LLM NOTES

A PREPRINT

August 1, 2025

Contents

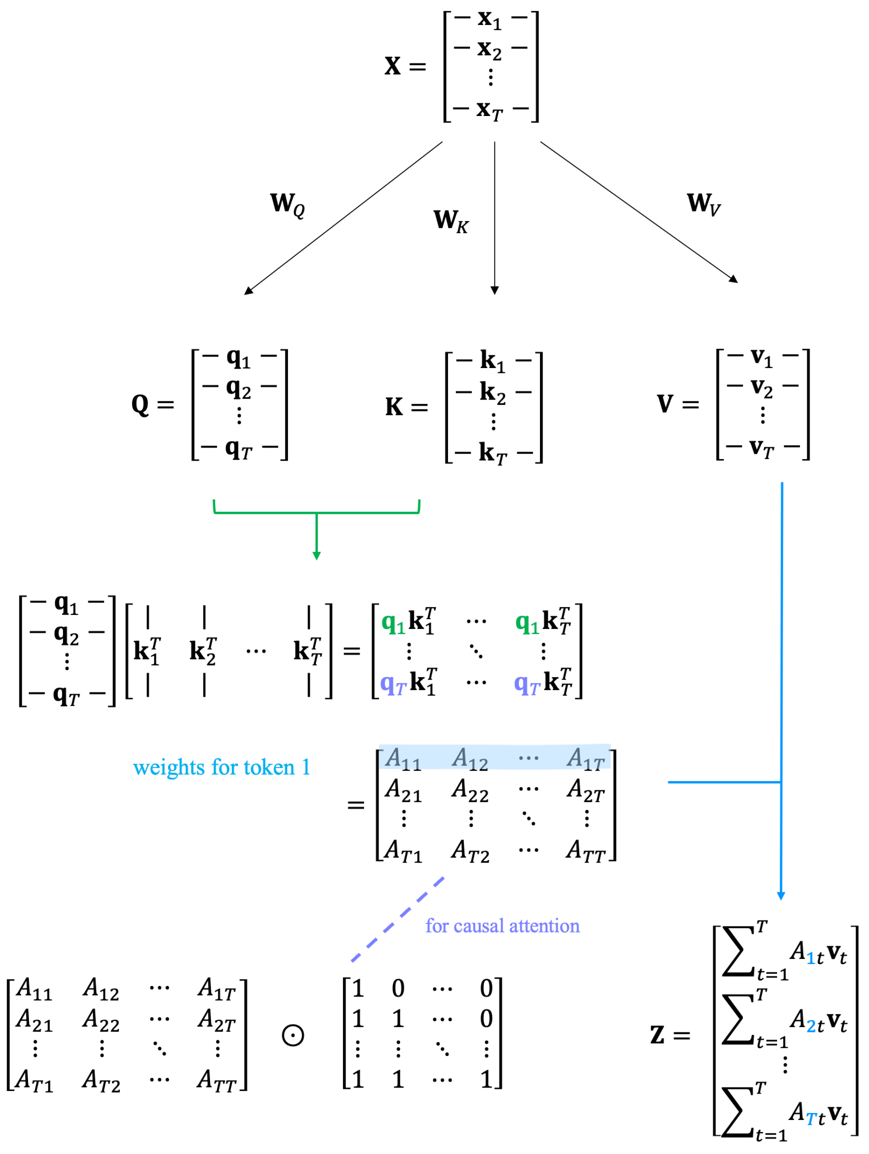
1 Attention

1.1 Self Attention

Given input $\boldsymbol{X} \in \mathbb{R}^{T\times D}$ where T is the sequence length and D is the embedding dimension, we have

$$\boldsymbol{Q} = \boldsymbol{W}_Q\boldsymbol{X} \in \mathbb{R}^{T\times d_k} \quad \boldsymbol{K} = \boldsymbol{W}_K\boldsymbol{X} \in \mathbb{R}^{T\times d_k} \quad \boldsymbol{V} = \boldsymbol{W}_V\boldsymbol{X} \in \mathbb{R}^{T\times d_v} \tag{1}$$

