

Transformer with PyTorch

Richard Xu

June 22, 2026

Contents

1	Dot-Product Attention (DPA)	2
1.1	single query $\mathbf{q}$	2
1.2	Scaled Dot-Product Attention (SDPA)	2
1.3	in the case of “seq2seq with attention”	3
1.4	multiple queries	3
2	Self-attention and Multi-head Attention	4
2.1	Detailed implementation	5
2.1.1	class CausalSelfAttention(nn.Module)	5
2.1.2	Step of each Transformer Layer	6
2.1.3	class TransformerBlock(nn.Module)	10
2.1.4	class TransformerLM(nn.Module)	11
2.2	Cross-Attention	14
3	Recent Developments	14
3.1	$\mathbf{K} - \mathbf{V}$ caching	14
3.2	Multi-head latent attention	15
3.3	Rotary Embedding RoPE	18
3.4	Decoupled RoPE	22
3.5	Lightening Indexer	24
4	Flash Attention	25
4.1	traditional softmax algorithm	25
4.2	online softmax algorithm	26
4.3	online attention	27
4.3.1	final algorithm	29
5	Manifold-Constrained Hyper-Connections (mHC)	29
5.1	issue with the implementation	31
5.2	need $\mathcal{H}_l^{\text{res}}$ to be doubly stochastic	32

1 Dot-Product Attention (DPA)

1.1 single query $\mathbf{q}$

The very famous paper, “Attention Is All You Need” (Google) [1] introduced the concept of attention, which is a key component of the Transformer architecture

We are given a single query vector $\mathbf{q}$, and matrix of keys $\mathbf{K}$ and values $\mathbf{V}$:

$$\mathbf{q} \equiv \underbrace{\begin{bmatrix} - & \mathbf{q} & - \end{bmatrix}}_{\in \mathbb{R}^{1 \times d_k}} \quad \mathbf{K} \equiv \underbrace{\begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_m & - \end{bmatrix}}_{\in \mathbb{R}^{m \times d_k}} \quad \mathbf{V} \equiv \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix}}_{\in \mathbb{R}^{m \times d_v}} \quad (1)$$

and let $(\mathbf{q}, \mathbf{K}, \mathbf{V})$ be tuples. Bear in mind that in Eq.(1), $\mathbf{q}$ is a row vector. This is to conform with Eq.(7), where all the $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ have each element expressed as a row vector.

the Dot-Product Attention (DPA) is defined as:

$$\begin{aligned} \Rightarrow \mathbf{qK}^\top &= \begin{bmatrix} \underbrace{\mathbf{qk}_1^\top}_{\in \mathbb{R}} & \dots & \underbrace{\mathbf{qk}_m^\top}_{\in \mathbb{R}} \end{bmatrix} \\ \Rightarrow A(\mathbf{q}, \mathbf{K}, \mathbf{V}) &\equiv \text{softmax}(\mathbf{qK}^\top) \mathbf{V} \\ &= \text{softmax} \left(\begin{bmatrix} \mathbf{qk}_1^\top & \dots & \mathbf{qk}_m^\top \end{bmatrix} \right) \begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix} \\ &= \sum_{i=1}^m \frac{\exp[\mathbf{qk}_i^\top]}{\underbrace{\sum_j \exp[\mathbf{qk}_j^\top]}_{\mathbf{a}_i}} \mathbf{v}_i \\ &\in \mathbb{R}^{d_v} \end{aligned} \quad (2)$$

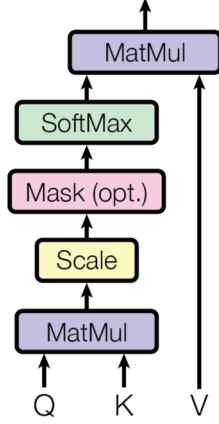
1.2 Scaled Dot-Product Attention (SDPA)

the **problem** starts as d_k becomes larger, variance of $\mathbf{qk}^\top$ increases. Then as a result, some dot product values gets very large, with $\exp(\cdot)$, softmax $\mathbf{p}$ gets peaky! Remember derivative of cross-entropy between softmax $\mathbf{p}$ and $\mathbf{y}$ is:

$$\begin{aligned} \mathbf{C}(\mathbf{z}) &= - \sum_{k=1}^K y_k [\log(p_k)] = - \sum_{k=1}^K y_k \left[\log \left(\frac{\exp^{z_k}}{\sum_l \exp^{z_l}} \right) \right] \\ \Rightarrow \frac{\mathbf{C}(\mathbf{z})}{\partial \mathbf{z}} &= (\mathbf{p} - \mathbf{y}) \end{aligned} \quad (3)$$

with a peaky softmax, lots of element in gradient vector $\frac{\mathbf{C}(\mathbf{z})}{\partial \mathbf{z}}$ are zero!
the **solution** then becomes that of scaling by length of d_k :

$$A(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}} \right) \mathbf{V} \quad (4)$$



“mask (opt.)” is only used at decoder during training

1.3 in the case of “seq2seq with attention”

Following the seq2seq framework [2] with the attention mechanism of [3], we have:

$$\begin{aligned}
 \mathbf{q} &\equiv \mathbf{h}_i \\
 \mathbf{k}_i &= \mathbf{v}_i = \mathbf{z}_i \\
 \implies A(\mathbf{q}, \mathbf{K}, \mathbf{V}) &\equiv A(\mathbf{h}_i, \mathbf{Z}, \mathbf{Z}) = c_i
 \end{aligned} \tag{5}$$

where $\mathbf{h}_i$ is our **conditional** or **context** vector. In order to confirm the notation in Eq.(1), we have assumed $\mathbf{h}_i$ to be a row vector, so instead of $\mathbf{h}_i^\top \mathbf{z}_j$, we express it as $\mathbf{h}_i \mathbf{z}_j^\top$:

$$a_{ij} = \frac{\exp(e_{i,j})}{\sum_{t=1}^m \exp(e_{i,t})} = \frac{\exp(\mathbf{h}_i \mathbf{z}_j^\top)}{\sum_{t=1}^m \exp(\mathbf{h}_i \mathbf{z}_t^\top)} \tag{6}$$

1.4 multiple queries

now we have multiple $\mathbf{Q} = \{\mathbf{q}_i\}$, e.g., N words in the decoder, we now have $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ expressed as:

$$\mathbf{Q} \equiv \underbrace{\begin{bmatrix} - & \mathbf{q}_1 & - \\ - & \dots & - \\ - & \mathbf{q}_n & - \end{bmatrix}}_{n \times d_k} \quad \mathbf{K} \equiv \underbrace{\begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_m & - \end{bmatrix}}_{m \times d_k} \quad \mathbf{V} \equiv \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix}}_{m \times d_v} \tag{7}$$

$A(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$ can be expressed as follows:

$$\implies \mathbf{Q}\mathbf{K}^\top = \begin{bmatrix} \mathbf{q}_1 \mathbf{k}_1^\top & \dots & \mathbf{q}_1 \mathbf{k}_m^\top \\ \dots & \dots & \dots \\ \mathbf{q}_n \mathbf{k}_1^\top & \dots & \mathbf{q}_n \mathbf{k}_m^\top \end{bmatrix} \tag{8}$$

in the case of self attention, where $m = n$, then $\mathbf{QK}^\top$ is a square matrix of $n \times n$.

$$\begin{aligned}
\Rightarrow \mathbf{O} &\equiv \text{softmax}(\mathbf{QK}^\top) \mathbf{V} \\
&= \mathbf{AV} \\
&= \underbrace{\begin{bmatrix} \text{softmax}([\mathbf{q}_1 \mathbf{k}_1^\top & \dots & \mathbf{q}_1 \mathbf{k}_m^\top]) \\ \vdots \\ \text{softmax}([\mathbf{q}_n \mathbf{k}_1^\top & \dots & \mathbf{q}_n \mathbf{k}_m^\top]) \end{bmatrix}}_{n \times m} \underbrace{\begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix}}_{m \times d_v} \\
&= \underbrace{\begin{bmatrix} \underbrace{\sum_{i=1}^m \frac{\exp[\mathbf{q}_1 \mathbf{k}_i^\top]}{\sum_j \exp[\mathbf{q}_1 \mathbf{k}_j^\top]} \mathbf{v}_i}_{\text{row vector 1 in } \mathbb{R}^{d_v}} & \vdots \\ \underbrace{\sum_{i=1}^m \frac{\exp[\mathbf{q}_n \mathbf{k}_i^\top]}{\sum_j \exp[\mathbf{q}_n \mathbf{k}_j^\top]} \mathbf{v}_i}_{\text{row vector } n \text{ in } \mathbb{R}^{d_v}} \end{bmatrix}}_{n \times d_v} \tag{9}
\end{aligned}$$

looking at the i -th row of $\mathbf{A}$, i.e., $\mathbf{a}_i$, it can be expressed as a weighted sum of the rows of $\mathbf{V}$.

$$\mathbf{a}_i \mathbf{V} = [a_{i,1} \quad \dots \quad a_{i,m}] \begin{bmatrix} - & \mathbf{v}_1 & - \\ \vdots & \ddots & \vdots \\ - & \mathbf{v}_m & - \end{bmatrix} = \sum_{i=1}^m a_{i,j} \mathbf{v}_i \tag{10}$$

The output $\mathbf{O}$'s dimension has number of row to be the same as $\mathbf{Q}$, and column dimension to be d_v . This means that it has the number of elements of $\mathbf{Q}$, and each element is the same dimension as $\mathbf{v}_i$.

