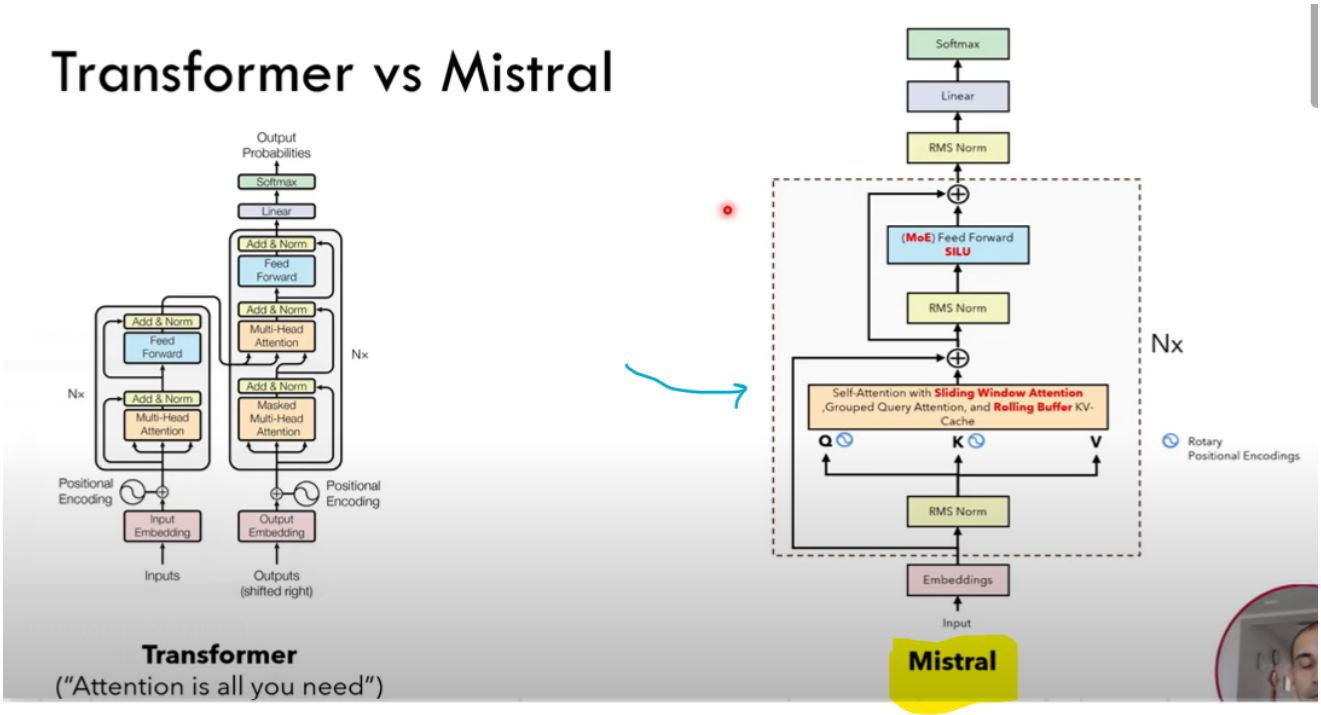
Video Link: https://youtu.be/UiX8K-xBUpE?feature=shared



Mistral similar to LLaMA, except the Red parts

8 FF layers for Mistral 7B, instead of only 1

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	

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**.