$$\text{Attn}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{2}$$

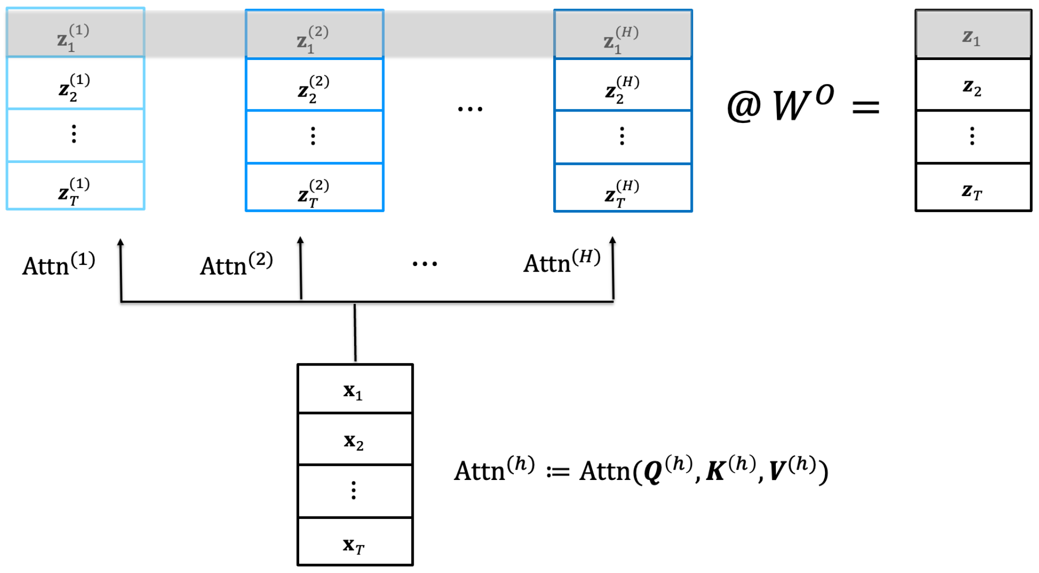


$\mathbf{X} = \begin{bmatrix} - \mathbf{x}_1 - \\ - \mathbf{x}_2 - \\ \vdots \\ - \mathbf{x}_T - \end{bmatrix}$
$\mathbf{W}_Q$
$\mathbf{W}_K$
$\mathbf{W}_V$
$\mathbf{Q} = \begin{bmatrix} - \mathbf{q}_1 - \\ - \mathbf{q}_2 - \\ \vdots \\ - \mathbf{q}_T - \end{bmatrix}$
$\mathbf{K} = \begin{bmatrix} - \mathbf{k}_1 - \\ - \mathbf{k}_2 - \\ \vdots \\ - \mathbf{k}_T - \end{bmatrix}$
$\mathbf{V} = \begin{bmatrix} - \mathbf{v}_1 - \\ - \mathbf{v}_2 - \\ \vdots \\ - \mathbf{v}_T - \end{bmatrix}$
$\begin{bmatrix} - \mathbf{q}_1 - \\ - \mathbf{q}_2 - \\ \vdots \\ - \mathbf{q}_T - \end{bmatrix}\begin{bmatrix} | & | & & | \\ \mathbf{k}_1^T & \mathbf{k}_2^T & \cdots & \mathbf{k}_T^T \\ | & | & & | \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1\mathbf{k}_1^T & \cdots & \mathbf{q}_1\mathbf{k}_T^T \\ \vdots & \ddots & \vdots \\ \mathbf{q}_T\mathbf{k}_1^T & \cdots & \mathbf{q}_T\mathbf{k}_T^T \end{bmatrix}$
weights for token 1
$= \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1T} \\ A_{21} & A_{22} & \cdots & A_{2T} \\ \vdots & \vdots & \ddots & \vdots \\ A_{T1} & A_{T2} & \cdots & A_{TT} \end{bmatrix}$
for causal attention
$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1T} \\ A_{21} & A_{22} & \cdots & A_{2T} \\ \vdots & \vdots & \ddots & \vdots \\ A_{T1} & A_{T2} & \cdots & A_{TT} \end{bmatrix} \odot \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$
$\mathbf{Z} = \begin{bmatrix} \sum_{t=1}^{T} A_{1t}\mathbf{v}_t \\ \sum_{t=1}^{T} A_{2t}\mathbf{v}_t \\ \vdots \\ \sum_{t=1}^{T} A_{Tt}\mathbf{v}_t \end{bmatrix}$