2 Self-attention and Multi-head Attention

The Transformer architecture uses multi-head attention, which is a key component of the Transformer architecture. Generally, input vectors are $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$

When we let:

$$\mathbf{Q} = \mathbf{XW}_Q, \quad \mathbf{K} = \mathbf{XW}_K, \quad \mathbf{V} = \mathbf{XW}_V \tag{11}$$

and let:

$$m = n = T \tag{12}$$

we achieved **self-attention**, in summary:

1. for each head i , linear transform $\mathbf{X}$ into several lower dimensional spaces via projection matrices $\mathbf{W}_i^q, \mathbf{KW}_i^k, \mathbf{VW}_i^v$ to obtain:

$$\mathbf{O}^{(i)} = \text{SDPA}(\mathbf{XW}_Q^{(i)}, \mathbf{XW}_K^{(i)}, \mathbf{XW}_V^{(i)}) \quad (13)$$

2. each iteration i correspond to **one “layer”** of the [“linear”, “Scaled Dot-Product Attention”] on the figure 1
3. then concatenate to produce output:
with multi-head attention, we have multiple $\mathbf{O}_i$ s therefore, we need to mix them together:

$$\mathbf{Y} = \underbrace{\text{concat}(\mathbf{O}_1, \dots, \mathbf{O}_h)}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_O}_{d_{\text{model}} \times d_{\text{model}}} \quad (14)$$

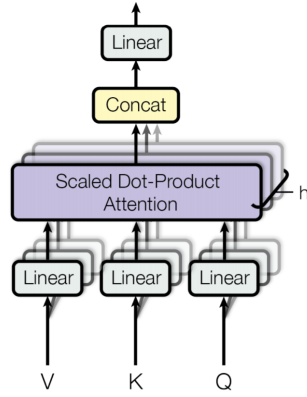


Figure 1: Multi-head Attention

2.1 Detailed implementation

2.1.1 class CausalSelfAttention(nn.Module)

```
class CausalSelfAttention(nn.Module):
```

this class defines the causal self-attention mechanism, the smallest unit of the Transformer architecture. We know that if we have h heads, then when we put multiple heads together, we have:

$$\begin{aligned} \mathbf{Q} &= \underbrace{\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} \underbrace{\mathbf{W}_Q^{(1)}}_{d_{\text{model}} \times d_k} & \dots & \underbrace{\mathbf{W}_Q^{(h)}}_{d_{\text{model}} \times d_k} \end{bmatrix}}_{d_{\text{model}} \times d_{\text{model}}} = \underbrace{\begin{bmatrix} \underbrace{\mathbf{Q}^{(1)}}_{T \times d_k} & \dots & \underbrace{\mathbf{Q}^{(h)}}_{T \times d_k} \end{bmatrix}}_{T \times d_{\text{model}}} \\ \mathbf{K} &= \underbrace{\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} \underbrace{\mathbf{W}_K^{(1)}}_{d_{\text{model}} \times d_k} & \dots & \underbrace{\mathbf{W}_K^{(h)}}_{d_{\text{model}} \times d_k} \end{bmatrix}}_{d_{\text{model}} \times d_{\text{model}}} = \underbrace{\begin{bmatrix} \underbrace{\mathbf{K}^{(1)}}_{T \times d_k} & \dots & \underbrace{\mathbf{K}^{(h)}}_{T \times d_k} \end{bmatrix}}_{T \times d_{\text{model}}} \end{aligned} \quad (15)$$

note that this also corresponds to the B, T, C format (after adding the batch dimension B).
where,

$$\begin{aligned} h &\equiv n_{\text{heads}} \\ d_k &= d_v = d_h \\ d_h \times h &\equiv d_{\text{model}} \end{aligned} \tag{16}$$

and the code has a few definitions:

```
def __init__(self, d_model: int, n_heads: int, dropout: float, max_seq_len: int)
:
    super().__init__()
    assert d_model % n_heads == 0
    self.n_heads = n_heads
    self.head_dim = d_model // n_heads  # per-head dimensionality

    # Single projection that produces query, key, and value in one matmul
    self.qkv = nn.Linear(d_model, 3 * d_model)
    # Final projection after concatenating heads
    self.proj = nn.Linear(d_model, d_model)
    self.attn_dropout = nn.Dropout(dropout)
    self.resid_dropout = nn.Dropout(dropout)
```

Precompute variable mask matrix and store lower-triangular mask of shape $(1, 1, T, T)$, so it can be broadcasted over (B, h, T, T) :

for example, for $T = 5$, we have:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{17}$$

```
mask = torch.tril(torch.ones(max_seq_len, max_seq_len))
mask = mask.view(1, 1, max_seq_len, max_seq_len)
```

2.1.2 Step of each Transformer Layer

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
```

1. At the conclusion of each Transformer layer, we have the output $\mathbf{Y}$ of shape (B, T, C) , where it looks like (assume $B = 1$). See Eq.(29).

$$\underbrace{\begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} \tag{18}$$

The column dimension is $h \times d_v$, which is also the size of the input vector $\mathbf{x}_t$. It can be thought as the “essential” form of the qkv matrix.

Now, we need to perform:

$$\mathbf{Q} = \mathbf{XW}_Q, \quad \mathbf{K} = \mathbf{XW}_K, \quad \mathbf{V} = \mathbf{XW}_V \quad (19)$$

to obtain the following single large matrix before splitting into $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices, we still assume $B = 1$.

$$\underbrace{\begin{bmatrix} -\mathbf{q}_1^{(1)} - & \dots & -\mathbf{q}_1^{(h)} - & -\mathbf{k}_1^{(1)} - & \dots & -\mathbf{k}_1^{(h)} - & -\mathbf{v}_1^{(1)} - & \dots & -\mathbf{v}_1^{(h)} - \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{q}_T^{(1)} - & \dots & -\mathbf{q}_T^{(h)} - & -\mathbf{k}_T^{(1)} - & \dots & -\mathbf{k}_T^{(h)} - & -\mathbf{v}_T^{(1)} - & \dots & -\mathbf{v}_T^{(h)} - \end{bmatrix}}_{T \times (d_{\text{model}} \times 3)} \quad (20)$$

```
B, T, C = x.size() # batch, time, channels
qkv = self.qkv(x)
```

after running the following code, we split the (B, T, C) format into (B, T, h, d_v) , i.e., the big multi-head matrix containing all $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into separate $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices:

$$\underbrace{\begin{bmatrix} - & \mathbf{q}_1^{(1)} & - & \dots & - & \mathbf{q}_1^{(h)} & - \\ \vdots & & & \ddots & & \vdots & \\ - & \mathbf{q}_T^{(1)} & - & \dots & - & \mathbf{q}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}} \quad \underbrace{\begin{bmatrix} - & \mathbf{k}_1^{(1)} & - & \dots & - & \mathbf{k}_1^{(h)} & - \\ \vdots & & & \ddots & & \vdots & \\ - & \mathbf{k}_T^{(1)} & - & \dots & - & \mathbf{k}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}} \quad \underbrace{\begin{bmatrix} - & \mathbf{v}_1^{(1)} & - & \dots & - & \mathbf{v}_1^{(h)} & - \\ \vdots & & & \ddots & & \vdots & \\ - & \mathbf{v}_T^{(1)} & - & \dots & - & \mathbf{v}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}} \quad (21)$$

```
q, k, v = qkv.split(C, dim=2) # each of shape (B, T, C)
```

after running the above code, each $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrix is still having their heads together in a big matrix, so we need to chop them up into separate heads:

2. now we split the QKV single large matrix into each separate $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices:

```
def reshape_heads(t: torch.Tensor) -> torch.Tensor:
    # Reshape to (B, n_heads, T, head_dim) for parallel head
    # computation
    return t.view(B, T, self.n_heads, self.head_dim).transpose(1, 2)

