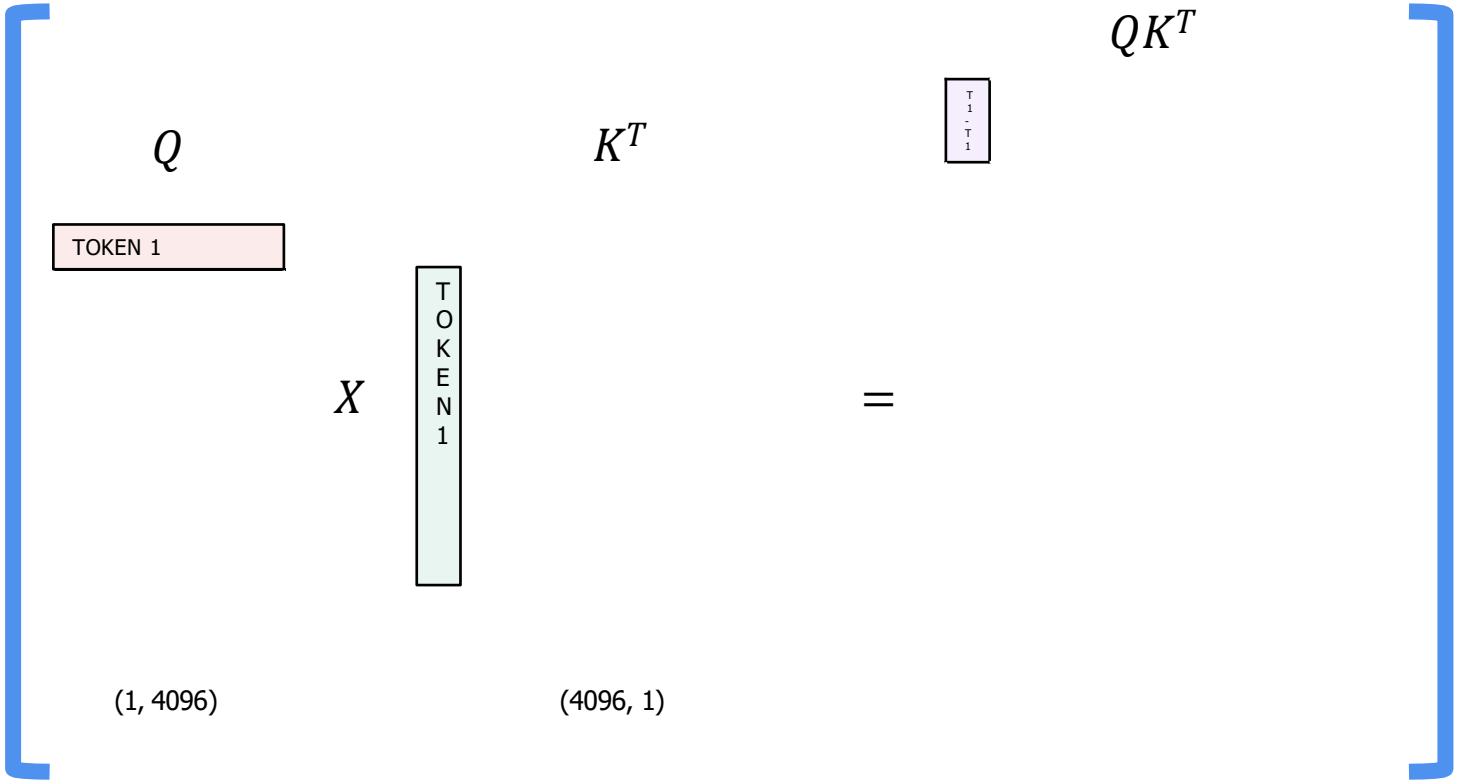
- At every step of the inference, we are only interested in the **last token** output by the model, because we already have the previous ones. However, the model needs access to all the previous tokens to decide on which token to output, since they constitute its context (or the "prompt").
- Is there a way to make the model do less computation on the token it has already seen **during inference**? YES! The solution is the **KV cache**!

Self-Attention during Next Token Prediction Task



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

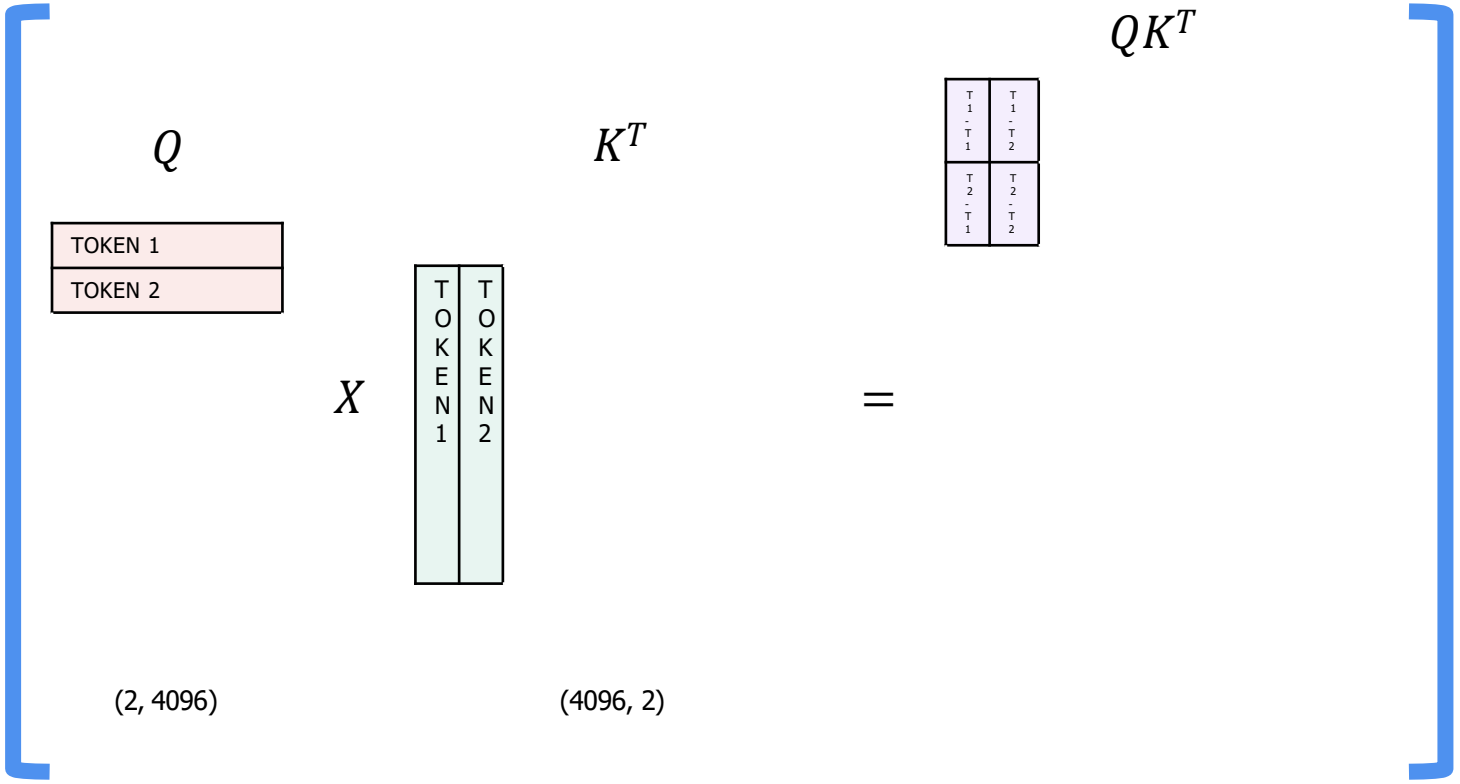
Self-Attention during Next Token Prediction Task