1.2 Multi-Head Attention

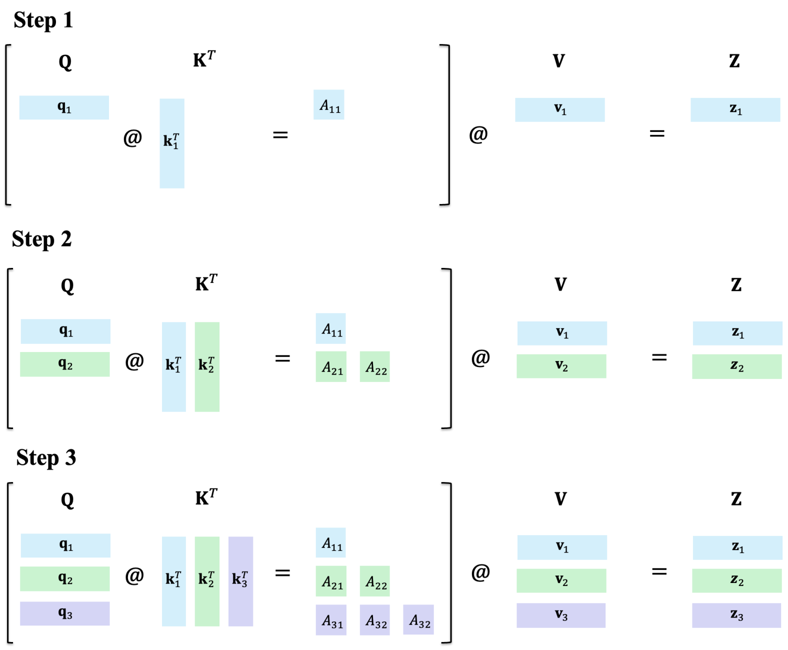
In multi-head attention, input $\boldsymbol{X}$ is first passed through H self-attention layer in parallel. Then, the output from each head is concatenated together and fused by a linear projection

$$\left[\boldsymbol{Z}^{(1)}, \boldsymbol{Z}^{(2)}, \ldots, \boldsymbol{Z}^{(H)}\right] \boldsymbol{W}^{O} = \begin{bmatrix} \boldsymbol{z}_1^{(1)} & \boldsymbol{z}_1^{(2)} & \ldots & \boldsymbol{z}_1^{(H)} \\ \boldsymbol{z}_2^{(1)} & \boldsymbol{z}_2^{(2)} & \ldots & \boldsymbol{z}_2^{(H)} \\ \vdots & \vdots & \ldots & \vdots \\ \boldsymbol{z}_T^{(1)} & \boldsymbol{z}_T^{(2)} & \ldots & \boldsymbol{z}_T^{(H)} \end{bmatrix} \boldsymbol{W}^{O} \tag{3}$$



1.3 KV-Cache

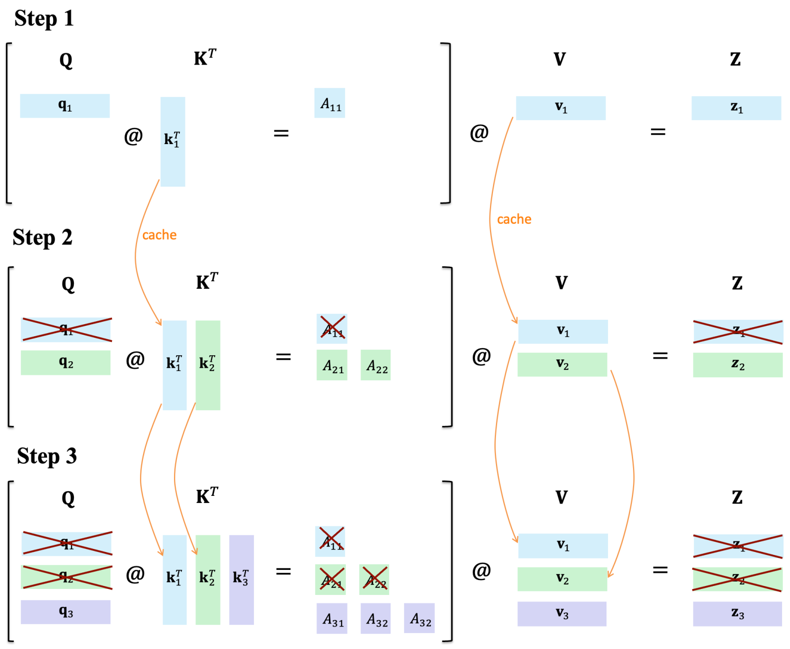
During inference we still use causal masking because this is how the model being trained. Let's look at a simple case where we only give the model a start token <s> and asks it to generate stuff:



KV-cache is built on this two observations:

- At each time step t, due to causal masking $\boldsymbol{k}_{<t}$ and $\boldsymbol{v}_{<t}$ will remain the same
- To predict <token$_{t+1}$> we only need embedding of <token$_t$>.