# (B, n_heads, T, head_dim)
q = reshape_heads(q)
k = reshape_heads(k)
v = reshape_heads(v)
```

after running the above code, it has split up $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$, so that each will be in the form of for simplicity let $B = 1$, and we would like to have the structure to look like:

$$\mathbf{Q} = \left[\underbrace{\begin{bmatrix} \begin{bmatrix} - & \mathbf{q}_1 & - \\ - & \dots & - \\ - & \mathbf{q}_T & - \end{bmatrix} & \dots & \begin{bmatrix} - & \mathbf{q}_1 & - \\ - & \dots & - \\ - & \mathbf{q}_T & - \end{bmatrix} \\ \mathbf{Q}_{(1)}: T \times d_k & & \mathbf{Q}_{(h)}: T \times d_k \end{bmatrix}}_{T \times d_{\text{model}}} \right] \quad \mathbf{K} = \left[\underbrace{\begin{bmatrix} \begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_T & - \end{bmatrix} & \dots & \begin{bmatrix} - & \mathbf{k}_1 & - \\ - & \dots & - \\ - & \mathbf{k}_T & - \end{bmatrix} \\ \mathbf{K}_{(1)}: T \times d_k & & \mathbf{K}_{(h)}: T \times d_k \end{bmatrix}}_{T \times d_{\text{model}}} \right] \\ \mathbf{V} = \left[\underbrace{\begin{bmatrix} \begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_T & - \end{bmatrix} & \dots & \begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_T & - \end{bmatrix} \\ \mathbf{V}_{(1)}: T \times d_v & & \mathbf{V}_{(h)}: T \times d_v \end{bmatrix}}_{T \times d_{\text{model}}} \right] \quad (22)$$

where we remind ourselves that:

$$\begin{aligned} h &\equiv n_{\text{heads}} \\ d_h \times h &= d_{\text{model}} \end{aligned} \quad (23)$$

3. we then apply $\mathbf{Q}\mathbf{K}^\top$:

```
att = (q @ k.transpose(-2, -1)) # (B, n_heads, T, T)
```

note the above code does not perform $\mathbf{Q}\mathbf{K}^\top$, after $\mathbf{Q}$ and $\mathbf{K}$ are concatenated from all the heads, but rather it performs $\mathbf{Q}\mathbf{K}^\top$ for each head, i.e., broadcasted matrix multiplication:

$$\begin{bmatrix} \mathbf{Q}_{(1)}\mathbf{K}_{(1)}^\top & \dots & \mathbf{Q}_{(h)}\mathbf{K}_{(h)}^\top \end{bmatrix} \quad (24)$$

4. we then scale by $1/\sqrt{d_h}$:

```
att = att / (self.head_dim ** 0.5)
```

where we obtained:

$$\begin{bmatrix} \frac{\mathbf{Q}_{(1)}\mathbf{K}_{(1)}^\top}{\sqrt{d_h}} & \dots & \frac{\mathbf{Q}_{(h)}\mathbf{K}_{(h)}^\top}{\sqrt{d_h}} \end{bmatrix} = \left[\underbrace{\begin{bmatrix} \frac{\mathbf{q}_1^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_h}} & \dots & \frac{\mathbf{q}_1^{(1)}\mathbf{k}_T^{(1)\top}}{\sqrt{d_h}} \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(1)}\mathbf{k}_1^{(1)\top}}{\sqrt{d_h}} & \dots & \frac{\mathbf{q}_T^{(1)}\mathbf{k}_T^{(1)\top}}{\sqrt{d_h}} \end{bmatrix}}_{\frac{\mathbf{Q}_{(1)}\mathbf{K}_{(1)}^\top}{\sqrt{d_h}}} \quad \dots \quad \underbrace{\begin{bmatrix} \frac{\mathbf{q}_1^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_h}} & \dots & \frac{\mathbf{q}_1^{(h)}\mathbf{k}_T^{(h)\top}}{\sqrt{d_h}} \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(h)}\mathbf{k}_1^{(h)\top}}{\sqrt{d_h}} & \dots & \frac{\mathbf{q}_T^{(h)}\mathbf{k}_T^{(h)\top}}{\sqrt{d_h}} \end{bmatrix}}_{\mathbf{Q}_{(h)}\mathbf{K}_{(h)}^\top} \right] \quad (25)$$

5. we apply a causal mask to the attention matrix, so that each word can only attend to the previous words:

```
# Apply causal mask so position t cannot attend to positions > t
att = att.masked_fill(self.mask[:, :, :T, :T] == 0, float('-inf'))
```

$$\left[\begin{array}{ccc} \frac{\mathbf{q}_1^{(1)} \mathbf{k}_1^{(1)\top}}{\sqrt{d_h}} & \dots & -\infty \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(1)} \mathbf{k}_1^{(1)\top}}{\sqrt{d_h}} & \dots & \frac{\mathbf{q}_T^{(1)} \mathbf{k}_T^{(1)\top}}{\sqrt{d_h}} \end{array} \right] \dots \left[\begin{array}{ccc} \frac{\mathbf{q}_1^{(h)} \mathbf{k}_1^{(h)\top}}{\sqrt{d_h}} & \dots & -\infty \\ \vdots & \ddots & \vdots \\ \frac{\mathbf{q}_T^{(h)} \mathbf{k}_1^{(h)\top}}{\sqrt{d_h}} & \dots & \frac{\mathbf{q}_T^{(h)} \mathbf{k}_T^{(h)\top}}{\sqrt{d_h}} \end{array} \right] \quad (26)$$

6. apply row-wise softmax to the attention matrix:

```
att = F.softmax(att, dim=-1)
att = self.attn_dropout(att)
```

$$\left[\begin{array}{c} \text{softmax} \left(\left[\frac{\mathbf{q}_1^{(1)} \mathbf{k}_1^{(1)\top}}{\sqrt{d_h}} \quad \dots \quad -\infty \right] \right) \\ \vdots \\ \text{softmax} \left(\left[\frac{\mathbf{q}_T^{(1)} \mathbf{k}_1^{(1)\top}}{\sqrt{d_h}} \quad \dots \quad \frac{\mathbf{q}_T^{(1)} \mathbf{k}_T^{(1)\top}}{\sqrt{d_h}} \right] \right) \end{array} \right] \dots \left[\begin{array}{c} \text{softmax} \left(\left[\frac{\mathbf{q}_1^{(h)} \mathbf{k}_1^{(h)\top}}{\sqrt{d_h}} \quad \dots \quad -\infty \right] \right) \\ \vdots \\ \text{softmax} \left(\left[\frac{\mathbf{q}_T^{(h)} \mathbf{k}_1^{(h)\top}}{\sqrt{d_h}} \quad \dots \quad \frac{\mathbf{q}_T^{(h)} \mathbf{k}_T^{(h)\top}}{\sqrt{d_h}} \right] \right) \end{array} \right] \quad (27)$$

$\underbrace{\hspace{10em}}_{\text{softmax} \left(\frac{\mathbf{Q}^{(1)} \mathbf{K}^{(1)\top}}{\sqrt{d_h}} \right)} \quad \underbrace{\hspace{10em}}_{\text{softmax} \left(\frac{\mathbf{Q}^{(h)} \mathbf{K}^{(h)\top}}{\sqrt{d_h}} \right)}$

7. apply weighted sum to the values:

```
# Weighted sum of values
y = att @ v # (B, n_heads, T, head_dim)
```

where we obtained:

$$\underbrace{\left[\underbrace{\text{softmax} \left(\frac{\mathbf{Q}^{(1)} \mathbf{K}^{(1)\top}}{\sqrt{d_h}} \right) \mathbf{V}^{(1)}}_{T \times d_h} \quad \dots \quad \underbrace{\text{softmax} \left(\frac{\mathbf{Q}^{(h)} \mathbf{K}^{(h)\top}}{\sqrt{d_h}} \right) \mathbf{V}^{(h)}}_{T \times d_h} \right]}_{T \times d_{\text{model}}} \quad (28)$$

8. transform the output back to the original dimension B, T, C , where contiguous() is a method that returns a contiguous tensor.

```
# After attention, y has shape (B, n_heads, T, head_dim).
# Each head processed tokens independently.
# We need to bring T back to dim 1 so that each token's
# head outputs sit next to each other: (B, T, n_heads, head_dim).
# Only then can we collapse n_heads * head_dim -> C = d_model.
```

```

y = y.transpose(1, 2)  # swap n_heads <-> T

# transpose() does NOT move data in memory, so view() would fail.
# contiguous() physically reorders the bytes first.
y = y.contiguous()

# Merge n_heads and head_dim into C: (B, T, C)
# This is the "concatenate heads" step from the paper.
y = y.view(B, T, C)

```

know that:

$$\begin{aligned}
\underbrace{\begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} &= \underbrace{\begin{bmatrix} - & \mathbf{o}_1^{(1)} & - \\ & \ddots & \\ - & \mathbf{o}_T^{(1)} & - \end{bmatrix} \cdots \begin{bmatrix} - & \mathbf{o}_1^{(h)} & - \\ & \ddots & \\ - & \mathbf{o}_T^{(h)} & - \end{bmatrix}}_{\substack{T \times d_{\text{model}} \\ \mathbf{O}_{(1)}: T \times d_h \quad \mathbf{O}_{(h)}: T \times d_h}} \underbrace{\mathbf{W}_O}_{d_{\text{model}} \times d_{\text{model}}} \\
&= \underbrace{\begin{bmatrix} - & \mathbf{o}_1^{(1)} & - & \cdots & - & \mathbf{o}_1^{(h)} & - \\ & \vdots & & \ddots & & \vdots & \\ - & \mathbf{o}_T^{(1)} & - & \cdots & - & \mathbf{o}_T^{(h)} & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_O}_{d_{\text{model}} \times d_{\text{model}}}
\end{aligned} \tag{29}$$

each row of $\mathbf{y}_i$ can be determined independently and in parallel.

9. we then perform a final linear projection on the concatenated heads:

```

# Final linear projection on the concatenated heads
y = self.resid_dropout(self.proj(y)) # resid_dropout is the residual
    dropout
return y

```

one can see that the final output is a tensor of shape B, T, d_{model} .

2.1.3 class TransformerBlock(nn.Module)

The following is the code for Pre-norm Transformer block with attention followed by an MLP, where the CausalSelfAttention() is from the previous section:

```

class TransformerBlock(nn.Module):
    def __init__(self, d_model: int, n_heads: int, d_ff: int, dropout: float,
                 max_seq_len: int):
        super().__init__()
        # Pre-layernorm improves optimization stability in autoregressive LMs
        self.ln1 = nn.LayerNorm(d_model)
        self.attn = CausalSelfAttention(d_model, n_heads, dropout, max_seq_len)
        self.ln2 = nn.LayerNorm(d_model)
        # Feed-forward network (position-wise MLP)

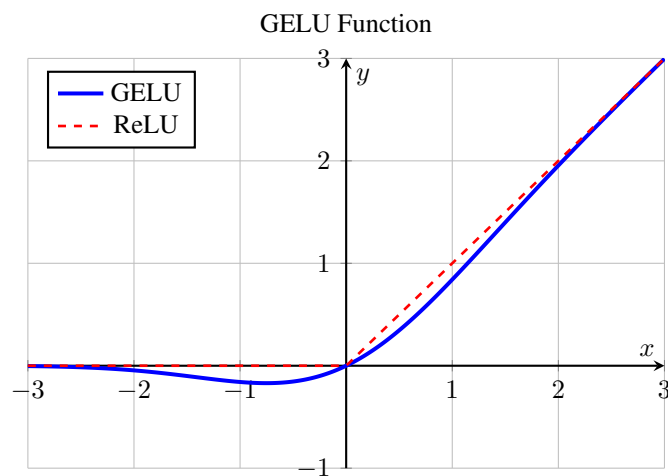
```

```

        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout),
        )
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Attention sub-layer with residual connection
        x = x + self.attn(self.ln1(x))
        # MLP sub-layer with residual connection
        x = x + self.mlp(self.ln2(x))
    return x

```

$x = x + f(x)$ is the residual connection. and $\text{nn.GELU}()$ is the activation function.



The mathematical definition involves the Cumulative Distribution Function (CDF) of the Gaussian (Normal) distribution, often denoted as $\Phi(x)$.

the advantage of GELU over ReLU is that it is smoother and has a better gradient flow.

$$\text{GELU}(x) = x \cdot \Phi(x) \quad (30)$$

In simpler terms:

Output=Input $\times$ (Probability that a normal distribution value is less than Input)

2.1.4 class TransformerLM(nn.Module)

```

class TransformerLM(nn.Module):
    """Causal language model built from Transformer blocks."""

    def __init__(self, config: ModelConfig):
        super().__init__()
        self.config = config

```

```

# Token and positional embeddings are summed to form the input stream
self.token_emb = nn.Embedding(config.vocab_size, config.d_model)
self.pos_emb = nn.Embedding(config.max_seq_len, config.d_model)
self.drop = nn.Dropout(config.dropout)
# Stack of identical Transformer blocks
self.blocks = nn.ModuleList([
    TransformerBlock(config.d_model, config.n_heads, config.d_ff, config
        .dropout, config.max_seq_len)
    for _ in range(config.n_layers)
])
# Final layernorm before projecting to vocabulary logits
self.ln_f = nn.LayerNorm(config.d_model)
# LM head ties to embedding size but does not share weights here
self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)

def forward(self, idx: torch.Tensor, targets: Optional[torch.Tensor] = None)
:
    """Forward pass producing next-token logits and optional cross-entropy
        loss.

    Parameters
    -----
    idx : torch.Tensor
        Token indices of shape (B, T).
    targets : Optional[torch.Tensor]
        Target indices of shape (B, T). If provided, a token-level
        cross-entropy loss is computed and returned.

    Returns
    -----
    Tuple[torch.Tensor, Optional[torch.Tensor]]
        - logits: (B, T, vocab_size)
        - loss: scalar loss if targets is provided, else None
    """
    B, T = idx.shape
    # Ensure sequence length fits within the configured context window
    assert T <= self.config.max_seq_len

    # Positions 0..T-1 for current sequence length
    pos = torch.arange(0, T, dtype=torch.long, device=idx.device).unsqueeze
        (0)

```

this is to create a tensor of shape $(1, T)$ with values from 0 to $T-1$. We call the `unsqueeze(0)` to add a batch dimension, so that it can be broadcasted over the batch dimension when adding to the token embeddings.

$$[[0 \ 1 \ 2 \ \dots \ T-1]] \quad (31)$$

```

# Embed tokens and positions, then apply dropout
x = self.token_emb(idx) + self.pos_emb(pos)
x = self.drop(x)

```

```

# Pass through the stack of Transformer blocks
for blk in self.blocks:
    x = blk(x)

# Final normalization and projection to vocabulary
x = self.ln_f(x)
logits = self.lm_head(x)

# Compute token-level cross-entropy if targets are given
loss = None
if targets is not None:
    loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.
                           view(-1))
return logits, loss

```

In above code, `F.cross_entropy()` function expects inputs expects the shape: (Total_Number_of_Predictions, Number_of_Classes). However, the current shape is $(B, T, \text{vocab_size})$, where `vocab_size` is the number of classes.

Therefore, we transform the logits to have the shape $(B \times T, \text{vocab_size})$, and the targets to have the shape $(B \times T)$.

```

@torch.no_grad()
def generate(self, idx: torch.Tensor, max_new_tokens: int) -> torch.Tensor:
    """Greedy-free sampling generation from the language model.

    Iteratively samples 'max_new_tokens' next tokens by:
    - conditioning on the last 'max_seq_len' tokens
    - computing logits for the next position
    - sampling from the softmax distribution
    - appending the sampled token to the context

    Parameters
    -----
    idx : torch.Tensor
        Initial context token indices of shape (B, T0).
    max_new_tokens : int
        Number of new tokens to sample and append.

    Returns
    -----
    torch.Tensor
        Extended token indices of shape (B, T0 + max_new_tokens).
    """
    self.eval()
    for _ in range(max_new_tokens):
        # Restrict conditioning length to model's maximum context window
        # only the last self.config.max_seq_len tokens are used to condition
        # the generation, as they are the most recent tokens
        idx_cond = idx[:, -self.config.max_seq_len:]
        logits, _ = self(idx_cond)
        # Select logits for the last time step (the next token prediction)
        logits = logits[:, -1, :]

```