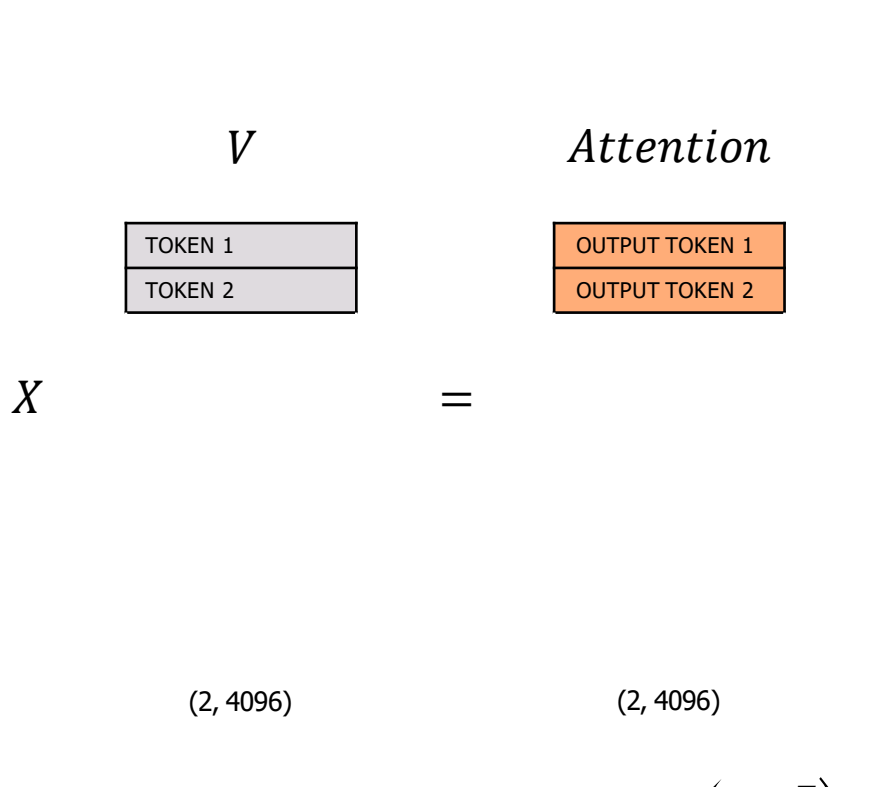


Inference
T = 1

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

Self-Attention during Next Token Prediction Task

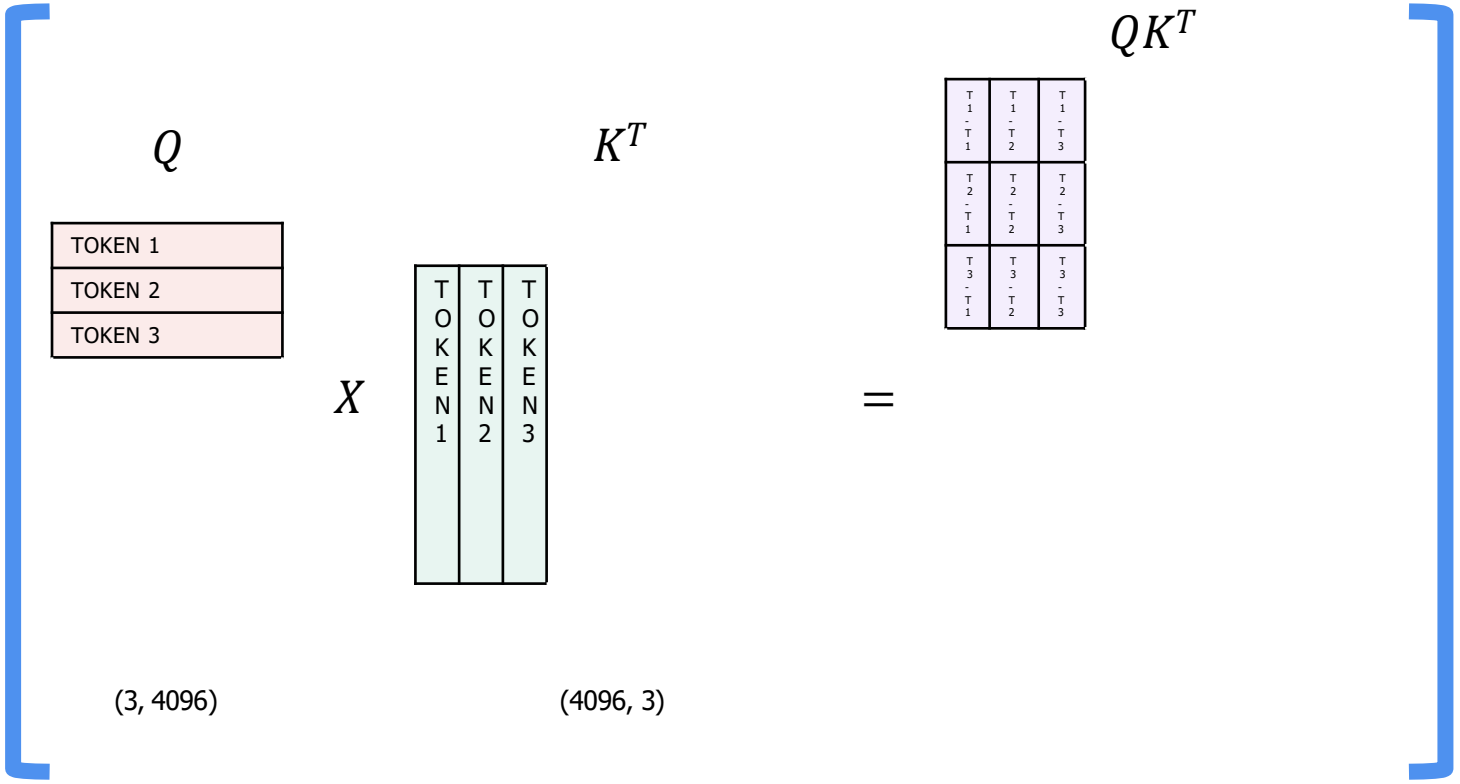




Inference
T = 2

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

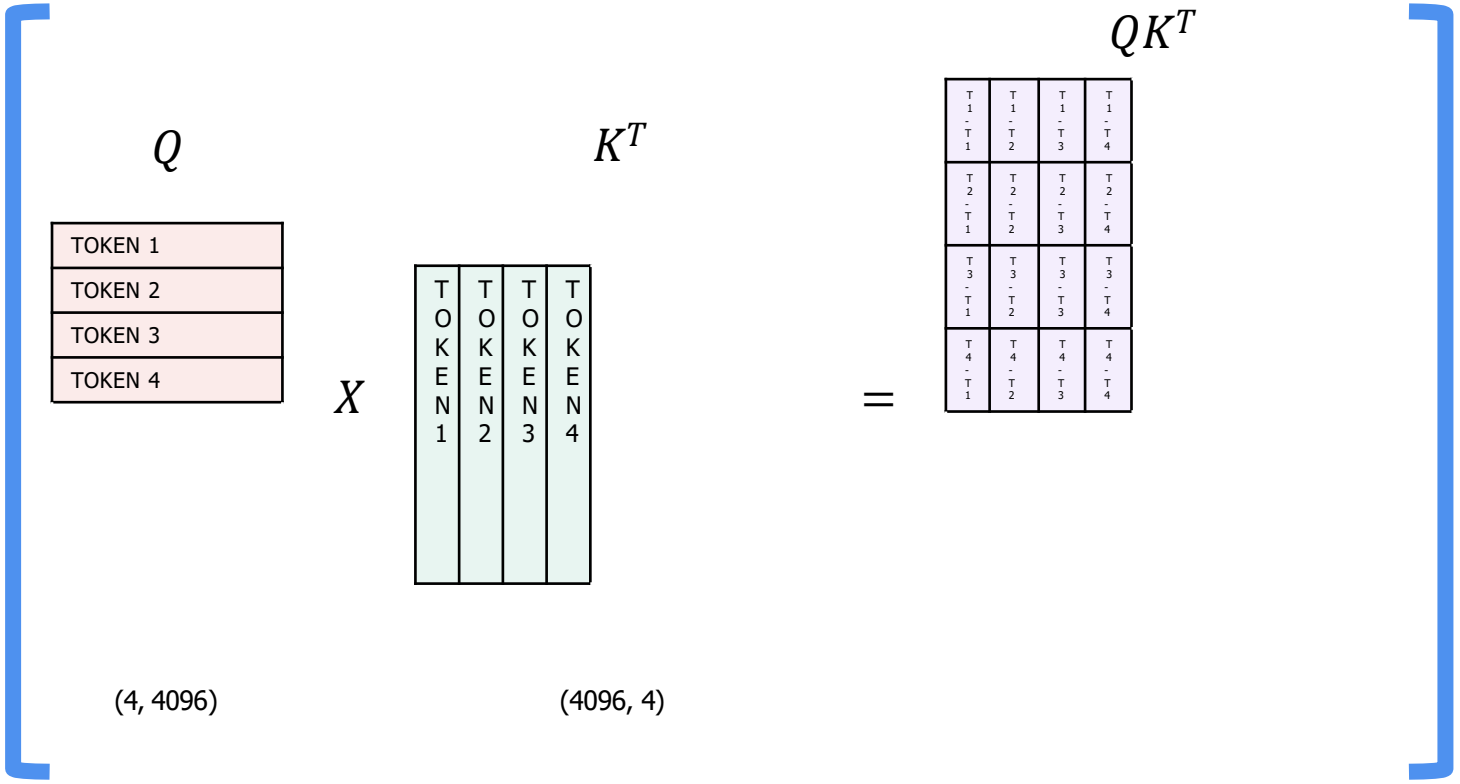
Self-Attention during Next Token Prediction Task



Inference
T = 3

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

Self-Attention during Next Token Prediction Task



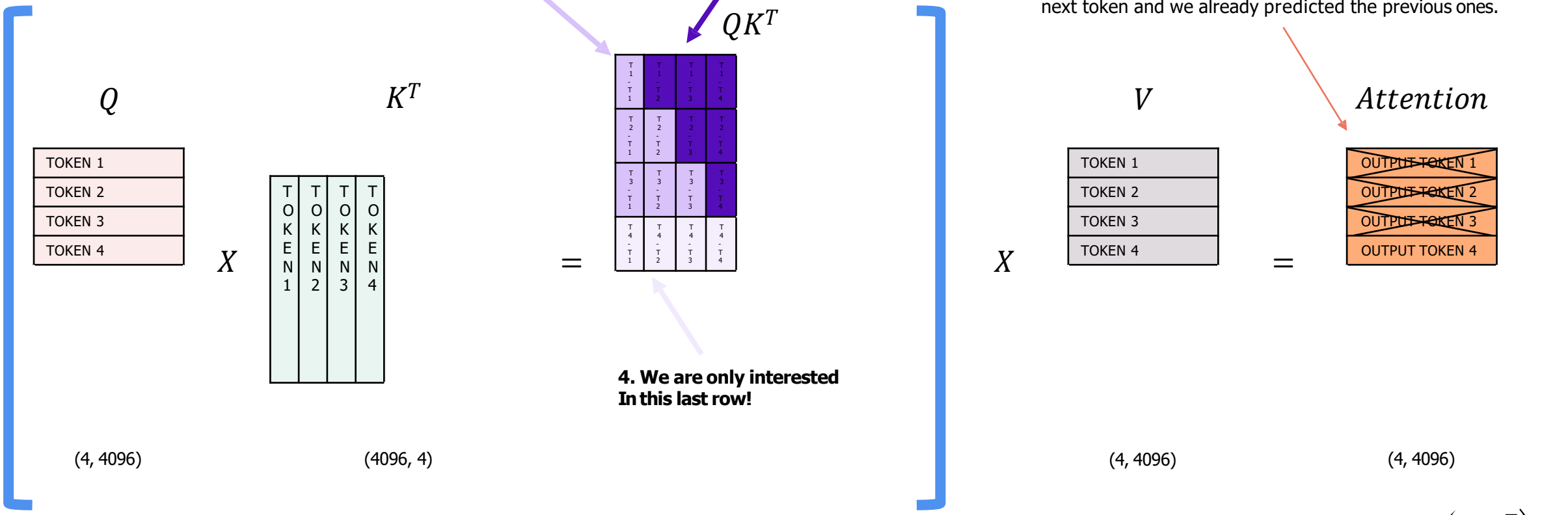
Inference
T = 4

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

1. We already computed these dot products In the previous steps. **Can we cache them**?

2. Since the model is causal, **we don't care about the attention of a token with its successors**, but only with the tokens before it.