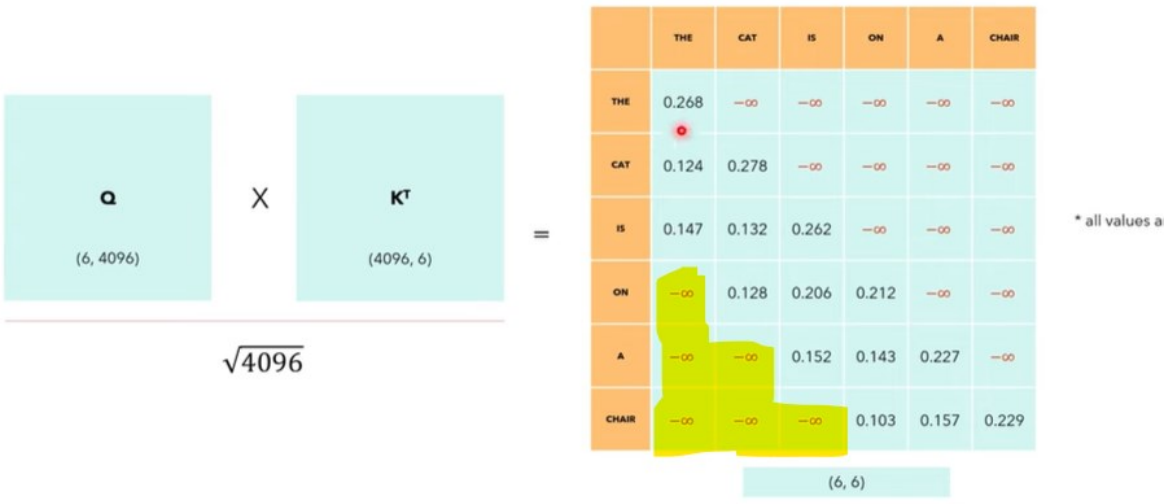
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Usual causal mask

Let's apply sliding window attention

The sliding window size is 3

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



Sliding Window Attention: details

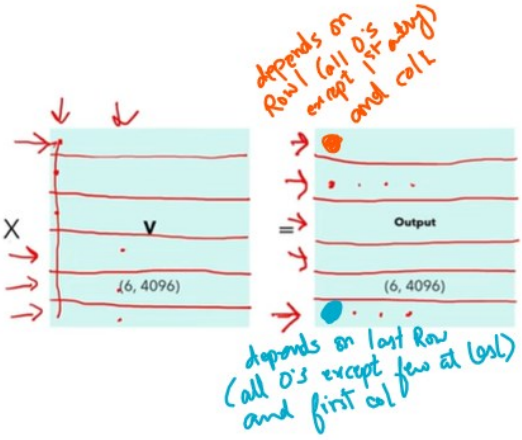
- Reduces the number of dot-products to perform, and thus, performance during training and inference.
- Sliding window attention may lead to degradation in the performance of the model, as some "interactions" between tokens will not be captured.
  The model mostly focuses on the **local context**, which 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.

Sliding Window Attention: information flow (2)

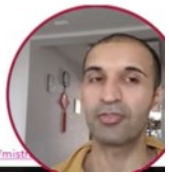
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

	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)

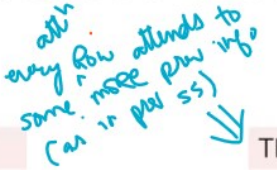


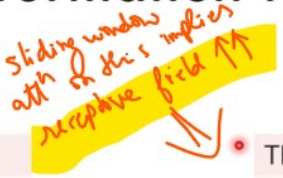
The output of the Self-Attention is a matrix of the same shape as the input sequence, but where each token now captures information about other tokens according to the mask applied. In our case, the last token of the output captures information about itself and the two preceding tokens.



Umar Jamil - https://github.com/hkproj/mist

Sliding Window Attention: information flow (3)





THE	THE	THE
CAT	THE, CAT	THE, CAT
IS	THE, CAT, IS	THE, CAT, IS
ON	CAT, IS, ON	THE, CAT, IS, ON
A	IS, ON, A	THE, CAT, IS, ON, A
CHAIR	ON, A, CHAIR	CAT, IS, ON, A, CHAIR
(6, 4096)	(6, 4096)	(6, 4096)

Layer 1 → **Layer 2**

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/CAT	0.461	0.538	0	0	0	0
THE/CAT/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.