Therefore, we can make prediction efficiently by drop redundant and unnecessary computation



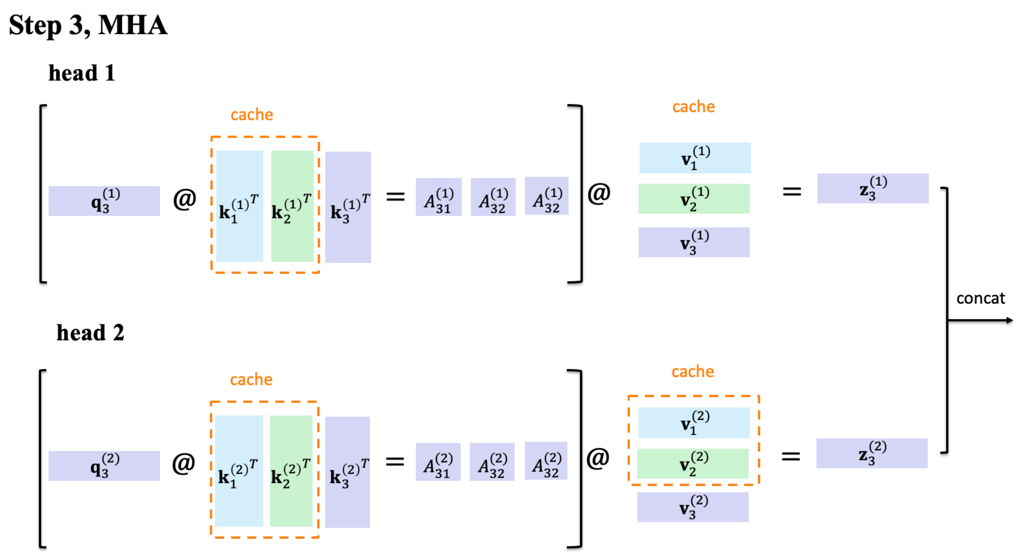
1.4 Multi-Query Attention

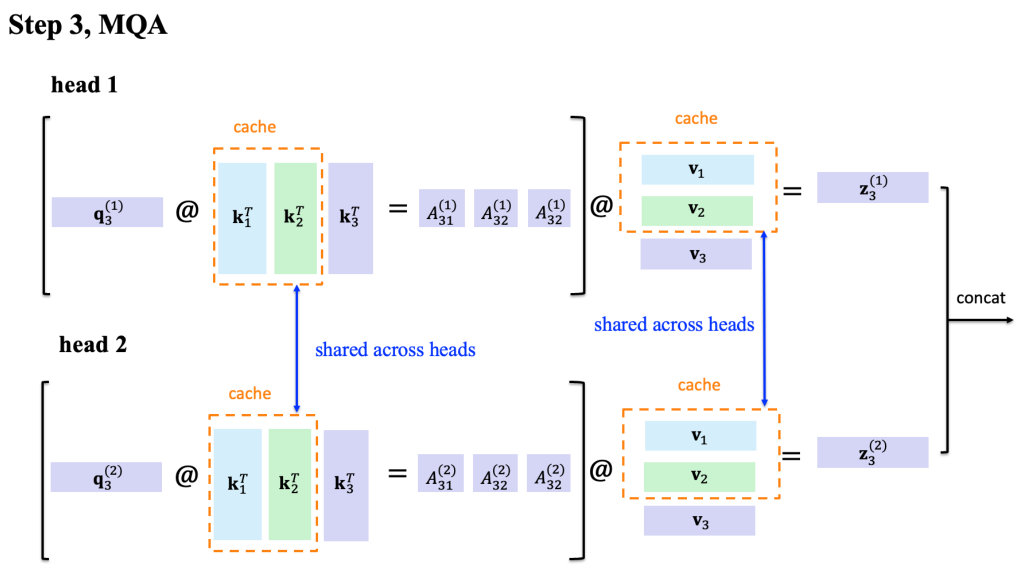
In KV-cache, at time step t, the past $t-1$ keys and values are cached, which corresponds to $2 * n_h * (t-1) * h_s$ memory. When t gets larger, storing them all on a single GPU will be impossible and the loading will slow things down.