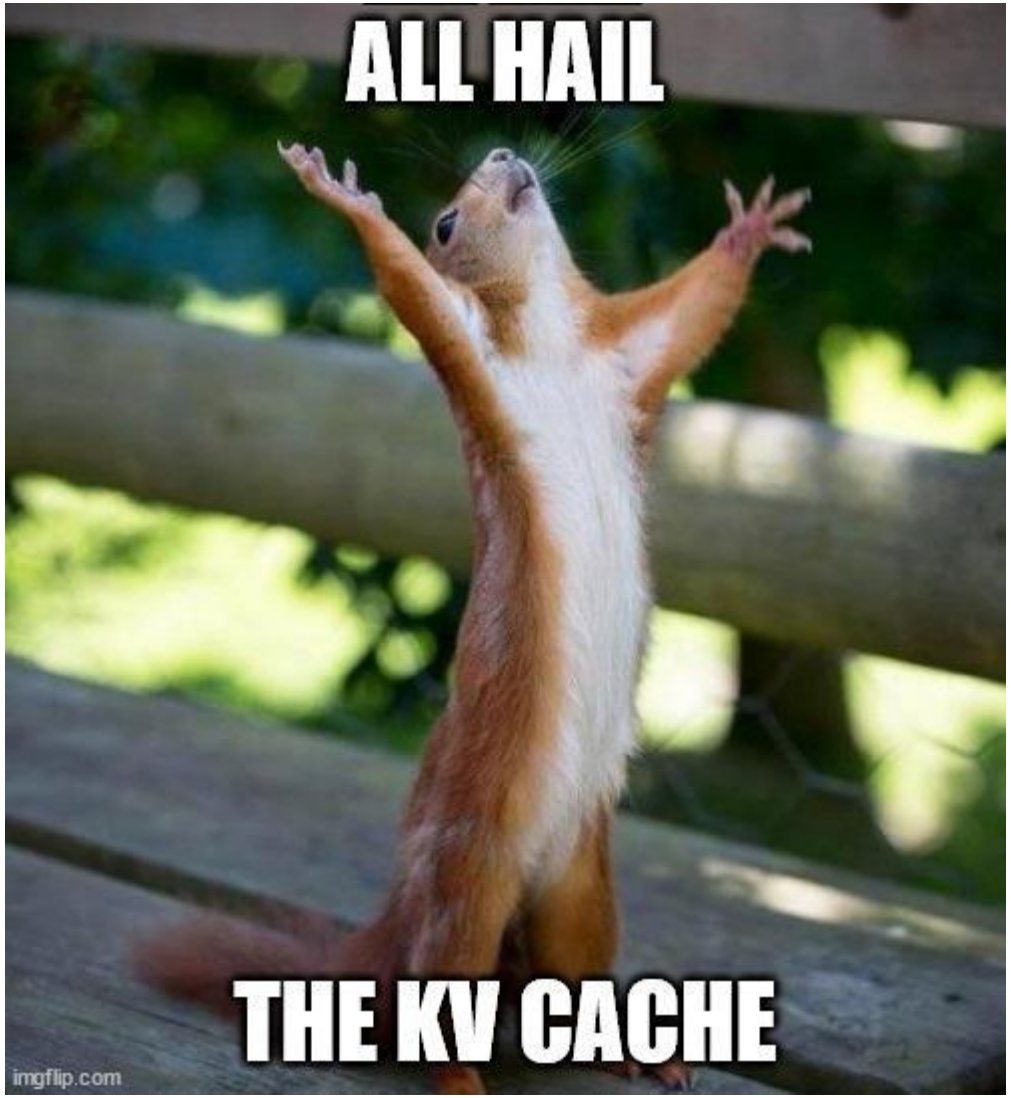
3. We don't care about these, as we want to predict the next token and we already predicted the previous ones.



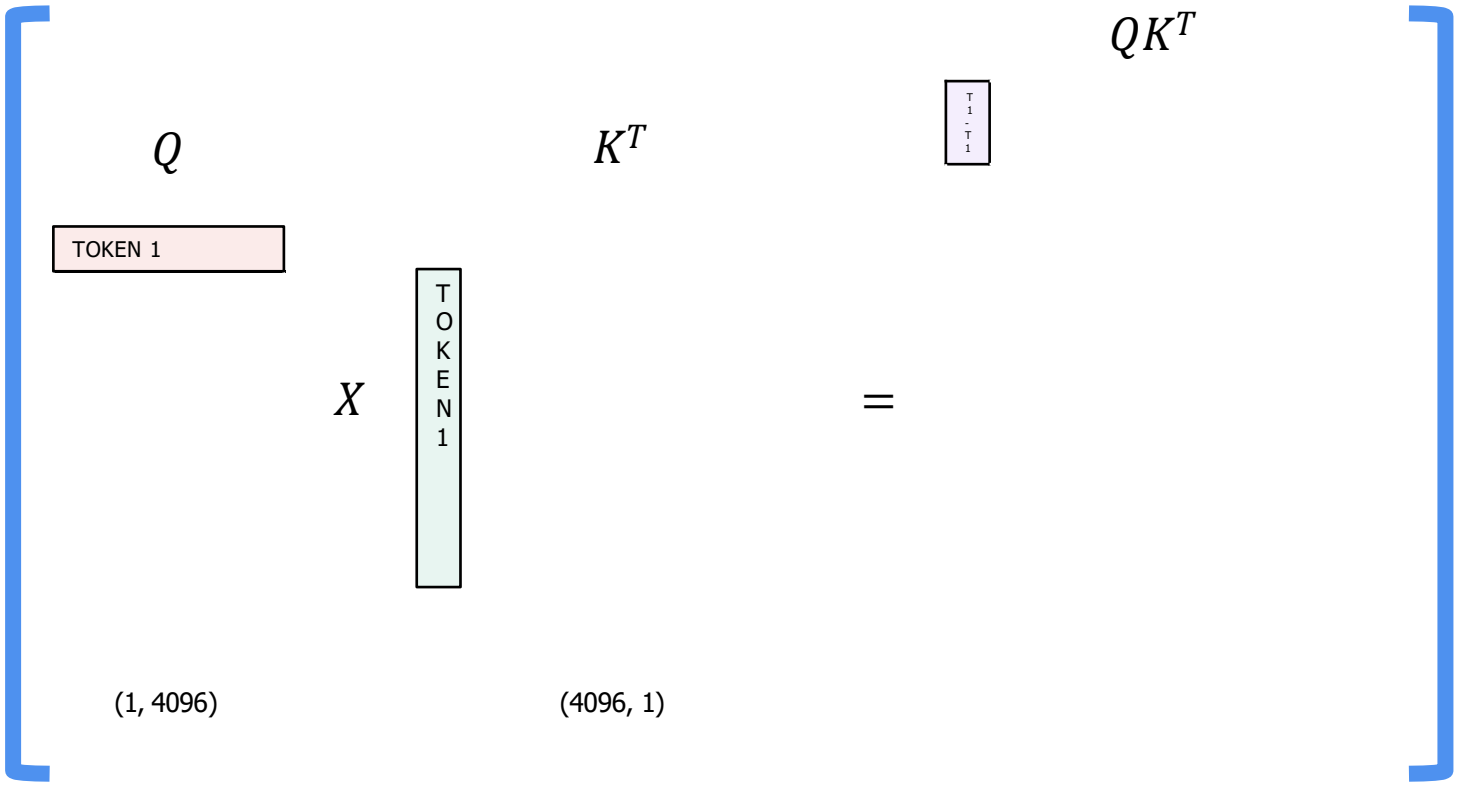
4. We are only interested In this last row!