```

probs = F.softmax(logits, dim=-1)
# Multinomial sampling allows for non-greedy, stochastic generation
next_id = torch.multinomial(probs, num_samples=1)
# Append sampled token and continue
idx = torch.cat((idx, next_id), dim=1)
return idx

```

2.2 Cross-Attention

If we were to perform cross-attention, $\mathbf{K}$ and $\mathbf{V}$ must be the same thing. For example,

1. Text-to-image Cross-Attention:

$$\begin{aligned} \mathbf{K} \text{ and } \mathbf{V} &\text{ are text embedding} \\ \mathbf{Q} &\text{ is image embedding} \end{aligned} \tag{32}$$

2. Image-to-text Cross-Attention:

$$\begin{aligned} \mathbf{K} \text{ and } \mathbf{V} &\text{ are image embedding} \\ \mathbf{Q} &\text{ is text embedding} \end{aligned} \tag{33}$$

3 Recent Developments

3.1 $\mathbf{K} - \mathbf{V}$ caching

looking at $\mathbf{QK}^\top$, imagine in a self-attention scenario, where the current sentence length is T , and $m = n = T$, then $\mathbf{QK}^\top$ is a square matrix of $T \times T$, where we have:

$$\mathbf{QK}^\top = \begin{bmatrix} \mathbf{q}_1 \mathbf{k}_1^\top & \dots & \mathbf{q}_1 \mathbf{k}_T^\top \\ \vdots & \ddots & \vdots \\ \mathbf{q}_T \mathbf{k}_1^\top & \dots & \mathbf{q}_T \mathbf{k}_T^\top \end{bmatrix} \tag{34}$$

let's say we have a new word w_{T+1} to be added to the sentence, then we have:

$$\begin{bmatrix} \mathbf{q}_1 \mathbf{k}_1^\top & \dots & \mathbf{q}_1 \mathbf{k}_T^\top & \mathbf{q}_1 \mathbf{k}_{T+1}^\top \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{q}_T \mathbf{k}_1^\top & \dots & \mathbf{q}_T \mathbf{k}_T^\top & \mathbf{q}_T \mathbf{k}_{T+1}^\top \\ \mathbf{q}_{T+1} \mathbf{k}_1^\top & \dots & \mathbf{q}_{T+1} \mathbf{k}_T^\top & \mathbf{q}_{T+1} \mathbf{k}_{T+1}^\top \end{bmatrix} = \begin{bmatrix} \mathbf{QK}^\top & \mathbf{Qk}_{T+1}^\top \\ \mathbf{q}_{T+1} \mathbf{K}^\top & \mathbf{q}_{T+1} \mathbf{k}_{T+1}^\top \end{bmatrix} \tag{35}$$

From here, one sees that in generic matrix multiplication, one needs to cache both $\mathbf{K}$ and $\mathbf{Q}$. However, the matrix masks out the upper triangular part of the matrix, i.e., it looks like this:

$$\begin{bmatrix} \mathbf{q}_1 \mathbf{k}_1^\top & - & - & - \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{q}_T \mathbf{k}_1^\top & \dots & \mathbf{q}_T \mathbf{k}_T^\top & - \\ \mathbf{q}_{T+1} \mathbf{k}_1^\top & \dots & \mathbf{q}_{T+1} \mathbf{k}_T^\top & \mathbf{q}_{T+1} \mathbf{k}_{T+1}^\top \end{bmatrix} = \begin{bmatrix} \underbrace{\mathbf{QK}^\top}_{\text{lower triangular}} & - \\ \mathbf{q}_{T+1} \mathbf{K}^\top & \mathbf{q}_{T+1} \mathbf{k}_{T+1}^\top \end{bmatrix} \quad (36)$$

One can see the newly added row (no new column is added) in the above matrix, requires $\mathbf{q}_{T+1}$, $\mathbf{k}_{T+1}$ and $\mathbf{K}$. This means that $\mathbf{K}$ needs to be cached.

Again looking at Eq.(9):

$$\begin{bmatrix} \frac{\exp(\mathbf{q}_1 \mathbf{k}_1^\top)}{\exp(\mathbf{q}_1 \mathbf{k}_1^\top)} \mathbf{v}_1 \\ \frac{\exp(\mathbf{q}_2 \mathbf{k}_1^\top)}{\exp(\mathbf{q}_2 \mathbf{k}_1^\top) + \exp(\mathbf{q}_2 \mathbf{k}_2^\top)} \mathbf{v}_1 + \frac{\exp(\mathbf{q}_2 \mathbf{k}_2^\top)}{\exp(\mathbf{q}_2 \mathbf{k}_1^\top) + \exp(\mathbf{q}_2 \mathbf{k}_2^\top)} \mathbf{v}_2 \\ \frac{\exp(\mathbf{q}_3 \mathbf{k}_1^\top)}{\exp(\mathbf{q}_3 \mathbf{k}_1^\top) + \exp(\mathbf{q}_3 \mathbf{k}_2^\top) + \exp(\mathbf{q}_3 \mathbf{k}_3^\top)} \mathbf{v}_1 + \frac{\exp(\mathbf{q}_3 \mathbf{k}_2^\top)}{\exp(\mathbf{q}_3 \mathbf{k}_1^\top) + \exp(\mathbf{q}_3 \mathbf{k}_2^\top) + \exp(\mathbf{q}_3 \mathbf{k}_3^\top)} \mathbf{v}_2 + \frac{\exp(\mathbf{q}_3 \mathbf{k}_3^\top)}{\exp(\mathbf{q}_3 \mathbf{k}_1^\top) + \exp(\mathbf{q}_3 \mathbf{k}_2^\top) + \exp(\mathbf{q}_3 \mathbf{k}_3^\top)} \mathbf{v}_3 \\ \vdots \\ \sum_{i=1}^T \frac{\exp(\mathbf{q}_T \mathbf{k}_i^\top)}{\sum_{j=1}^T \exp(\mathbf{q}_T \mathbf{k}_j^\top)} \mathbf{v}_i \\ \sum_{i=1}^{T+1} \frac{\exp(\mathbf{q}_{T+1} \mathbf{k}_i^\top)}{\sum_{j=1}^{T+1} \exp(\mathbf{q}_{T+1} \mathbf{k}_j^\top)} \mathbf{v}_i \end{bmatrix} \quad (37)$$

the last row in the above matrix contains the newly added row, where all $\mathbf{v}_i$ are needed for computation and hence they need to be cached.

3.2 Multi-head latent attention

Multi-head latent attention (MLA) was introduced in [4]. Say we have sentence of length T , and we have each word represented by a $d = 7168$ dimensional vector. Then we can have:

$$\begin{aligned} \mathbf{C}_{K,V} &= \mathbf{XW}_{\downarrow K,V} \\ &= \underbrace{\begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} \mathbf{w}_{1,1} & \dots & \mathbf{w}_{1,576} \\ \vdots & \ddots & \vdots \\ \mathbf{w}_{7168,1} & \dots & \mathbf{w}_{7168,576} \end{bmatrix}}_{d_{\text{model}} \times 576} \end{aligned} \quad (38)$$

where $\mathbf{C}_{K,V}$ is the “common” part of the latent representation of the sentence for both $\mathbf{K}$ and $\mathbf{V}$. They are also the same for all heads. This is why this quantity is referred to directly, instead of having $\mathbf{XW}_{\downarrow K,V}$.

normally, in the case of multi-head self-attention, one should have: $\mathbf{Q}^{(i)} = \mathbf{XW}_Q^{(i)}$, $\mathbf{K}^{(i)} = \mathbf{XW}_K^{(i)}$, $\mathbf{V}^{(i)} = \mathbf{XW}_V^{(i)}$, but in here, we are having:

traditional		multi-head latent attention	
$\mathbf{Q}^{(i)} = \mathbf{X} \mathbf{W}_Q^{(i)}$	$\rightarrow$	$\underbrace{\mathbf{Q}^{(i)}}_{T \times 128} = \mathbf{X} \underbrace{\mathbf{W}_Q^{(i)}}_{7168 \times 128}$	
$\mathbf{K}^{(i)} = \mathbf{X} \mathbf{W}_K^{(i)}$	$\rightarrow$	$\underbrace{\mathbf{K}^{(i)}}_{T \times 128} = \mathbf{X} \underbrace{\mathbf{W}_{\downarrow K, V}}_{d_{\text{model}} \times 576} \underbrace{\mathbf{W}_{\uparrow K}^{(i)}}_{576 \times 128} = \underbrace{\mathbf{C}_{K, V}}_{T \times 576} \underbrace{\mathbf{W}_{\uparrow K}^{(i)}}_{576 \times 128}$	(39)
$\mathbf{V}^{(i)} = \mathbf{X} \mathbf{W}_V^{(i)}$	$\rightarrow$	$\underbrace{\mathbf{V}^{(i)}}_{T \times 128} = \mathbf{X} \underbrace{\mathbf{W}_{\downarrow K, V}}_{d_{\text{model}} \times 576} \underbrace{\mathbf{W}_{\uparrow V}^{(i)}}_{576 \times 128} = \underbrace{\mathbf{C}_{K, V}}_{T \times 576} \underbrace{\mathbf{W}_{\uparrow V}^{(i)}}_{576 \times 128}$	

making $\mathbf{W}_{\downarrow K, V}$ to be common for all heads for both $\mathbf{K}$ and $\mathbf{V}$. Therefore only $\mathbf{W}_{\uparrow K}^{(i)}$ and $\mathbf{W}_{\uparrow V}^{(i)}$ need to be trained per head for both $\mathbf{K}$ and $\mathbf{V}$. This means that:

$$\begin{aligned} \mathbf{W}_K^{(i)} &\rightarrow \mathbf{W}_{\downarrow K, V} \mathbf{W}_{\uparrow K}^{(i)} \\ \mathbf{K}^{(i)} &\rightarrow \mathbf{X} \mathbf{W}_{\downarrow K, V} \mathbf{W}_{\uparrow K}^{(i)} \end{aligned} \quad (40)$$

substitute:

$$\begin{aligned} \mathbf{Q}^{(i)} \mathbf{K}^{(i)\top} &= \mathbf{X} \mathbf{W}_Q^{(i)} (\mathbf{X} \mathbf{W}_{\downarrow K, V} \mathbf{W}_{\uparrow K}^{(i)})^\top \\ &= \mathbf{X} \mathbf{W}_Q^{(i)} ((\mathbf{X} \mathbf{W}_{\downarrow K, V}) \mathbf{W}_{\uparrow K}^{(i)})^\top \\ &= \underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{(\mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top})}_{d_{\text{model}} \times 576} \underbrace{(\mathbf{X} \mathbf{W}_{\downarrow K, V})^\top}_{576 \times T} \\ &= \underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{(\mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top})}_{d_{\text{model}} \times 576} \underbrace{\mathbf{C}_{K, V}^\top}_{576 \times T} \end{aligned} \quad (41)$$

Remember that although we have different $\mathbf{W}_{\uparrow K}^{(i)}$ for each head, $(\mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top})$ can be computed first only once (for every head i) before they multiple with $\mathbf{X}$ and $\mathbf{C}_{K, V}^\top$ (and they are shared for all heads) during inference time. This means that we do not need to store $\mathbf{W}_Q^{(i)}$ and $\mathbf{W}_{\uparrow K}^{(i)}$ separately, but rather we can store $\mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top}$ altogether. It's like having a modified $\mathbf{W}_Q^{(i)}$ for each head.

*Note: some people may be confused about “storing” $(\mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top})$. This in fact, has nothing to do with the KV cache. The KV cache stores $\mathbf{C}_{KV}$ (the compressed latent) across time steps — $\mathbf{Q}$ vectors are still computed fresh for each new token using whatever weight matrix you have (merged or not). $\mathbf{W}_Q^{(i)}$ here refers to a model **parameter** (always in memory), not a cached activation.*

In terms of caching, assumed that at the length of T words, we have cached $\mathbf{X}(\mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top})$ and $\mathbf{x} \mathbf{W}_{\downarrow K, V}$. Then after adding a new word, i.e., $\mathbf{X}_{T+1}$, we can just compute one more row of the matrix $\mathbf{x}_{T+1} \mathbf{W}_Q^{(i)} \mathbf{W}_{\uparrow K}^{(i)\top}$ and $\mathbf{x}_{T+1} \mathbf{W}_{\downarrow K, V} = \mathbf{c}_{K, V}^{T+1}$ and append it to the cache, i.e.,

$$\begin{aligned}
& \begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix} [(\mathbf{W}_Q \mathbf{W}_{\uparrow K}^\top)] = \text{cache} \\
\Rightarrow & \begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \\ - & \mathbf{x}_{T+1} & - \end{bmatrix} [(\mathbf{W}_Q \mathbf{W}_{\uparrow K}^\top)] = \begin{bmatrix} \text{cache} \\ \mathbf{x}_{T+1} (\mathbf{W}_Q \mathbf{W}_{\uparrow K}^\top) \end{bmatrix}
\end{aligned} \tag{42}$$

the same applies to $\mathbf{XW}_{\downarrow K, V}$, i.e.,

$$\begin{aligned}
& \begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \end{bmatrix} [\mathbf{W}_{\downarrow K, V}] = \text{cache} \\
\Rightarrow & \begin{bmatrix} - & \mathbf{x}_1 & - \\ - & \dots & - \\ - & \mathbf{x}_T & - \\ - & \mathbf{x}_{T+1} & - \end{bmatrix} [\mathbf{W}_{\downarrow K, V}] = \begin{bmatrix} \text{cache} \\ \mathbf{x}_{T+1} \mathbf{W}_{\downarrow K, V} \end{bmatrix}
\end{aligned} \tag{43}$$

Now, let's look at the output of the multi-head attention:

$$\begin{aligned}
\mathbf{Y} &= \underbrace{\begin{bmatrix} - & \mathbf{y}_1 & - \\ & \vdots & \\ - & \mathbf{y}_T & - \end{bmatrix}}_{T \times d_{\text{model}}} = \underbrace{\text{concat}(\mathbf{O}_1, \dots, \mathbf{O}_h)}_{T \times (d_h \times h)} \underbrace{\mathbf{W}_O}_{d_{\text{model}} \times d_{\text{model}}} \\
&= \begin{bmatrix} \text{softmax}\left(\frac{\mathbf{Q}_{(1)} \mathbf{K}_{(1)}^\top}{\sqrt{d_h}}\right) \mathbf{V}_{(1)} & \dots & \text{softmax}\left(\frac{\mathbf{Q}_{(h)} \mathbf{K}_{(h)}^\top}{\sqrt{d_h}}\right) \mathbf{V}_{(h)} \end{bmatrix} \mathbf{W}_O \\
&= \begin{bmatrix} \mathbf{O}_{(1)} \mathbf{V}_{(1)} & \dots & \mathbf{O}_{(h)} \mathbf{V}_{(h)} \end{bmatrix} \mathbf{W}_O \\
&= \begin{bmatrix} \underbrace{\mathbf{O}_{(1)}}_{T \times d_h} & \dots & \underbrace{\mathbf{O}_{(h)}}_{T \times d_h} \end{bmatrix} \begin{bmatrix} \underbrace{\mathbf{V}_{(1)}}_{d_h \times d_h} & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \underbrace{\mathbf{V}_{(h)}}_{d_h \times d_h} \end{bmatrix} \begin{bmatrix} \underbrace{\mathbf{W}_O^{1:d_h, 1:d_{\text{model}}}}_{d_h \times d_{\text{model}}} \\ \vdots \\ \underbrace{\mathbf{W}_O^{d_h \times (h-1) + 1:d_{\text{model}}, 1:d_{\text{model}}}}_{d_h \times d_{\text{model}}} \end{bmatrix} \\
&= \underbrace{\begin{bmatrix} \mathbf{O}_{(1)} & \dots & \mathbf{O}_{(h)} \end{bmatrix}}_{T \times d_{\text{model}}} \underbrace{\begin{bmatrix} \underbrace{\mathbf{V}_{(1)} \mathbf{W}_O^{1:d_h, 1:d_{\text{model}}}}_{d_h \times d_{\text{model}}} \\ \vdots \\ \underbrace{\mathbf{V}_{(h)} \mathbf{W}_O^{d_h \times (h-1) + 1:d_{\text{model}}, 1:d_{\text{model}}}}_{d_h \times d_{\text{model}}} \end{bmatrix}}_{d_{\text{model}} \times d_{\text{model}}}
\end{aligned} \tag{44}$$

now that we replaced $\mathbf{W}_V$ with $\mathbf{W}_{\downarrow K, V} \mathbf{W}_{\uparrow V}$, we can have:

$$\mathbf{V} = \underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_{\downarrow K, V}}_{d_{\text{model}} \times 576} \underbrace{\mathbf{W}_{\uparrow V}}_{576 \times d_h} \quad (45)$$

therefore, we can have each $\mathbf{V}$, for example $\mathbf{V}_{(1)}$ as

$$\mathbf{V}_{(1)} \mathbf{W}_O^{1:d_h, 1:d_{\text{model}}} = \underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_{\downarrow K, V}}_{d_{\text{model}} \times 576} \underbrace{\mathbf{W}_{\uparrow V}}_{576 \times d_h} \underbrace{\mathbf{W}_O^{1:d_h, 1:d_{\text{model}}}}_{d_h \times d_{\text{model}}} \quad (46)$$

we already computed $\mathbf{XW}_{\downarrow K, V}$ as $\mathbf{L}_{K, V}$, and we can also pre-compute $\mathbf{W}_{\uparrow V} \mathbf{W}_O^{1:d_h, 1:d_{\text{model}}}$ as single matrix, instead of storing $\mathbf{W}_{\uparrow V}$ and $\mathbf{W}_O^{1:d_h, 1:d_{\text{model}}}$ separately.

Finally, we have $\mathbf{Y}$ as the output of the multi-head attention layer.

3.3 Rotary Embedding RoPE

From the paper, [5], here is the RoPE Python code implementation:

```
class RotaryEmbedding(nn.Module):
    def __init__(self, head_dim=64, rope_theta=100000):
        super().__init__()
        self.head_dim = head_dim
        self.rope_theta = rope_theta

    def forward(self, x):
        # x shape: [batch, num_heads, seq_len, head_dim]

        # Compute frequencies
        i = torch.arange(0, self.head_dim, 2)
        freqs = 1.0 / (self.rope_theta ** (i / self.head_dim))
```

$$i = [0 \quad 2 \quad 4 \quad \dots \quad d_v - 2]$$

$$\text{freqs} = \underbrace{\left[\frac{1}{\theta^{0/d_h}} \quad \frac{1}{\theta^{2/d_h}} \quad \frac{1}{\theta^{4/d_h}} \quad \dots \quad \frac{1}{\theta^{(d_h-2)/d_h}} \right]}_{1 \times (d_h/2)} \quad (47)$$

to avoid notational clutter, we write θ^{t/d_h} as θ^t for short.

$$\text{freqs} = \underbrace{\left[\frac{1}{\theta^0} \quad \frac{1}{\theta^2} \quad \frac{1}{\theta^4} \quad \dots \quad \frac{1}{\theta^{d_h-2}} \right]}_{1 \times (d_h/2)} \quad (48)$$

this is in the decreasing frequency order.