The idea of multi-query attention (MQA) is quite simple. In MHA, different heads have different query, key and value projection matrix. In MQA, the key and value projection matrics will be shared across all heads, only query project matrix will be different:

$$\begin{array}{ll} \text{MHA} & \text{MQA} \\ \boldsymbol{Q}^{(h)} = \boldsymbol{W}_Q^{(h)}\boldsymbol{X} & \boldsymbol{Q}^{(h)} = \boldsymbol{W}_Q^{(h)}\boldsymbol{X} \\ \boldsymbol{K}^{(h)} = \boldsymbol{W}_K^{(h)}\boldsymbol{X} & \boldsymbol{K}^{(h)} = \boldsymbol{W}_K\boldsymbol{X} \quad \text{(shared across heads)} \\ \boldsymbol{V}^{(h)} = \boldsymbol{W}_V^{(h)}\boldsymbol{X} & \boldsymbol{V}^{(h)} = \boldsymbol{W}_V\boldsymbol{X} \quad \text{(shared across heads)} \end{array} \tag{4}$$

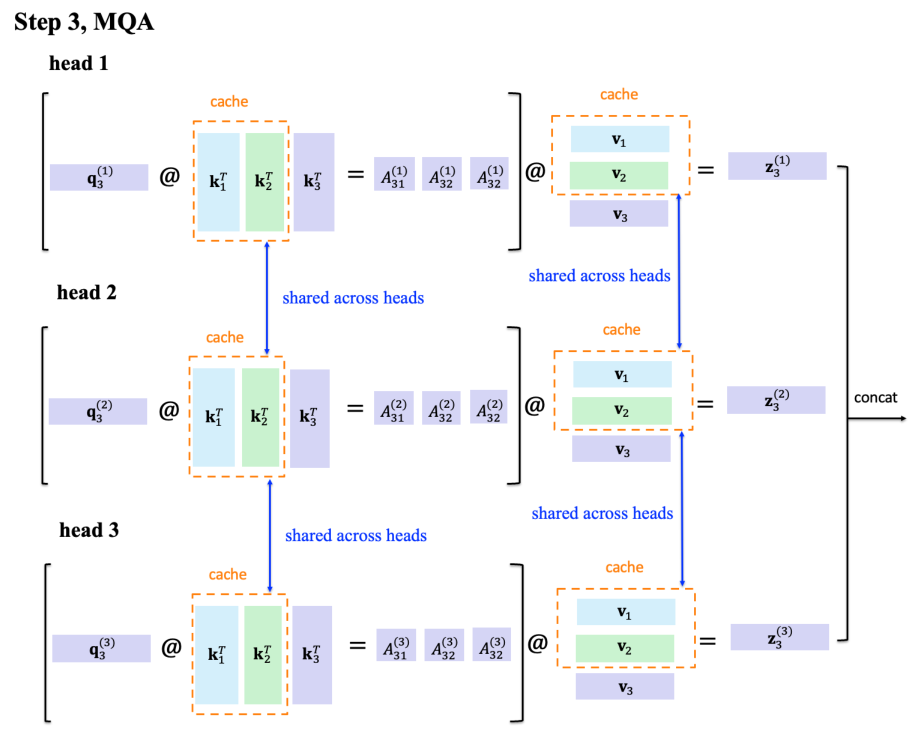
This way, we only need to store one set of KV-cache and share it across all heads.