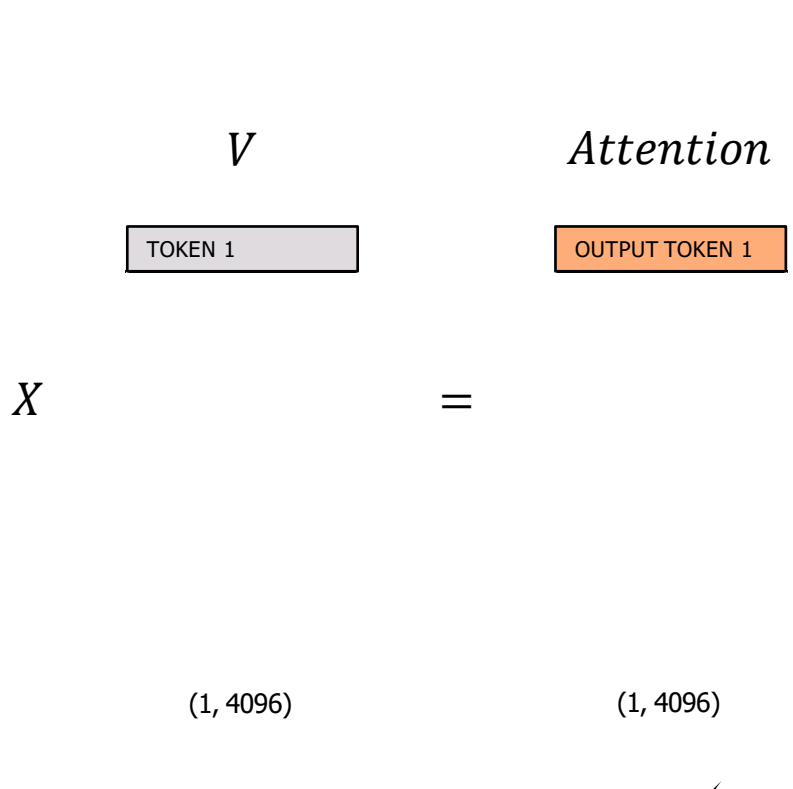
Inference
T = 4

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


ALL HAIL
THE KV CACHE
imgflip.com

Self-Attention with KV-Cache

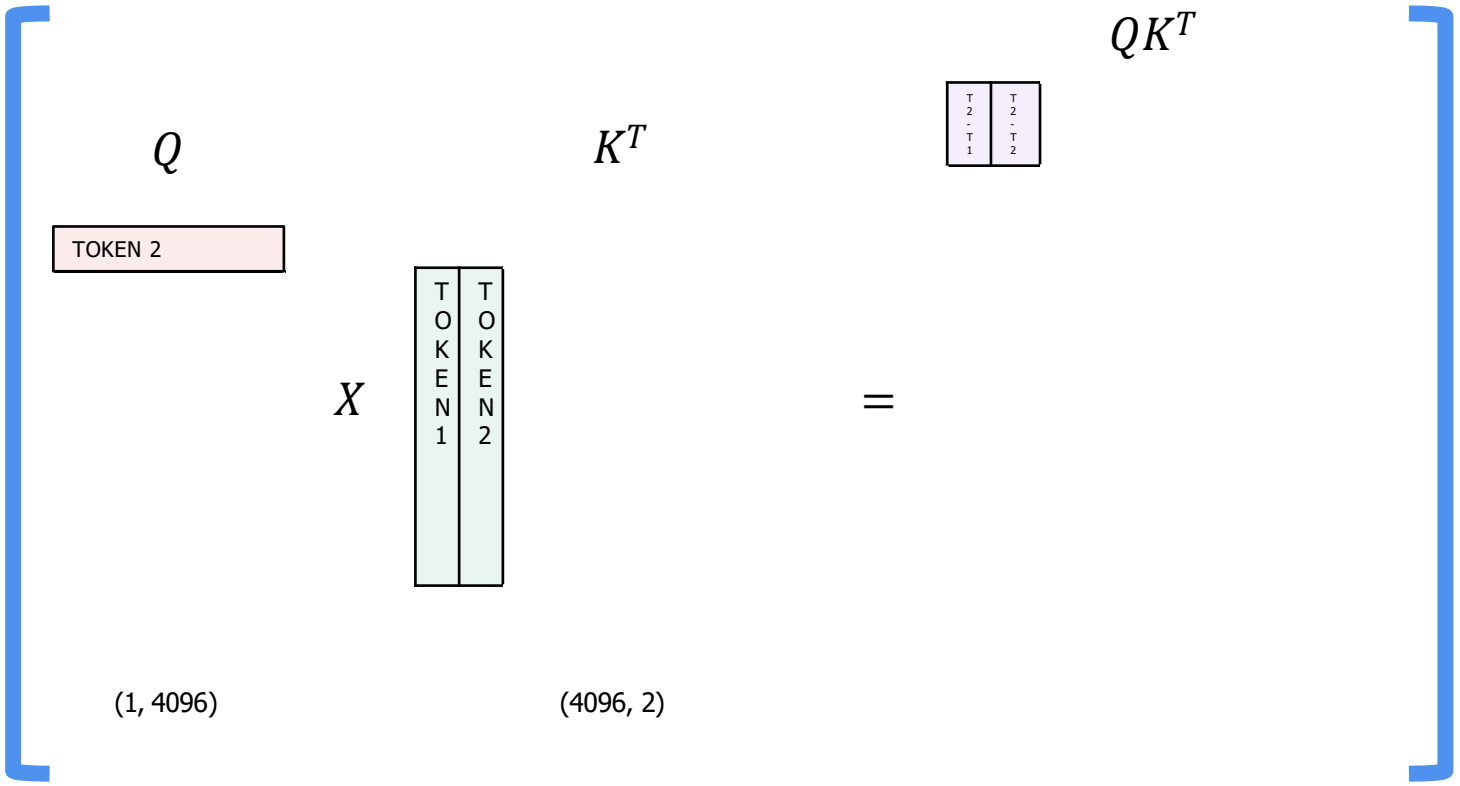


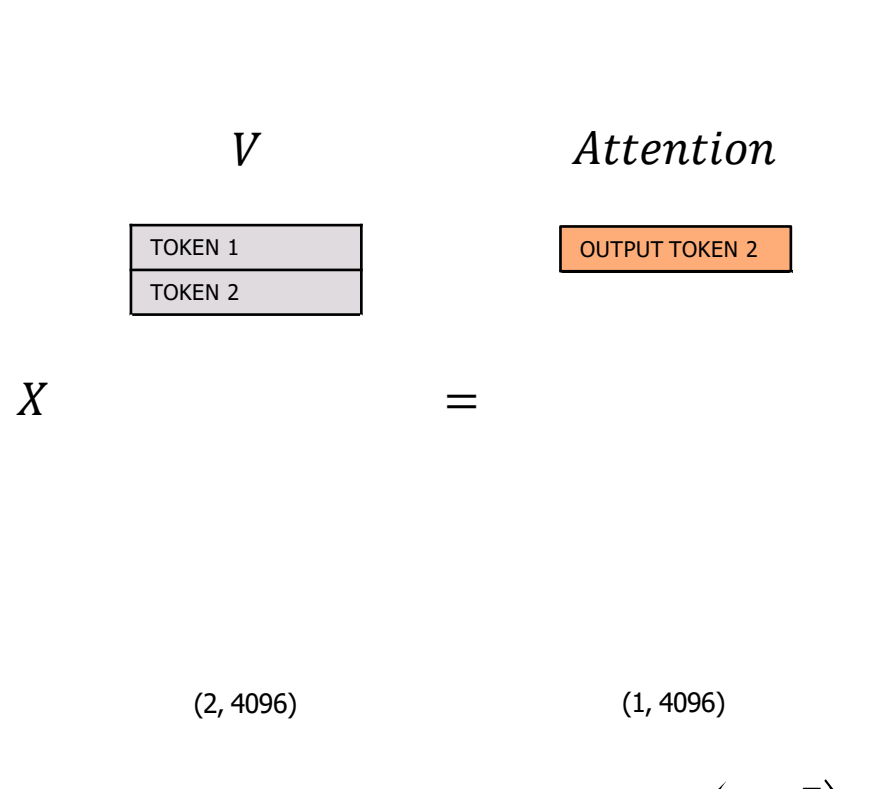


Inference
T = 1

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

Self-Attention with KV-Cache

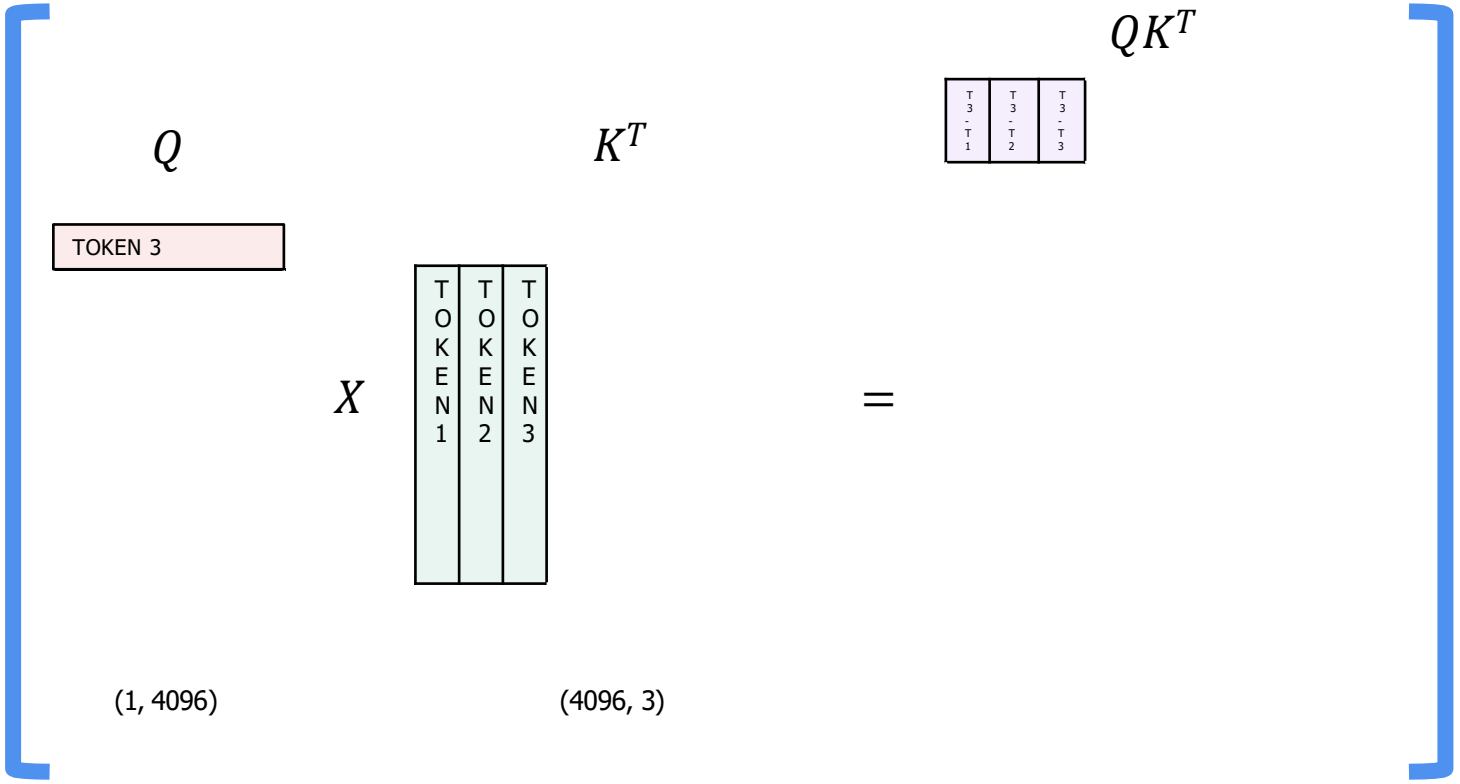


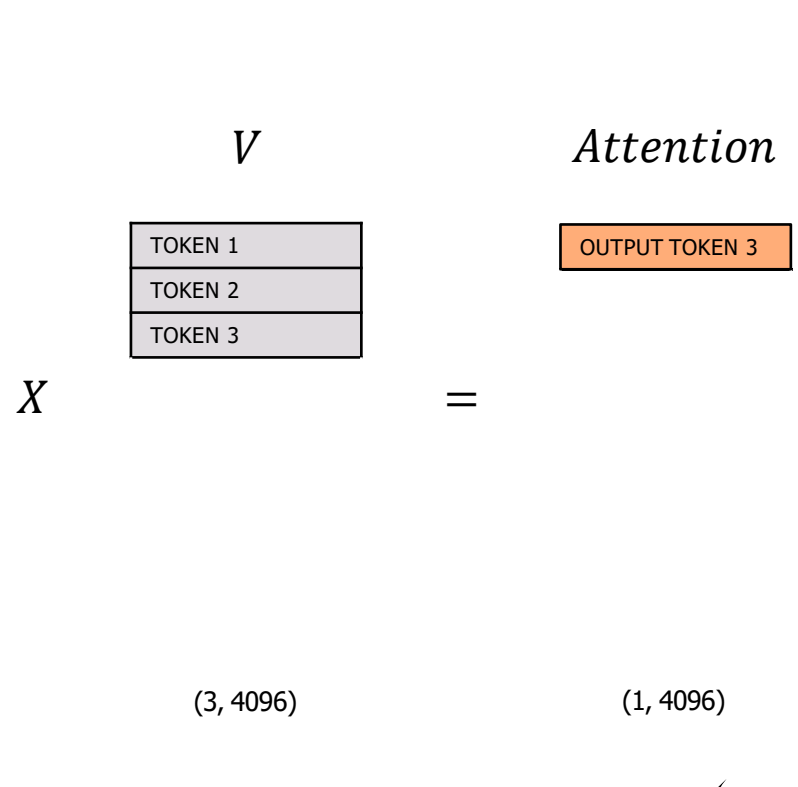


Inference
T = 2

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