```
# Compute angles for all positions
pos = torch.arange(x.shape[2])
angles = torch.outer(pos, freqs)
```

since x shape is (batch, num_heads, T, head_dim), therefore $x.shape[2]$ is the seq_len, i.e., T . Therefore, we have:

$$\begin{aligned} \text{angles} &= \begin{bmatrix} 0 \times \frac{1}{\theta^0} & 0 \times \frac{1}{\theta^2} & 0 \times \frac{1}{\theta^4} & \dots & 0 \times \frac{1}{\theta^{d_h-2}} \\ 1 \times \frac{1}{\theta^0} & 1 \times \frac{1}{\theta^2} & 1 \times \frac{1}{\theta^4} & \dots & 1 \times \frac{1}{\theta^{d_h-2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (T-1) \times \frac{1}{\theta^0} & (T-1) \times \frac{1}{\theta^2} & (T-1) \times \frac{1}{\theta^4} & \dots & (T-1) \times \frac{1}{\theta^{d_h-2}} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} \frac{0}{\theta^0} & \frac{0}{\theta^2} & \frac{0}{\theta^4} & \dots & \frac{0}{\theta^{d_h-2}} \\ \frac{1}{\theta^0} & \frac{1}{\theta^2} & \frac{1}{\theta^4} & \dots & \frac{1}{\theta^{d_h-2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{T-1}{\theta^0} & \frac{T-1}{\theta^2} & \frac{T-1}{\theta^4} & \dots & \frac{T-1}{\theta^{d_h-2}} \end{bmatrix}}_{T \times (d_h/2)} \end{aligned} \quad (49)$$

for each t^{th} token, there is a corresponding frequency for each feature, i.e., $\frac{t}{\theta^0}, \frac{t}{\theta^2}, \frac{t}{\theta^4}, \dots, \frac{t}{\theta^{d_h-2}}$

```
# Precompute sin/cos
cos = torch.cos(angles)
sin = torch.sin(angles)
cos = cos.unsqueeze(0).unsqueeze(0)
sin = sin.unsqueeze(0).unsqueeze(0)
```

squeeze / unsqueeze example

```
x = torch.tensor([1., 2., 3.]) -- shape: (3,)
# unsqueeze: insert a new dim of size 1
x.unsqueeze(0) → shape (1, 3) [[1, 2, 3]]
x.unsqueeze(1) → shape (3, 1) [[1],[2],[3]]
# chaining: add two leading dims
x.unsqueeze(0).unsqueeze(0) → shape (1, 1, 3)
# squeeze: remove all dims of size 1
x.unsqueeze(0).squeeze() → shape (3,) [1, 2, 3]
```

Why here? cos/sin start as shape (T, head_dim). Two unsqueeze(0) calls add a batch and a heads dim, giving (1, 1, T, head_dim) so they broadcast over (B, n_heads, T, head_dim) without copying data.

$$\cos = \left[\begin{bmatrix} \cos\left(\frac{0}{\theta^0}\right) & \cos\left(\frac{0}{\theta^2}\right) & \cos\left(\frac{0}{\theta^4}\right) & \dots & \cos\left(\frac{0}{\theta^{d_h-2}}\right) \\ \cos\left(\frac{1}{\theta^0}\right) & \cos\left(\frac{1}{\theta^2}\right) & \cos\left(\frac{1}{\theta^4}\right) & \dots & \cos\left(\frac{1}{\theta^{d_h-2}}\right) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \cos\left(\frac{T-1}{\theta^0}\right) & \cos\left(\frac{T-1}{\theta^2}\right) & \cos\left(\frac{T-1}{\theta^4}\right) & \dots & \cos\left(\frac{T-1}{\theta^{d_h-2}}\right) \end{bmatrix} \right] \quad (50)$$

$$\sin = \left[\begin{bmatrix} \sin(\frac{0}{\theta^0}) & \sin(\frac{0}{\theta^2}) & \sin(\frac{0}{\theta^4}) & \dots & \sin(\frac{0}{\theta^{d_h-2}}) \\ \sin(\frac{1}{\theta^0}) & \sin(\frac{1}{\theta^2}) & \sin(\frac{1}{\theta^4}) & \dots & \sin(\frac{1}{\theta^{d_h-2}}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sin(\frac{T-1}{\theta^0}) & \sin(\frac{T-1}{\theta^2}) & \sin(\frac{T-1}{\theta^4}) & \dots & \sin(\frac{T-1}{\theta^{d_h-2}}) \end{bmatrix} \right] \quad (51)$$

```
# Split into pairs
x_even = x[..., 0::2]
x_odd = x[..., 1::2]
```

Now we adhere to programming language convention, making the index run from 0 to -1 (i.e. 0-indexed). For example, if x is $\mathbf{Q}$, then we have:

$$\mathbf{Q}_{\text{even}} = \begin{bmatrix} q_{0,0} & q_{0,2} & q_{0,4} & \dots & q_{0,d_h-2} \\ q_{1,0} & q_{1,2} & q_{1,4} & \dots & q_{1,d_h-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{T-1,0} & q_{T-1,2} & q_{T-1,4} & \dots & q_{T-1,d_h-2} \end{bmatrix} \quad \mathbf{Q}_{\text{odd}} = \begin{bmatrix} q_{0,1} & q_{0,3} & q_{0,5} & \dots & q_{0,d_h-1} \\ q_{1,1} & q_{1,3} & q_{1,5} & \dots & q_{1,d_h-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{T-1,1} & q_{T-1,3} & q_{T-1,5} & \dots & q_{T-1,d_h-1} \end{bmatrix} \quad (52)$$

```
# Apply rotation (broadcasting cos/sin across batch and heads)
x_even_new = x_even * cos - x_odd * sin
x_odd_new = x_even * sin + x_odd * cos
```

$$\mathbf{Q}'_{\text{even}} = \underbrace{\begin{bmatrix} q_{0,0}\cos(\frac{0}{\theta^0}) - q_{0,1}\sin(\frac{0}{\theta^0}) & q_{0,2}\cos(\frac{0}{\theta^2}) - q_{0,3}\sin(\frac{0}{\theta^2}) & \dots & q_{0,d_h-2}\cos(\frac{0}{\theta^{d_h-2}}) - q_{0,d_h-1}\sin(\frac{0}{\theta^{d_h-2}}) \\ q_{1,0}\cos(\frac{1}{\theta^0}) - q_{1,1}\sin(\frac{1}{\theta^0}) & \boxed{q_{1,2}\cos(\frac{1}{\theta^2}) - q_{1,3}\sin(\frac{1}{\theta^2})} & \dots & q_{1,d_h-2}\cos(\frac{1}{\theta^{d_h-2}}) - q_{1,d_h-1}\sin(\frac{1}{\theta^{d_h-2}}) \\ \vdots & \vdots & \ddots & \vdots \\ q_{T-2,0}\cos(\frac{T-2}{\theta^0}) - q_{T-2,1}\sin(\frac{T-2}{\theta^0}) & q_{T-2,2}\cos(\frac{T-2}{\theta^2}) - q_{T-2,3}\sin(\frac{T-2}{\theta^2}) & \dots & q_{T-2,d_h-2}\cos(\frac{T-2}{\theta^{d_h-2}}) - q_{T-2,d_h-1}\sin(\frac{T-2}{\theta^{d_h-2}}) \\ q_{T-1,0}\cos(\frac{T-1}{\theta^0}) - q_{T-1,1}\sin(\frac{T-1}{\theta^0}) & q_{T-1,2}\cos(\frac{T-1}{\theta^2}) - q_{T-1,3}\sin(\frac{T-1}{\theta^2}) & \dots & q_{T-1,d_h-2}\cos(\frac{T-1}{\theta^{d_h-2}}) - q_{T-1,d_h-1}\sin(\frac{T-1}{\theta^{d_h-2}}) \end{bmatrix}}_{T \times (d_h/2)} \quad (53)$$

each $(t, j)^{\text{th}}$ element is of the form, for $t = 0, 1, \dots, T-1$ and $j = 0, 1, \dots, (d_h/2) - 1$:

$$q'_{t,j} = q_{t,2j}\cos(\frac{t}{\theta^j}) - q_{t,2j+1}\sin(\frac{t}{\theta^j}) \quad (54)$$

$$\mathbf{Q}'_{\text{odd}} = \underbrace{\begin{bmatrix} q_{0,0}\sin(\frac{0}{\theta^0}) + q_{0,1}\cos(\frac{0}{\theta^0}) & q_{0,2}\sin(\frac{0}{\theta^2}) + q_{0,3}\cos(\frac{0}{\theta^2}) & \dots & q_{0,d_h-2}\sin(\frac{0}{\theta^{d_h-2}}) + q_{0,d_h-1}\cos(\frac{0}{\theta^{d_h-2}}) \\ q_{1,0}\sin(\frac{1}{\theta^0}) + q_{1,1}\cos(\frac{1}{\theta^0}) & \boxed{q_{1,2}\sin(\frac{1}{\theta^2}) + q_{1,3}\cos(\frac{1}{\theta^2})} & \dots & q_{1,d_h-2}\sin(\frac{1}{\theta^{d_h-2}}) + q_{1,d_h-1}\cos(\frac{1}{\theta^{d_h-2}}) \\ \vdots & \vdots & \ddots & \vdots \\ q_{T-2,0}\sin(\frac{T-2}{\theta^0}) + q_{T-2,1}\cos(\frac{T-2}{\theta^0}) & q_{T-2,2}\sin(\frac{T-2}{\theta^2}) + q_{T-2,3}\cos(\frac{T-2}{\theta^2}) & \dots & q_{T-2,d_h-2}\sin(\frac{T-2}{\theta^{d_h-2}}) + q_{T-2,d_h-1}\cos(\frac{T-2}{\theta^{d_h-2}}) \\ q_{T-1,0}\sin(\frac{T-1}{\theta^0}) + q_{T-1,1}\cos(\frac{T-1}{\theta^0}) & q_{T-1,2}\sin(\frac{T-1}{\theta^2}) + q_{T-1,3}\cos(\frac{T-1}{\theta^2}) & \dots & q_{T-1,d_h-2}\sin(\frac{T-1}{\theta^{d_h-2}}) + q_{T-1,d_h-1}\cos(\frac{T-1}{\theta^{d_h-2}}) \end{bmatrix}}_{T \times (d_h/2)} \quad (55)$$

each $(t, j)^{\text{th}}$ element is of the form, for $t = 0, 1, \dots, T-1$ and $j = 0, 1, \dots, (d_h/2) - 1$:

$$q'_{t,j} = q_{t,2j}\sin(\frac{t}{\theta^j}) + q_{t,2j+1}\cos(\frac{t}{\theta^j}) \quad (56)$$

One can see that the addition of the corresponding even and odd elements will be a $2-d$ rotation by each corresponding θ with different frequency. For example, the addition of the two blue terms will be a $2-d$ rotation by θ^2 with frequency 1, i.e., by $\frac{1}{\theta^2}$ radians, and the addition of the two red terms will be a $2-d$ rotation by $\frac{1}{\theta^2}$ radians.

$$\begin{bmatrix} q'_{1,2} & q'_{1,3} \end{bmatrix} = \begin{bmatrix} q_{1,2} & q_{1,3} \end{bmatrix} \begin{bmatrix} \cos(\frac{1}{\theta^2}) & \sin(\frac{1}{\theta^2}) \\ -\sin(\frac{1}{\theta^2}) & \cos(\frac{1}{\theta^2}) \end{bmatrix} = \begin{bmatrix} q_{1,2}\cos(\frac{1}{\theta^2}) - q_{1,3}\sin(\frac{1}{\theta^2}) & q_{1,2}\sin(\frac{1}{\theta^2}) + q_{1,3}\cos(\frac{1}{\theta^2}) \end{bmatrix} \quad (57)$$

or more generally, for each token t , and each feature block $(2i, 2i+1)$ (i.e., the i^{th} feature block after we stack them together in the next step), we have:

$$\begin{aligned} \begin{bmatrix} q'_{t,2i} & q'_{t,2i+1} \end{bmatrix} &= \begin{bmatrix} q_{t,2i} & q_{t,2i+1} \end{bmatrix} \begin{bmatrix} \cos(\frac{t}{\theta^{2i}}) & \sin(\frac{t}{\theta^{2i}}) \\ -\sin(\frac{t}{\theta^{2i}}) & \cos(\frac{t}{\theta^{2i}}) \end{bmatrix} \\ &= \begin{bmatrix} q_{t,2i}\cos(\frac{t}{\theta^{2i}}) - q_{t,2i+1}\sin(\frac{t}{\theta^{2i}}) & q_{t,2i}\sin(\frac{t}{\theta^{2i}}) + q_{t,2i+1}\cos(\frac{t}{\theta^{2i}}) \end{bmatrix} \end{aligned} \quad (58)$$