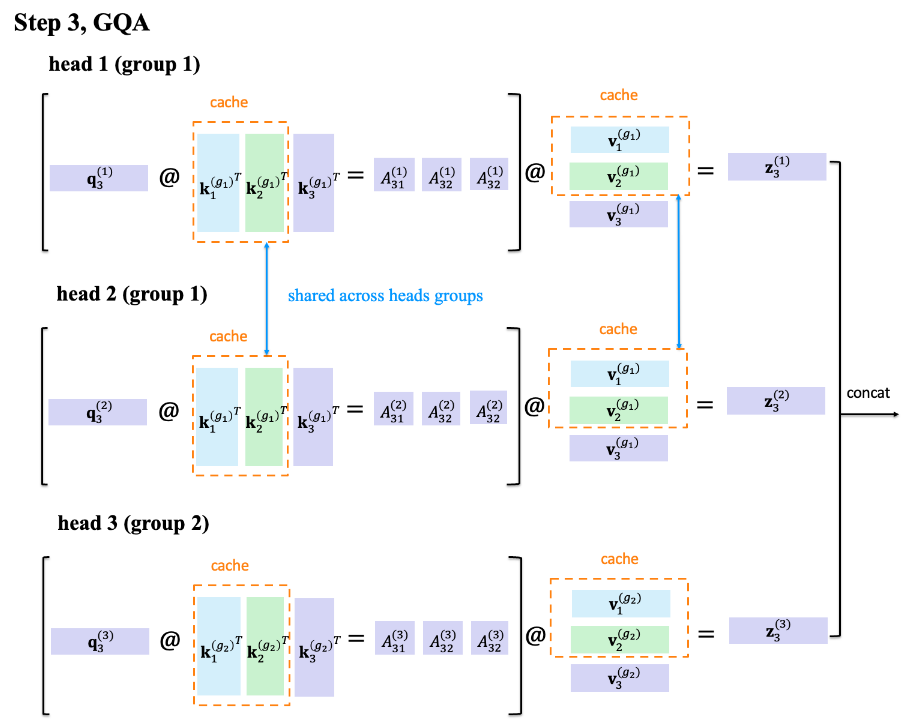

Step 3, MHA
head 1
cache
$\mathbf{q}_3^{(1)}$ @ $\mathbf{k}_1^{(1)T}$ $\mathbf{k}_2^{(1)T}$ $\mathbf{k}_3^{(1)T}$ = $A_{31}^{(1)}$ $A_{32}^{(1)}$ $A_{32}^{(1)}$ @ $\mathbf{v}_1^{(1)}$ $\mathbf{v}_2^{(1)}$ $\mathbf{v}_3^{(1)}$ = $\mathbf{z}_3^{(1)}$
concat
head 2
cache
$\mathbf{q}_3^{(2)}$ @ $\mathbf{k}_1^{(2)T}$ $\mathbf{k}_2^{(2)T}$ $\mathbf{k}_3^{(2)T}$ = $A_{31}^{(2)}$ $A_{32}^{(2)}$ $A_{32}^{(2)}$ @ $\mathbf{v}_1^{(2)}$ $\mathbf{v}_2^{(2)}$ $\mathbf{v}_3^{(2)}$ = $\mathbf{z}_3^{(2)}$


Step 3, MQA
head 1
cache
$\mathbf{q}_3^{(1)}$ @ $\mathbf{k}_1^T$ $\mathbf{k}_2^T$ $\mathbf{k}_3^T$ = $A_{31}^{(1)}$ $A_{32}^{(1)}$ $A_{32}^{(1)}$ @ $\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ = $\mathbf{z}_3^{(1)}$
concat
shared across heads
shared across heads
head 2
cache
$\mathbf{q}_3^{(2)}$ @ $\mathbf{k}_1^T$ $\mathbf{k}_2^T$ $\mathbf{k}_3^T$ = $A_{31}^{(2)}$ $A_{32}^{(2)}$ $A_{32}^{(2)}$ @ $\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ = $\mathbf{z}_3^{(2)}$

1.5 Grouped-Query Attention

Sharing the key and value projection across all attention heads might be too brutal, in grouped-query attention, $\boldsymbol{W}_K$ and $\boldsymbol{W}_V$ are shared in grouped heads. So basically different groups have different $\boldsymbol{K}$ and $\boldsymbol{V}$.


Step 3, MQA
head 1
head 2
head 3
cache
shared across heads
concat
$\mathbf{q}_3^{(1)}$ @ $\mathbf{k}_1^T$ $\mathbf{k}_2^T$ $\mathbf{k}_3^T$ = $A_{31}^{(1)}$ $A_{32}^{(1)}$ $A_{32}^{(1)}$ @ $\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ = $\mathbf{z}_3^{(1)}$
$\mathbf{q}_3^{(2)}$ @ $\mathbf{k}_1^T$ $\mathbf{k}_2^T$ $\mathbf{k}_3^T$ = $A_{31}^{(2)}$ $A_{32}^{(2)}$ $A_{32}^{(2)}$ @ $\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ = $\mathbf{z}_3^{(2)}$
$\mathbf{q}_3^{(3)}$ @ $\mathbf{k}_1^T$ $\mathbf{k}_2^T$ $\mathbf{k}_3^T$ = $A_{31}^{(3)}$ $A_{32}^{(3)}$ $A_{32}^{(3)}$ @ $\mathbf{v}_1$ $\mathbf{v}_2$ $\mathbf{v}_3$ = $\mathbf{z}_3^{(3)}$