Self-Attention with KV-Cache

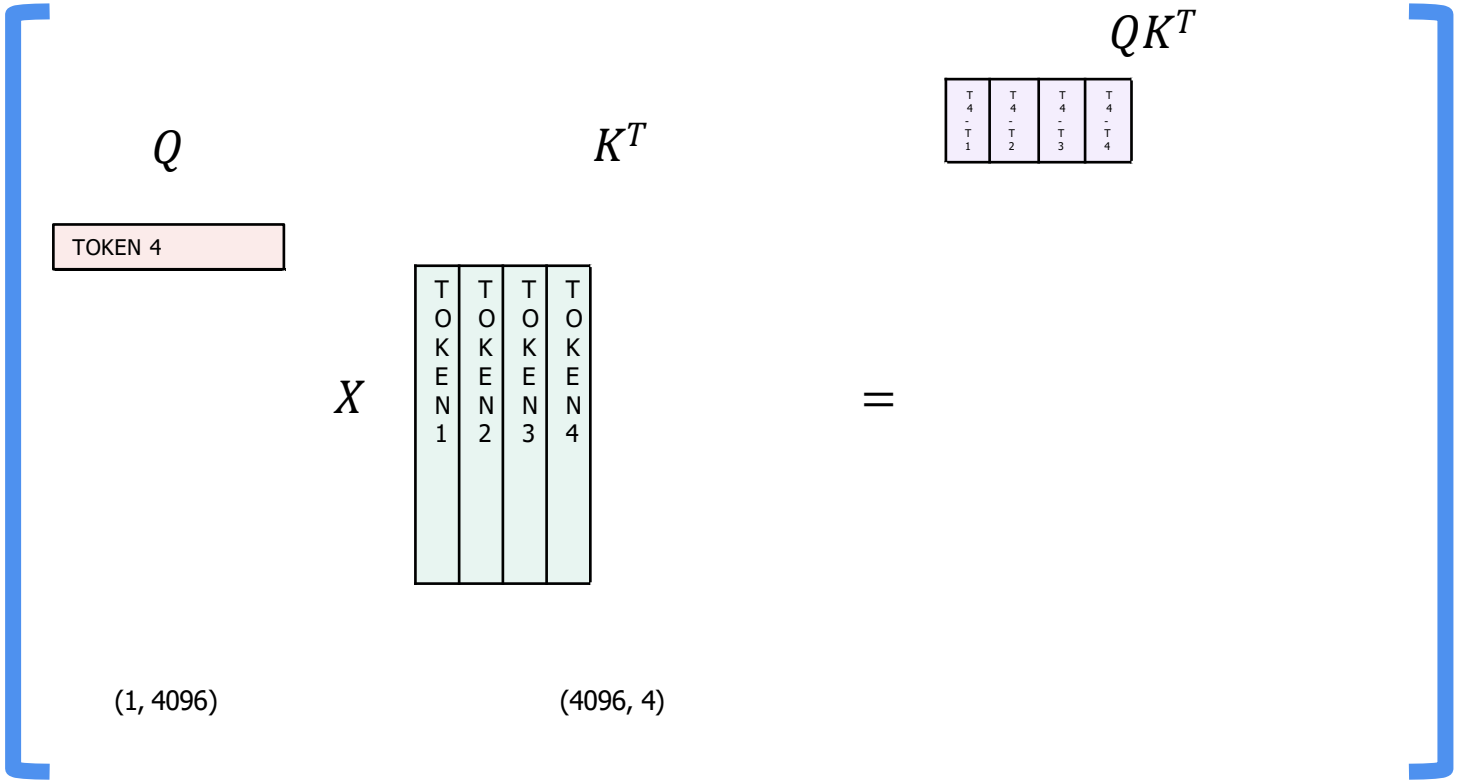


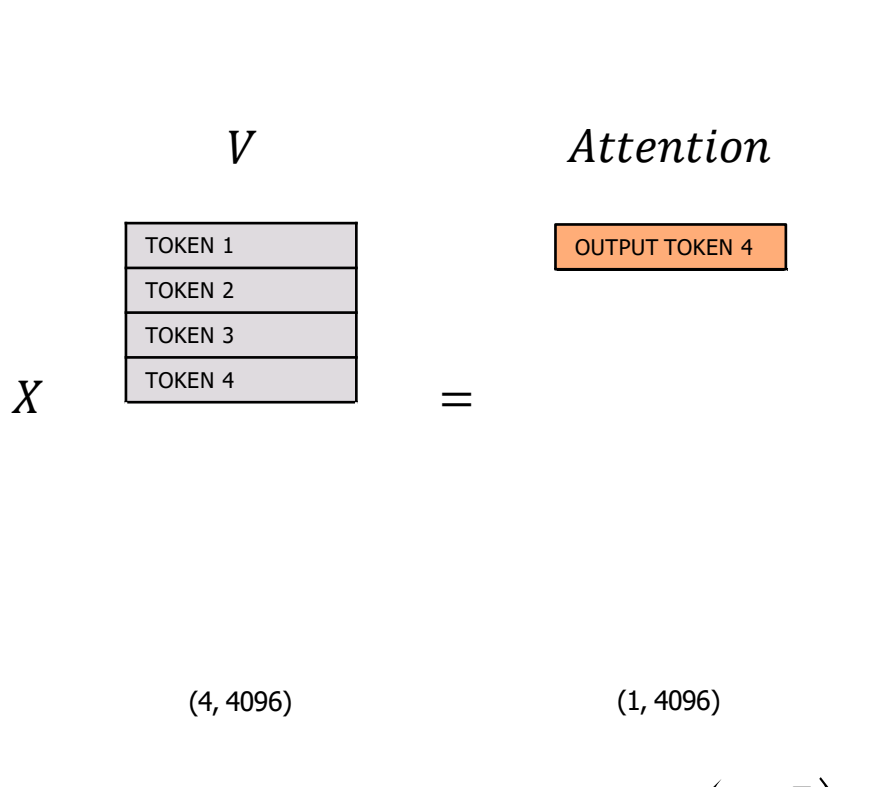


Inference
T = 3

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

Self-Attention with KV-Cache





Inference
T = 4

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

Rolling Buffer Cache

- Since we are using **Sliding Window Attention** (with size **W**), we don’t need to keep all the previous tokens in the KV-Cache, but we can limit it to the latest W tokens.