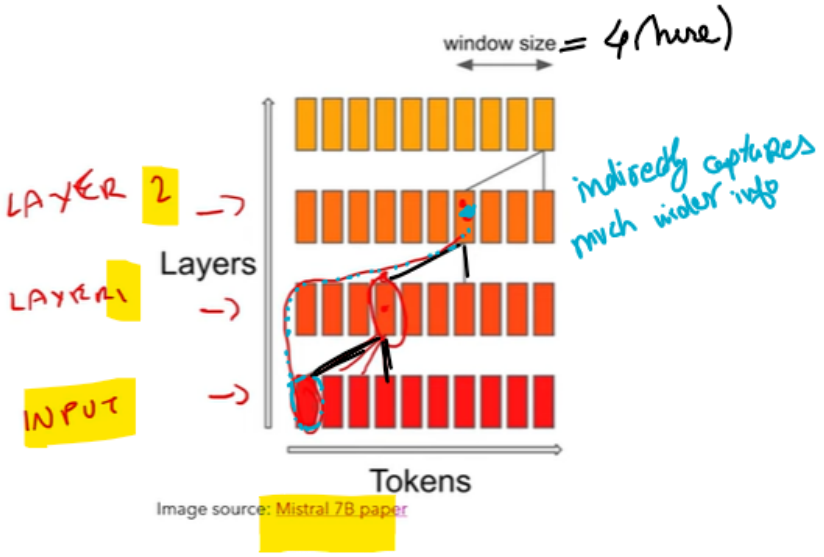
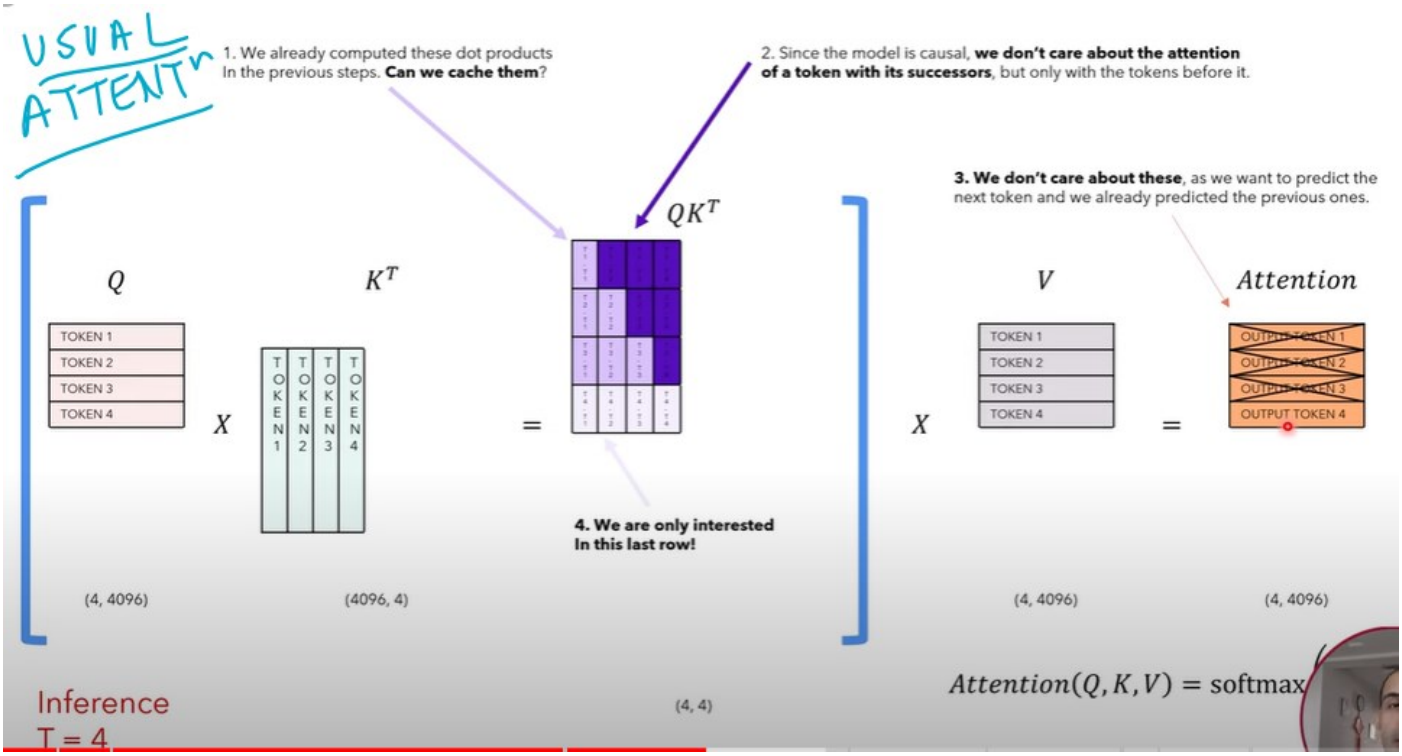