Step 3, GQA
head 1 (group 1)
head 2 (group 1)
head 3 (group 2)
cache
shared across heads groups
concat
$\mathbf{q}_3^{(1)}$ @ $\mathbf{k}_1^{(g_1)T}$ $\mathbf{k}_2^{(g_1)T}$ $\mathbf{k}_3^{(g_1)T}$ = $A_{31}^{(1)}$ $A_{32}^{(1)}$ $A_{32}^{(1)}$ @ $\mathbf{v}_1^{(g_1)}$ $\mathbf{v}_2^{(g_1)}$ $\mathbf{v}_3^{(g_1)}$ = $\mathbf{z}_3^{(1)}$
$\mathbf{q}_3^{(2)}$ @ $\mathbf{k}_1^{(g_1)T}$ $\mathbf{k}_2^{(g_1)T}$ $\mathbf{k}_3^{(g_1)T}$ = $A_{31}^{(2)}$ $A_{32}^{(2)}$ $A_{32}^{(2)}$ @ $\mathbf{v}_1^{(g_1)}$ $\mathbf{v}_2^{(g_1)}$ $\mathbf{v}_3^{(g_1)}$ = $\mathbf{z}_3^{(2)}$
$\mathbf{q}_3^{(3)}$ @ $\mathbf{k}_1^{(g_2)T}$ $\mathbf{k}_2^{(g_2)T}$ $\mathbf{k}_3^{(g_2)T}$ = $A_{31}^{(3)}$ $A_{32}^{(3)}$ $A_{32}^{(3)}$ @ $\mathbf{v}_1^{(g_2)}$ $\mathbf{v}_2^{(g_2)}$ $\mathbf{v}_3^{(g_2)}$ = $\mathbf{z}_3^{(3)}$

2 Positional Embedding

2.1 RoPE

Once and for all vs add all the time In vanilla position embedding, it adds a vector at the beginning of the NN once and for all. The positional encoding might get lost in deep networks. If we look at the computation in transformer, the position only matters in the attention score computation. So RoPE encode position into query and key vector in every attention layer instead.

Absolute position vs relative position Another thing RoPE did is encode the *relative* position. The idea is, for a given word when computing its attention score, only the relative position between this word and other word should matter. Specifically, if we add the position into the computation of query and key computation:

$$\boldsymbol{q}_m \triangleq f_q(\boldsymbol{x}_m, m), \qquad \boldsymbol{k}_n \triangleq f_k(\boldsymbol{x}_n, n) \tag{5}$$

Then the attention score (which is an inner product) should only depends on the word embeddings $\boldsymbol{x}_m$, $\boldsymbol{x}_n$ and their relative position $m - n$:

$$\boldsymbol{q}_m^T \boldsymbol{k}_n = \langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n)\rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, m - n) \tag{6}$$

One simple choice for $f_{\{q,k\}}(\boldsymbol{x}_m, m)$ is rotate them:

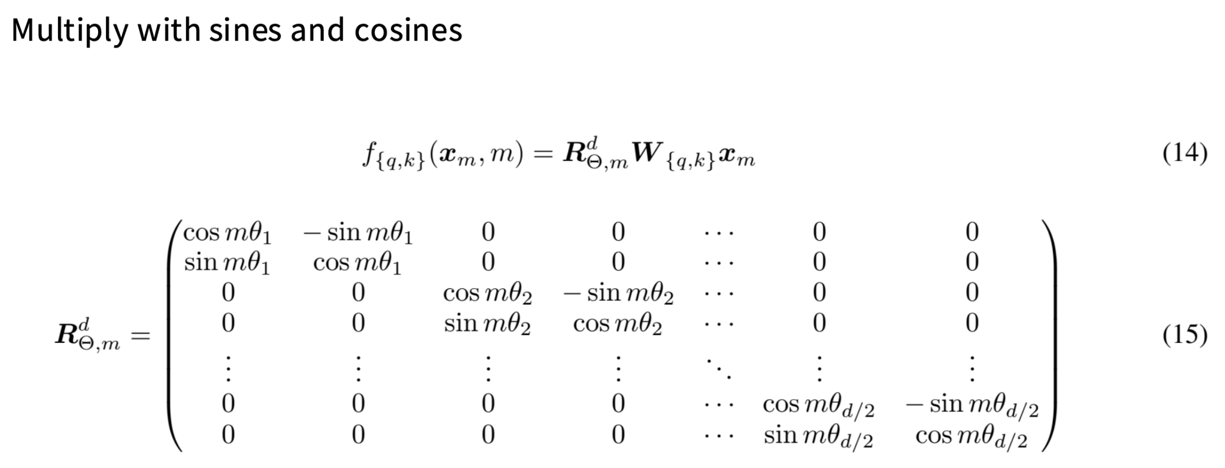
$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}(m)\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m \tag{7}$$

where $\boldsymbol{R}(m)$ is a rotation matrix whose angle depends on m.