- When generating text, the model remembers previously computed hidden states instead of recomputing them for every new token.
- Instead of storing all past hidden states, it stores only the necessary ones (W) in a buffer and reuses them efficiently.
- The oldest hidden states are gradually replaced (hence “rolling” buffer).
- This keeps memory usage bounded, while still preserving past context.
- Avoids redundant calculations and speeds up generation.

How Is This Different from Traditional Caching?

•Traditional KV Cache: Stores key-value (KV) pairs for all past tokens.

- **Problem:** Memory grows **linearly** with sequence length.

•Rolling Buffer Cache: Keeps only the most useful states (size W), discarding old ones.

- **Benefit:** Memory remains **constant**, regardless of input length.

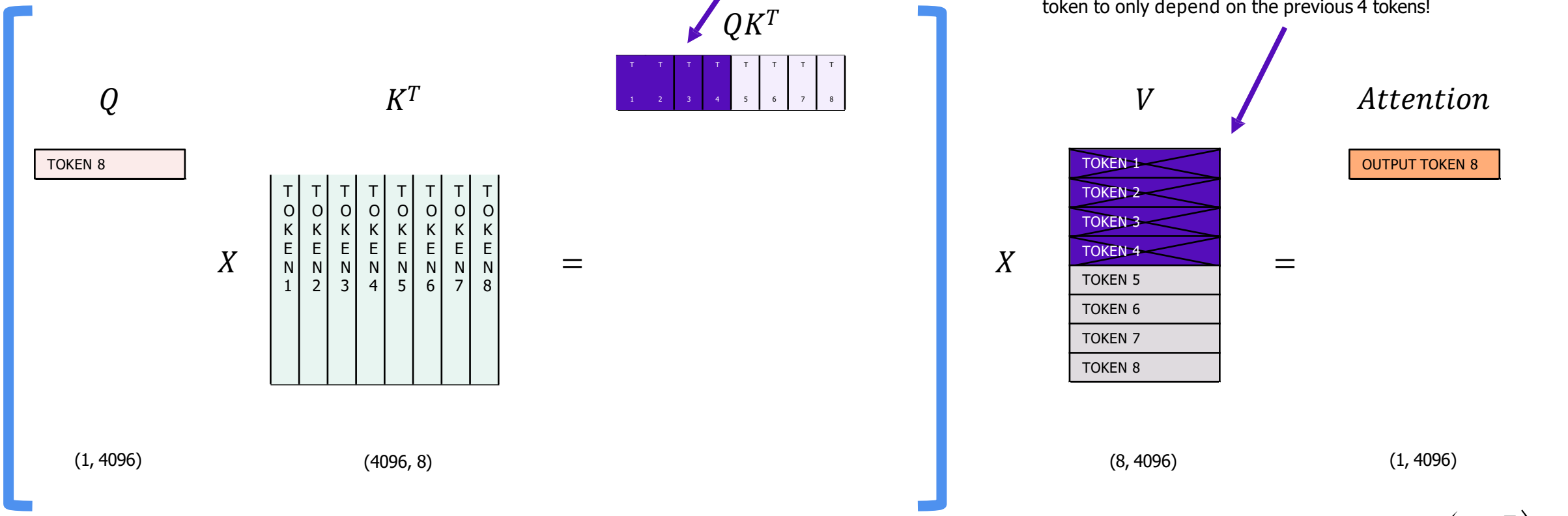
Rolling Buffer Cache

- Since we are using Sliding Window Attention (with size **W**), we don't need to keep all the previous tokens in the KV-Cache, but we can limit it to the latest **W** tokens.

The motivation

Since our sliding window size is 4, we only want the dot-product of the current token with the previous 4 (including the token itself)

We don't care about these either, as we want the output token to only depend on the previous 4 tokens!



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

Inference
T = 4

Rolling Buffer Cache: how it works

We keep track of write pointer that tells us where we added the last token in the rolling buffer cache.

Let's add the sentence "**The cat is on a chair**"

Rolling Buffer Cache: how it works

We add a new token and move the pointer forward

Let's add the sentence "**The cat is on a chair**"

Rolling Buffer Cache: how it works

We add a new token and move the pointer forward

Let's add the sentence "**The cat is on a chair**"

Rolling Buffer Cache: how it works

We add a new token and move the pointer forward

Let's add the sentence "**The cat is on a chair**"

Rolling Buffer Cache: how it works

We add a new token and move the pointer forward

Let's add the sentence "**The cat is on a chair**"

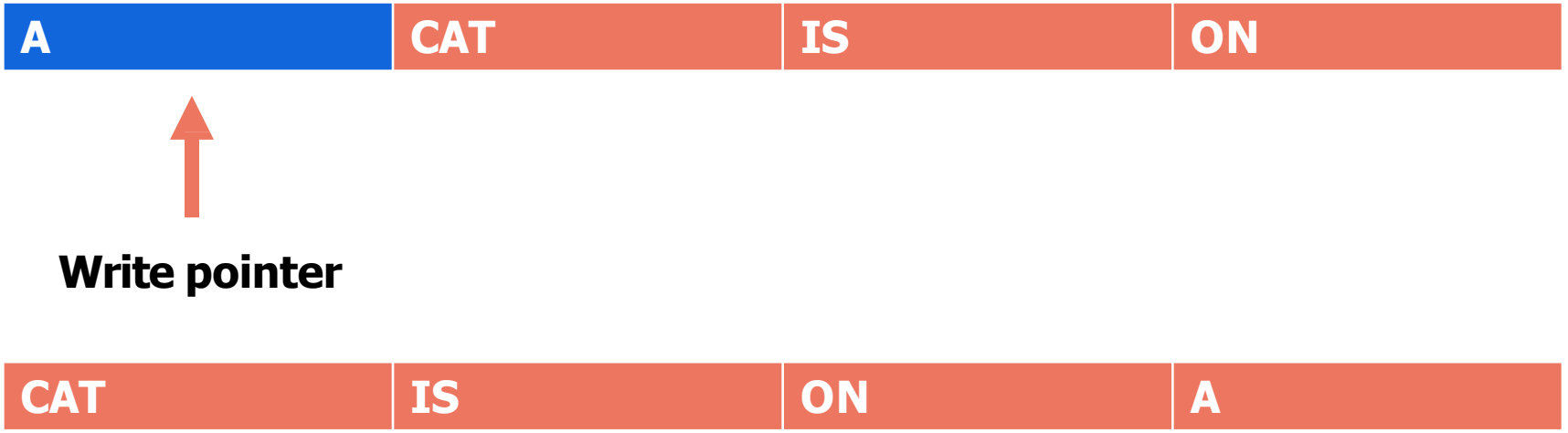
Rolling Buffer Cache: how it works

We add a new token and move the pointer forward

Let's add the sentence "**The cat is on a chair**"

Rolling Buffer Cache: unrolling

Imagine we want to “unroll” the cache because we want to calculate the attention of the incoming token. It’s very easy! We just need to use the write pointer to understand how to order the items: we first take all the items AFTER the write pointer, and then all the items from the 0th index to the position of the write pointer.




```python
def unrotate(cache: torch.Tensor, seqlen: int) -> torch.Tensor:
    assert cache.ndim == 3  # (W, H, D)
    position = seqlen % cache.shape[0]
    if seqlen < cache.shape[0]:
        return cache[:seqlen]
    elif position == 0:
        return cache
    else:
        return torch.cat([cache[position:], cache[:position]], dim=0)
```

Since the cache is not full yet, ignore unfilled items.

Since the cache is not full yet, ignore unfilled items.

Rotate the cache around the write pointer

Pre-fill and chunking

- When generating text using a Language Model, we use a prompt and then generate tokens one by one using the previous tokens. When dealing with a KV-Cache, we first need to add all the prompt tokens to the KV-Cache so that we can then exploit it to generate the next tokens.
- Since the prompt is known in advance (we don't need to generate it), we can prefill the KV-Cache using the tokens of the prompt. But what if the prompt is very big?
- We can either add one token at a time, but this can be time-consuming, otherwise we can add all the tokens of the prompt at once, but in that case the attention matrix (which is $N \times N$) may be very big and not fit in the memory.
- The solution is to use pre-filling and chunking. Basically, we divide the prompt into chunks of a fixed size set to **W** (except the last one), where **W** is the size of the sliding window of the attention.
- Imagine we have a large prompt, with a sliding window size of $W = 4$.
- For simplicity, let's pretend that each word is a token.
- **Prompt**: "Can you tell me who is the richest man in history"

Pre-fill and chunking: first chunk

Prompt: "Can you tell me who is the richest man in history"

The KV-Cache

The attention mask

	CAN	YOU	TELL	ME
CAN	0.268	−∞	−∞	−∞
YOU	0.124	0.278	−∞	−∞
TELL	0.147	0.132	0.262	−∞
ME	0.132	0.128	0.206	0.212

At every step, we calculate the attention using the tokens of the KV-Cache + the tokens of the current chunk as **Keys** and **Values**, while only the tokens of the incoming chunk as **Query**.
During the first step of pre-fill, the KV-Cache is initially empty.