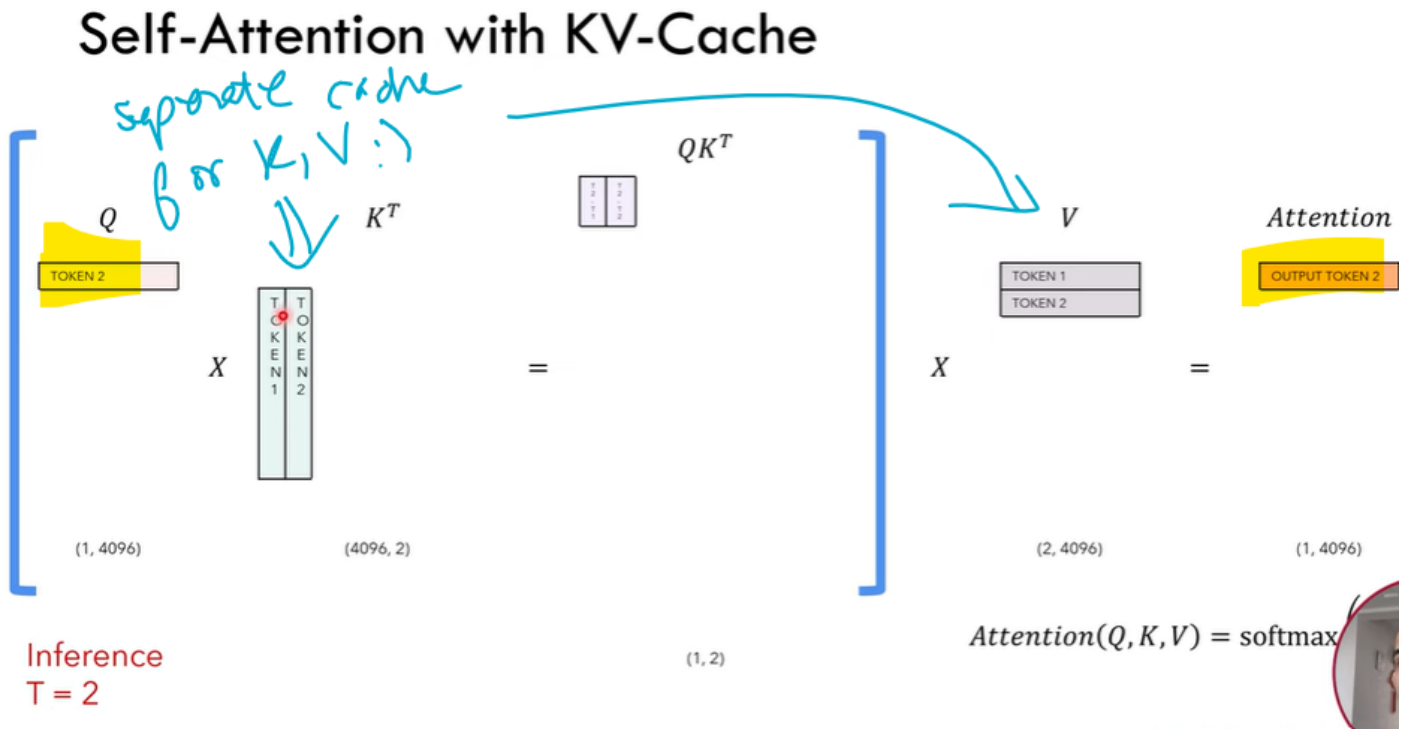