Define $\widetilde{\boldsymbol{q}}_m = \boldsymbol{W}_q\boldsymbol{x}_m$ and $\widetilde{\boldsymbol{k}}_m = \boldsymbol{W}_k\boldsymbol{x}_m$, now we have

$$(\boldsymbol{R}(m)\boldsymbol{W}_q\boldsymbol{x}_m)^T(\boldsymbol{R}(n)\boldsymbol{W}_k\boldsymbol{x}_n) = \widetilde{\boldsymbol{q}}_m^T \boldsymbol{R}(m)^T\boldsymbol{R}(n)\widetilde{\boldsymbol{k}}_n = \widetilde{\boldsymbol{q}}_m^T \boldsymbol{R}(n-m)\widetilde{\boldsymbol{k}}_n \tag{8}$$

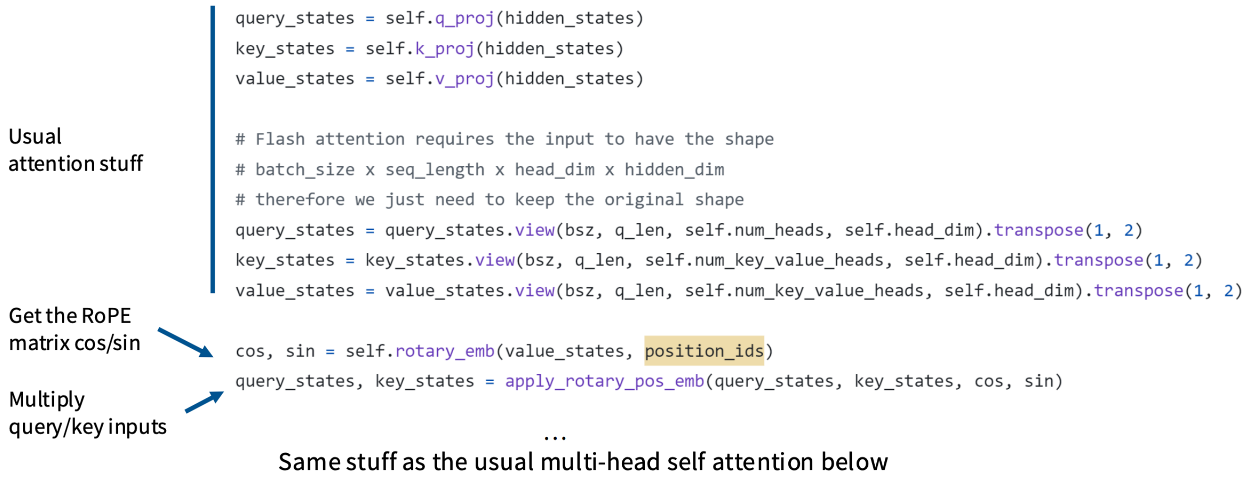
Because rotation in higher dimension than 2D is not deterministic, it does a funky thing to split query vector in a bunch of 2D pairs, and then rotate them each.



Multiply with sines and cosines

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m \tag{14}$$

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix} \tag{15}$$



```
query_states = self.q_proj(hidden_states)
key_states = self.k_proj(hidden_states)
value_states = self.v_proj(hidden_states)

# Flash attention requires the input to have the shape
# batch_size x seq_length x head_dim x hidden_dim
# therefore we just need to keep the original shape
query_states = query_states.view(bsz, q_len, self.num_heads, self.head_dim).transpose(1, 2)
key_states = key_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).transpose(1, 2)
value_states = value_states.view(bsz, q_len, self.num_key_value_heads, self.head_dim).transpose(1, 2)

cos, sin = self.rotary_emb(value_states, position_ids)
query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos, sin)

...
```

Same stuff as the usual multi-head self attention below

Note: embedding at *each attention operation* to enforce position invariance

References