After calculating the attention, we add the tokens of the current chunk to the KV-Cache.
This is different from token generation in which we first add the previously-generated token to the KV-Cache and then calculate the attention**.**

Pre-fill and chunking: second chunk

Prompt: "Can you tell me who is the richest man in history"

The KV-Cache

CAN	YOU	TELL	ME

The attention mask

	CAN	YOU	TELL	ME	WHO	IS	THE	RICHEST
WHO	−∞	0.132	0.262	0.132	0.951	−∞	−∞	−∞
IS	−∞	−∞	0.956	0.874	0.148	0.253	−∞	−∞
THE	−∞	−∞	−∞	0.132	0.262	0.259	0.456	−∞
RICHEST	−∞	−∞	−∞	−∞	0.132	0.687	0.159	0.357

You may have noticed that the size of the attention mask is bigger than the size of the KV-Cache.

This is done on purpose, otherwise the newly added tokens will not have their dot products calculated with items that were previously in the cache (for example the word "*Who*" will not have its attention calculated with the previous tokens). **This mechanism is only used during pre-fill of the prompt.**

Why do we do like this? Because the KV-Cache has a fixed size, but at the same time we need all these attentions computed.

Pre-fill and chunking: last chunk

Prompt: "Can you tell me who is the richest man in history"

The KV-Cache

WHO	IS	THE	RICHEST

The attention mask

	WHO	IS	THE	RICHEST	MAN	IN	HISTORY
MAN	−∞	0.132	0.262	0.132	0.951	−∞	−∞
IN	−∞	−∞	0.956	0.874	0.148	0.253	−∞
HISTORY	−∞	−∞	−∞	0.132	0.262	0.259	0.456

The last chunk may be smaller, that's why we have less rows.

When moving to the **third chunk**, the **first chunk is discarded from memory**, but its influence remains in the KV cache.
The third chunk will **only attend to the KV cache (past stored tokens) and itself**, **not the first chunk directly**.
However, because attention is cumulative, the second chunk **already encoded information from the first chunk** into the KV cache.

Pre-fill and chunking: code reference

Only during prefill the attention is calculated using an attention mask that is bigger than the KV-Cache.

As you can see, during the prefill of the first chunk and the subsequent chunks, we use the size of the KV-Cache and the number of tokens in the current chunk to generate the attention mask.

During pre-fill, the attention mask is calculated using the KV-Cache + the tokens in the current chunk, **so the size of the attention mask can be bigger than that of the KV-Cache (W)**.

During generation, we first add the previous token to the KV-Cache and then use the contents of the KV-Cache to generate the attention mask. **During generation, the size of the attention mask is the size of the KV-Cache (W).**

First chunk (KV-Cache is empty)

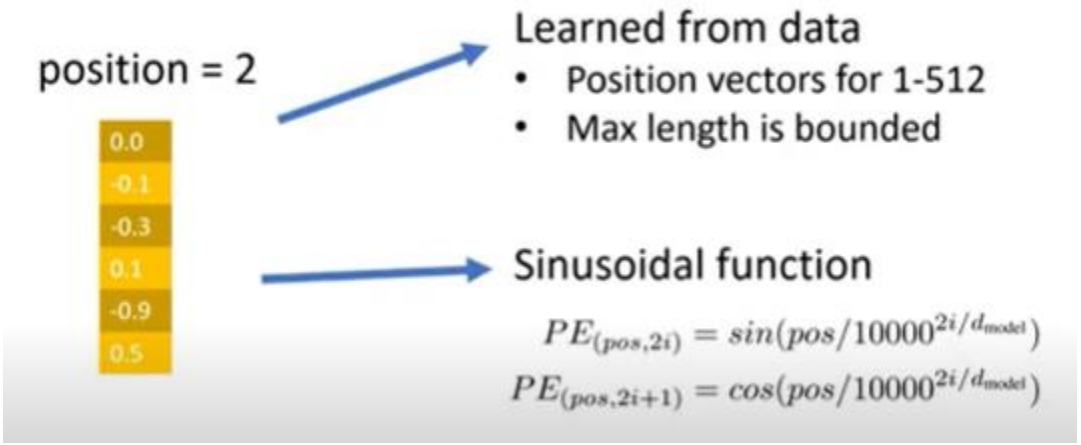
Subsequent chunks (KV-Cache is not empty)

Token generation

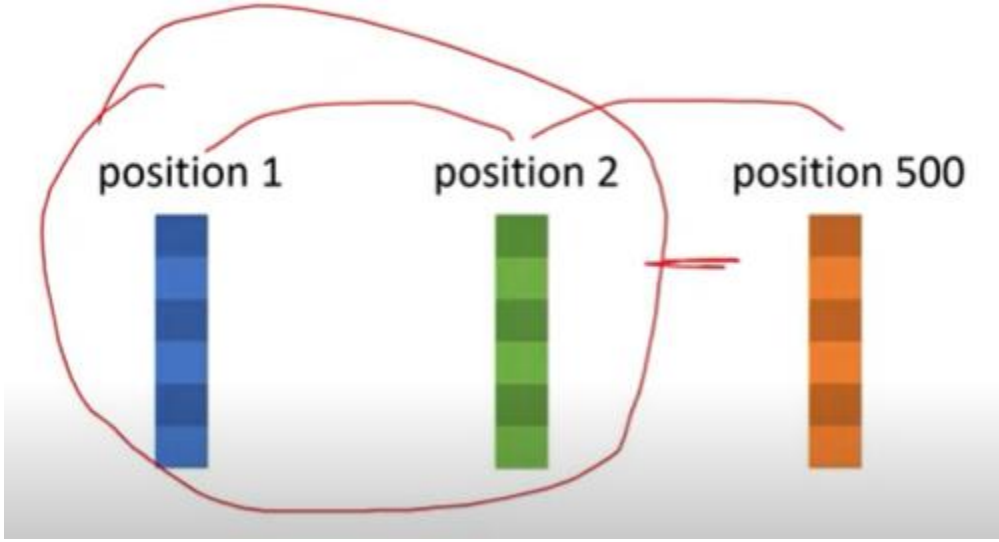
```
if first_prefill:
    assert all([pos == 0 for pos in seqpos]), (seqpos)
    mask = BlockDiagonalCausalMask.from_seqlens(seqlens).make_local_attention(self.sliding_window)
elif subsequent_prefill:
    mask = BlockDiagonalMask.from_seqlens(
        q_seqlen=seqlens,
        kv_seqlen=[s + cached_s.clamp(max=self.sliding_window).item() for (s, cached_s) in zip(seqlens, self.kv_seqlens)]
    ).make_local_attention_from_bottomright(self.sliding_window)
else:
    mask = BlockDiagonalCausalWithOffsetPaddedKeysMask.from_seqlens(
        q_seqlen=seqlens,
        kv_padding=self.sliding_window,
        kv_seqlen=(self.kv_seqlens + cached_elements).clamp(max=self.sliding_window).tolist()
    )
```

Challenges in Absolute PE

1.Limited Sequence Length: If a model learns positional vectors up to a certain point, it cannot inherently represent positions beyond that limit.
2.Independence of Positional Embeddings: Each positional embedding is independent of others. This means that in the model's view, the difference between positions 1 and 2 is the same as between positions 2 and 500.
However, intuitively, positions 1 and 2 should be more closely related than position 500, which is significantly farther away. This lack of relative positioning can hinder the model's ability to understand the nuances of language structure.



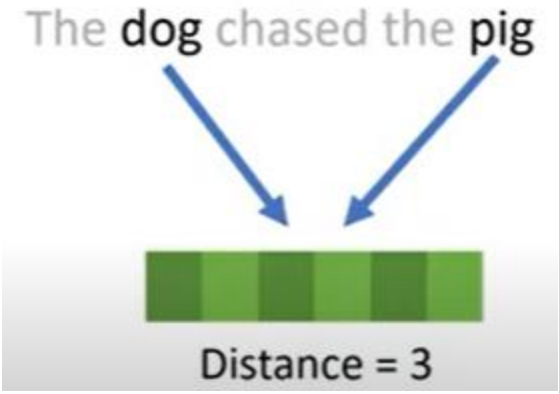
Limited Sequence Length



Independence of Positional Embeddings

Relative PE

- **R**elative positional embeddings concentrate on the distances between pairs of tokens.
- This method doesn’t add a position vector to the word vector directly. Instead, it alters the attention mechanism to incorporate relative positional information.
- **A** floating-point number, to represent each possible positional offset. For example, a bias B1 might represent the relative distance between any two tokens that are one position apart, regardless of their absolute positions in the sentence.