Image source: Mistral 7B paper



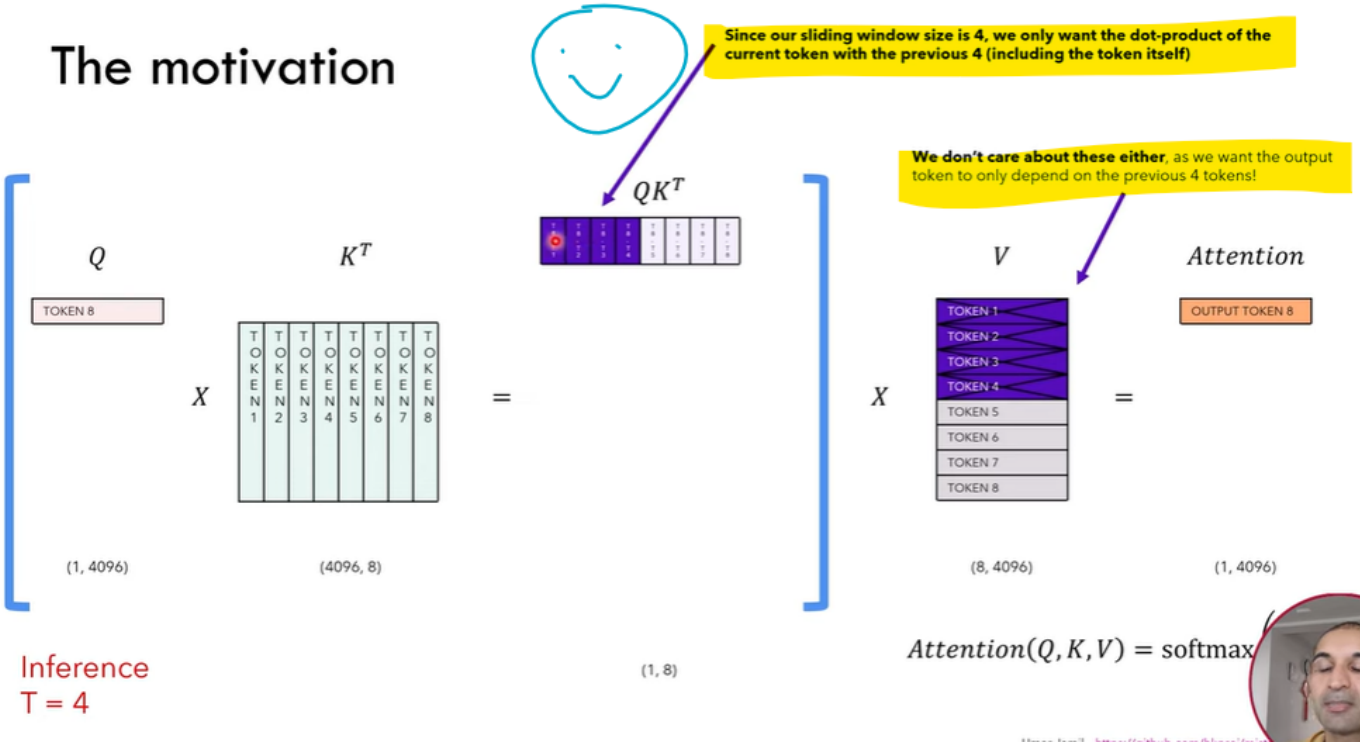


Rolling Buffer Cache

- 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.

OP! Way of limiting the cache

The motivation



Inference
T = 4

$$Attention(Q, K, V) = \text{softmax}$$

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	$-\infty$	$-\infty$	$-\infty$
YOU	0.124	0.278	$-\infty$	$-\infty$
TELL	0.147	0.132	0.262	$-\infty$
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**. We will see layer why.**

Pre-fill and chunking: second chunk

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

The KV-Cache

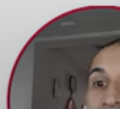
CAN	YOU	TELL	ME

The attention mask (current chunk)

	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.