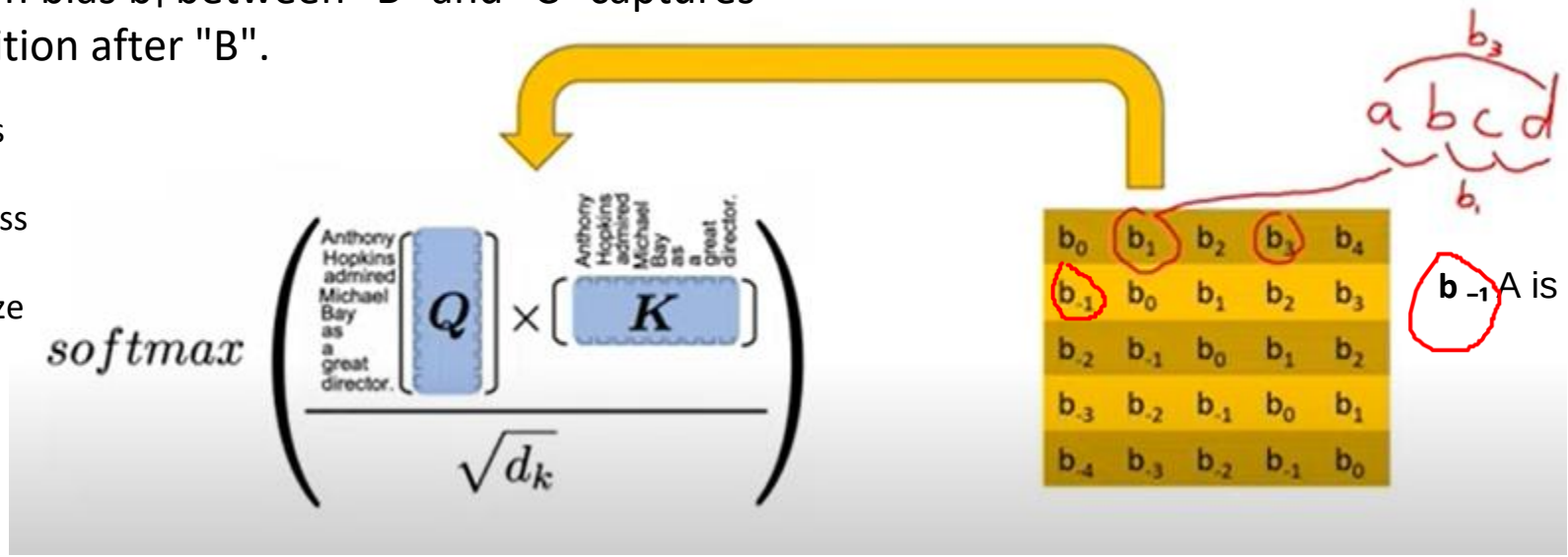


For example, if we have a sentence like: A B C D E

- The relative position bias b_{-1} between "C" and "B" captures that "B" is one position before "C".
- The relative position bias b_1 between "B" and "C" captures that "C" is one position after "B".

A bias B1 might represent the relative distance between any two tokens that are one position apart, regardless of their absolute positions in the sentence.

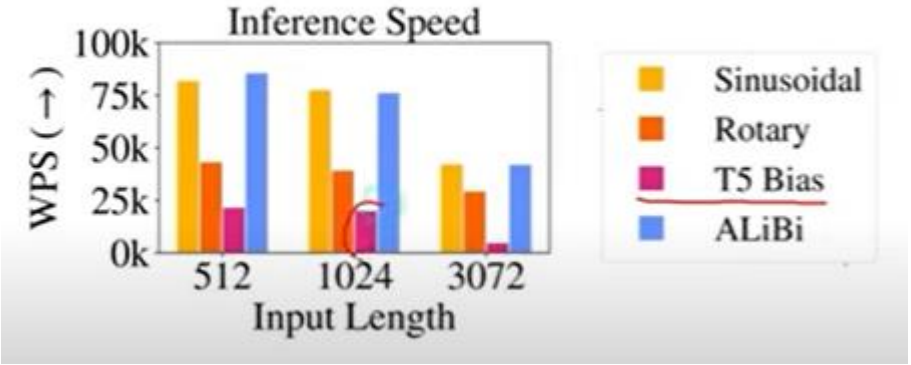
This means that the same bias value is applied consistently across the sequence, regardless of the absolute position.
This helps the model generalize to different lengths and placements of words while maintaining relative dependencies.



This **matrix** of **relative position biases** is **added** to the **product** of the query and key matrices in the self-attention layer. This ensures that tokens at the same relative distance are always represented by the same bias, regardless of their position in the sequence.

Challenges in Relative PE

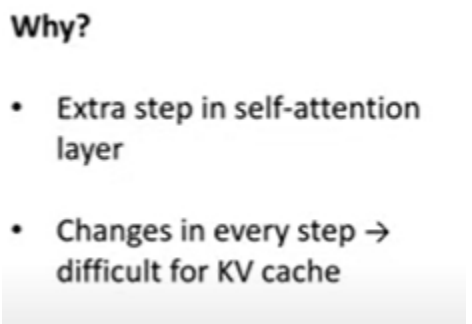
Performance Issues: Benchmarks comparing T5's relative embeddings with other types have shown that they can be slower, particularly for longer sequences.
This is primarily due to the additional computational step in the self-attention layer, where the positional matrix is added to the query-key matrix.



Challenges in Relative PE

- **Complexity in Key-Value Cache Usage:** As each additional token alters the embedding for every other token, this complicates the effective use of key-value caches in Transformers.
- For those unfamiliar, key-value caches are crucial in enhancing efficiency and speed in Transformer models.

As more tokens are added in the sequence, u have to compute the bias matrix, and then compute attention n add bias value too

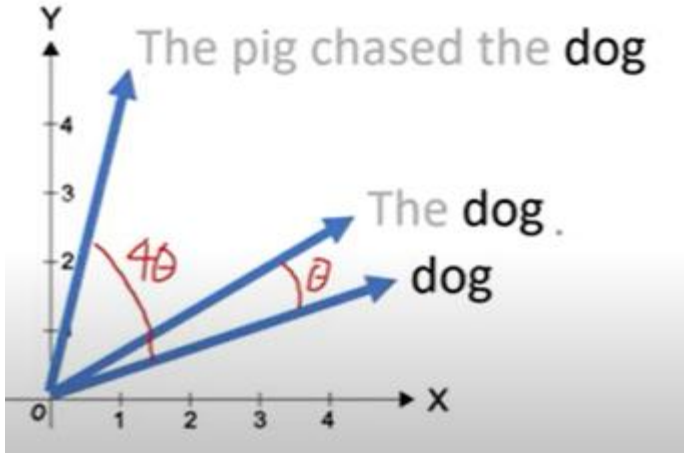


Pros
- Generalizes better to different sequence lengths.
- Improves understanding of word order relationships.
- More adaptable to long-context reasoning.

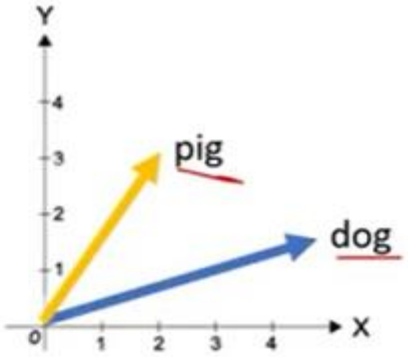
Cons:
- Higher computational cost due to bias matrix updates.
- More complex implementation than absolute encoding.
- Slightly increased inference time.

RoPE: Rotary Position Embedding

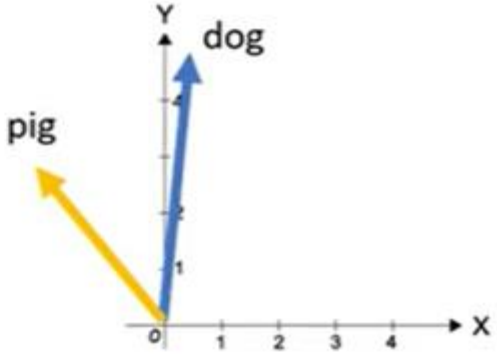
- It encodes positional information in a way that allows the model to understand both the absolute position (direct rotation) of tokens and their relative distances (dot product).
- This is achieved through a rotational mechanism, where each position in the sequence is represented by a rotation in the embedding space.

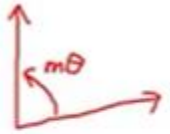


The pig chased the dog



Once upon a time, the pig chased the dog





$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

As long as the distance bw the two words remains the same, even if more words are added in the beginning or end, the angle bw those 2 words will remain the same.

Two words are rotated by exact same amount, if distance is preserved

$$R(x) = \begin{pmatrix} \cos(x) & -\sin(x) \\ \sin(x) & \cos(x) \end{pmatrix}$$

Figure-2: Euler's formula

1. **RoPE Uses Rotations to Encode Position**
   - Each token's **Q** and **K** vectors are **rotated** by a certain angle based on its absolute position.
   - The rotation matrix ensures that different tokens (at positions i and j) are rotated **differently**.
2. **Dot Product Naturally Encodes Relative Positions**
   - When we compute the dot product $Q_i \cdot K_j^T$, the **phase difference** (introduced by different rotation angles) **implicitly encodes** $j - i$, i.e., the relative position.
   - This means the model doesn't need to explicitly store or learn absolute position embeddings—it gets the relative order **for free**.
3. **Better for Extrapolation**
   - Since RoPE only applies rotations (which are periodic and unbounded), it generalizes better to longer sequences.
   - Unlike absolute position embeddings (which are learned and fixed), RoPE can handle unseen positions because **rotation matrices work consistently beyond training positions**.

the dot product between two rotated vectors contains information about how far apart their positions are.

Absolute positions are encoded as rotations.

Relative distances (order information) are captured as phase shifts.

Dot products in attention measure these phase shifts, allowing the model to understand ordering naturally.

Extrapolation works because phase shifts are predictable at unseen positions.

- https://medium.com/dair-ai/papers-explained-mistral-7b-b9632dedf580
- RoPE: https://www.youtube.com/watch?v=o29P0Kpobz0
- https://medium.com/@parulsharmmaa/understanding-rotary-positional-embedding-and-implementation-9f4ad8b03e32#:~:text=Unlike%20traditional%20positional%20embeddings%2C%20such,embeddings%20in%20a%20complex%20plane.
- https://medium.com/ai-insights-cobet/rotary-positional-embeddings-a-detailed-look-and-comprehensive-understanding-4ff66a874d83
- KV Cache: https://www.youtube.com/watch?v=80bIUggRJf4
- https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f
- https://ashishjaiman.medium.com/large-language-models-llms-260bf4f39007
- https://github.com/hkproj/mistral-src-commented
- https://github.com/hkproj/mistral-llm-notes