Let's review how this is implemented in code



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()
    )
```

Query: Current chunk

Key: Current chunk + info in kv cache

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

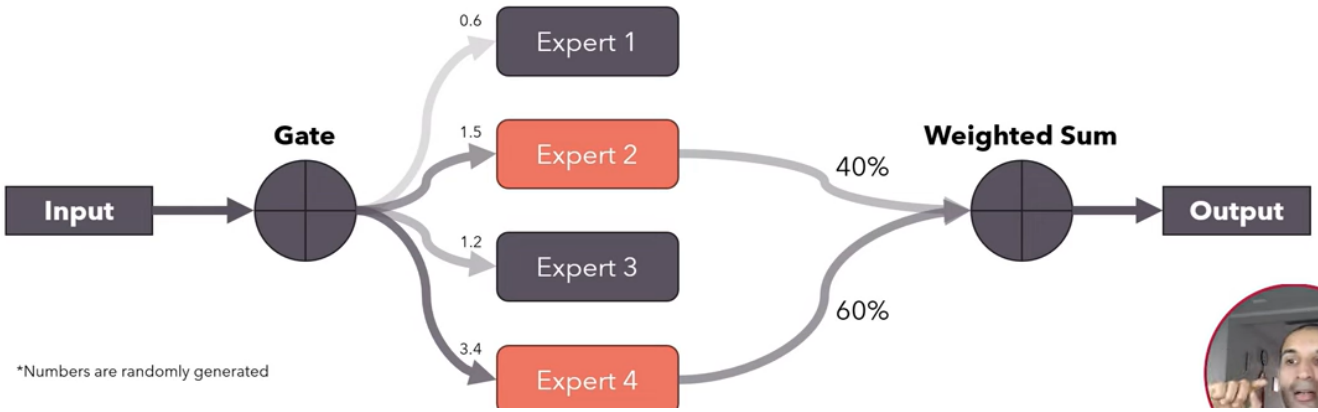
The last chunk may be smaller, that's why we have less rows.

Mixture of Experts: an introduction

Mixture of Experts is an ensemble technique, in which we have multiple "expert" models, each trained on a subset of the data, such that each model specializes on it and then the output of the experts are combined (usually a weighted sum or by averaging) to produce one single output.

In the case of **Mistral 8x7B**, we talk about **Sparse Mixture of Experts (SMoE)**, because only 2 out of 8 experts are used for every token.

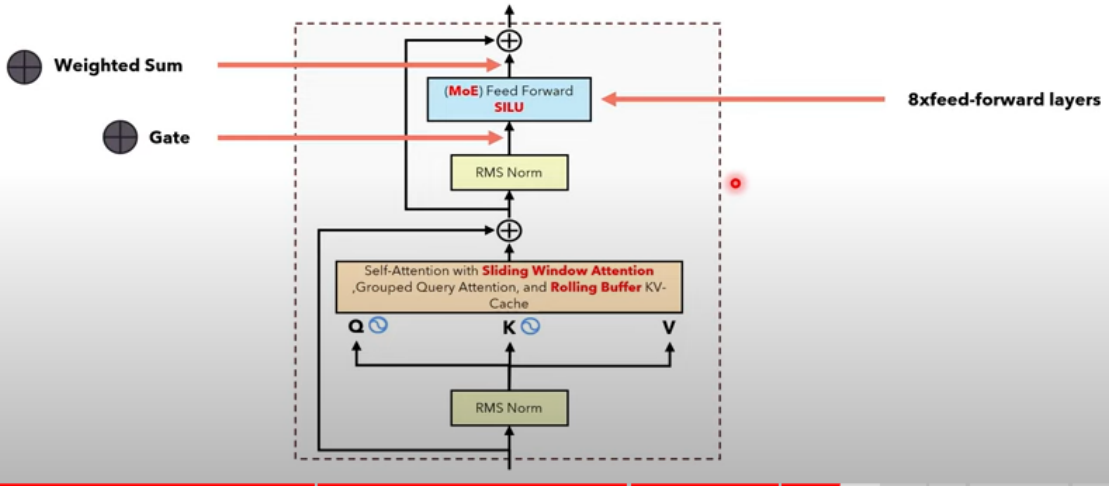
The gate produces **logits** which are used to select the top-k experts. The top-k logits are then run through a **softmax** to produce weights.



*Numbers are randomly generated

Mistral 8x7B: expert feed-forward layers

- In the case of Mistral 8x7B, the experts are the Feed-Forward layers present at every Encoder layer. Each Encoder layer is comprised of a single Self-Attention mechanism, followed by a mixture of experts of 8 FFN. The gate function selects the top 2 experts for each incoming token. The output is combined with a weighted sum.
- This allows to increase the parameters of the model, but without impacting the computation time, since the input will only pass through the top 2 experts, so the intermediate matrix multiplications will be performed only on the selected experts.



Mistral 8x7B : the gating function

- The gating function is just a linear layer **(in_features=4096, out_feature=8, bias=False)** that's trained along with the rest of the model. For each token embedding, it produces 8 logits, which indicate which expert to select.

Mistral 8x7B

```
self.feed_forward: nn.Module
if args.moe is not None:
    self.feed_forward = MoeLayer(
        experts=[FeedForward(args=args) for _ in range(args.moe.num_experts)],
        gate=nn.Linear(args.dim, args.moe.num_experts, bias=False),
        moe_args=args.moe,
    )
else:
    self.feed_forward = FeedForward(args=args)
```

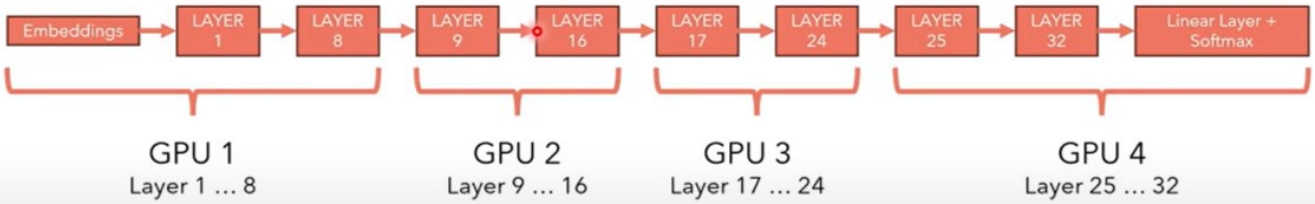
Mistral 7B

Model sharding

When we have a model that is too big to fit in a single GPU, we can divide the model into "groups of layers" and place each group of layers in a GPU. When we inference iteratively: **the output of each GPU is fed as input to the next GPU, and so on...**

This technique is known as model sharding.

In the case of Mistral, since we have 32 Encoder layers, if we have 4 GPUs, we can store 8 of them in each GPU.



A pipeline like this one works fine, but **it is not very efficient, because at any time only one GPU is working**. A better approach (**not** implemented in the open-source release of Mistral), used especially during training, is to work on multiple batches at the same time, but shift them on the time scale. This approach is known as **Pipeline Parallelism**. Let's see how it works.

Pipeline Parallelism: the problem

Imagine we want to train our sharded model on a single batch: it will take 8 timesteps to do it, and at each time step, only one GPU is working while the others are waiting idle.

Timestep	0	1	2	3	4	5	6	7
GPU 4				FORWARD	BACKWARD			
GPU 3			FORWARD			BACKWARD		
GPU 2		FORWARD					BACKWARD	
GPU 1	FORWARD							BACKWARD

Pipeline Parallelism: the solution

We divide our batch into smaller microbatches, and shift each microbatch's forward and backward step on the timeline.

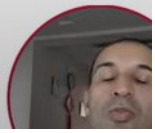
Each timestep in this case takes less, since we are working on a small microbatch. The gradients for each microbatch is accumulated (**gradient accumulation**) and then we can run the optimizer to update the weights.

Timestep	0	1	2	3	4	5	6	7	8	9	10	11	12	13
GPU 4				FW-1	FW-2	FW-3	FW-4	BCW-4	BCK-3	BCK-2	BCK-1			
GPU 3			FW-1	FW-2	FW-3	FW-4			BCW-4	BCK-3	BCK-2	BCK-1		
GPU 2		FWD-1	FW-2	FW-3	FW-4					BCW-4	BCK-3	BCK-2	BCK-1	
GPU 1	FWD-1	FW-2	FW-3	FW-4							BCW-4	BCK-3	BCK-2	BCK-

xformers BlockDiagonalCausalMask

	Write	a	poem	Write	a	historical	novel	Tell	me	a	funny	joke
Write	0	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞
a	0	0	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞
poem	0	0	0	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞
Write	−∞	−∞	−∞	0	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞
a	−∞	−∞	−∞	0	0	−∞	−∞	−∞	−∞	−∞	−∞	−∞
historical	−∞	−∞	−∞	0	0	0	−∞	−∞	−∞	−∞	−∞	−∞
novel	−∞	−∞	−∞	0	0	0	0	−∞	−∞	−∞	−∞	−∞
Tell	−∞	−∞	−∞	−∞	−∞	−∞	−∞	0	−∞	−∞	−∞	−∞
me	−∞	−∞	−∞	−∞	−∞	−∞	−∞	0	0	−∞	−∞	−∞
a	−∞	−∞	−∞	−∞	−∞	−∞	−∞	0	0	0	−∞	−∞
funny	−∞	−∞	−∞	−∞	−∞	−∞	−∞	0	0	0	0	−∞
joke	−∞	−∞	−∞	−∞	−∞	−∞	−∞	0	0	0	0	0

The mask may be different in case we use Sliding Window Attention.
Let's have a look at the code!



Apparently better solution ithan combining each in a batch is to combine everything in one big sentence an dapply this block diagonal causal mask