```
# Interleave back
x_rotated = torch.stack([x_even_new, x_odd_new], dim=-1)
x_rotated = x_rotated.flatten(-2, -1)
return x_rotated
```

How `stack + flatten` reassemble the rotated tensor

After applying the rotation, we have two separate tensors:

- $\mathbf{Q}'_{even}$: the rotated even-indexed dimensions, shape $(T, d_v/2)$
- $\mathbf{Q}'_{odd}$: the rotated odd-indexed dimensions, shape $(T, d_v/2)$

We need to interleave them back into a single tensor of shape (T, d_v) . This is done in two steps:

Step 1 — `torch.stack([x_even_new, x_odd_new], dim=-1)`

`stack` creates a *new* dimension at position `dim=-1`, pairing each even element with its odd counterpart:

$$(T, d_v/2) \xrightarrow{\text{stack}} (T, d_v/2, 2)$$

The last dimension of size 2 holds the pair $[\mathbf{Q}'_{even,i}, \mathbf{Q}'_{odd,i}]$ for each position i .

Step 2 — `.flatten(-2, -1)`

`flatten` merges the last two dimensions $(d_v/2, 2)$ into one, effectively interleaving even and odd elements:

$$(T, d_v/2, 2) \xrightarrow{\text{flatten}} (T, d_v)$$

The result is a tensor of shape (T, d_v) — the same shape as the original $\mathbf{Q}$, now with rotary embeddings applied.

3.4 Decoupled RoPE

Decoupled RoPE is used in [4] to apply rotary position embeddings only to a portion of the query/key dimensions while keeping the rest for the compressed latent. We have, given h heads:

$$\mathbf{Y}^{(i)} = \text{softmax} \left(\frac{1}{\sqrt{d_k}} \underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_Q^{(i)}}_{d_{\text{model}} \times 128} \underbrace{\mathbf{W}_K^{(i)\top}}_{128 \times 576} \underbrace{(\mathbf{X} \mathbf{W}_{\downarrow K, V})^\top}_{576 \times T} \right) \underbrace{\mathbf{X} \mathbf{W}_{\downarrow K, V}}_{T \times 576} \underbrace{\mathbf{W}_V^{(i)}}_{576 \times d_v} \underbrace{\mathbf{W}_O^{(i-1) \times d_v + 1:i \times d_v, 1:d_{\text{model}}}}_{d_v \times d_{\text{model}}}$$

$$\mathbf{Y} = \sum_{i=1}^h \mathbf{Y}^{(i)} \quad (59)$$

when we have multiple heads, the output is the same size, except there are multiple of $\mathbf{Y}^{(i)}$ added together.

Here is something about RoPE, let's take the terms inside the softmax function, and remove the $\frac{1}{\sqrt{d_k}}$ term, we have:

$$\underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_Q^{(i)}}_{d_{\text{model}} \times 128} \underbrace{\mathbf{W}_K^{(i)\top}}_{128 \times 576} \underbrace{(\mathbf{X} \mathbf{W}_{\downarrow K, V})^\top}_{576 \times T} \quad (60)$$

Since RoPE is applied to the $\mathbf{Q}$ and $\mathbf{K}$ matrices, where there is not a single RoPE matrix for all the tokens, i.e., we do not have something like:

$$\underbrace{\mathbf{X}}_{T \times d_{\text{model}}} \underbrace{\mathbf{W}_Q^{(i)}}_{d_{\text{model}} \times 128} \mathbf{R}^{(i)} \mathbf{R}^{(i)T} \underbrace{\mathbf{W}_{\uparrow K}^{(i)T}}_{128 \times 576} \underbrace{(\mathbf{X} \mathbf{W}_{\downarrow K, V})^T}_{576 \times T} \quad (61)$$

For this reason, we may consider a decoupled RoPE mechanism, where we have two separate part of each row of $\mathbf{Q}$ and $\mathbf{K}$ matrices, i.e., we break up $\mathbf{q}_t$ and $\mathbf{k}_t$ into two parts, one is the part that does not involve RoPE (of a larger dimension), and the other is the part that involves RoPE (of a smaller dimension).

For each head i , and each token t , we have, note that the following, we changed from right multiplication to left multiplication:

$$\mathbf{k}_t^C = \mathbf{W}_{\uparrow K}^{(i)} \mathbf{c}_{K, V}^t, \quad \mathbf{v}_t^C = \mathbf{W}_{\uparrow V}^{(i)} \mathbf{c}_{K, V}^t \quad (62)$$

for the t -th token, we have the following concatenation of outputs:

$$\mathbf{o}_t = [\mathbf{o}_{t,1} \quad \mathbf{o}_{t,2} \quad \dots \quad \mathbf{o}_{t,h}] \quad (63)$$

which is the weighted sum of $\mathbf{v}_j^C$ for $j = 1, 2, \dots, t$, responding to a specific query $\mathbf{q}_t$, we now drop the head index i for simplicity:

$$\mathbf{o}_t = \sum_{j=1}^t \text{Softmax} \left(\frac{\mathbf{q}_t^T \mathbf{k}_j}{\sqrt{d_h + d_h^R}} \right) \mathbf{v}_j^C \quad (64)$$

by letting: $\mathbf{c}_t^Q = \mathbf{W}_{\downarrow Q} \mathbf{x}_t$, $\mathbf{c}_t^Q \in \mathbb{R}^{d_C}$, we have:

$$\begin{aligned} \mathbf{q}_t^{(i)} &= \begin{bmatrix} \underbrace{(\mathbf{W}_{\uparrow Q}^{(i)} \mathbf{W}_{\downarrow Q} \mathbf{x}_t)}_{d_h \times d_C} & \underbrace{\text{RoPE}(\mathbf{W}_{(R)Q}^{(i)} \mathbf{W}_{\downarrow Q} \mathbf{x}_t)}_{d_h^R \times d_C} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} \mathbf{W}_{\uparrow Q}^{(i)} \mathbf{c}_t^Q & \text{RoPE}(\mathbf{W}_{(R)Q}^{(i)} \mathbf{c}_t^Q) \end{bmatrix}}_{(d_h + d_h^R) \times 1} \end{aligned} \quad (65)$$

you see that $\mathbf{q}_t^{(i)}$ has two parts, each projecting the hidden state $\mathbf{x}_t$ into a different space by matrices $\mathbf{W}_{\uparrow Q}^{(i)}$ and $\mathbf{W}_{(R)Q}^{(i)}$. Similarly, by letting: $\mathbf{c}_{K, V}^t = \mathbf{W}_{\downarrow K, V} \mathbf{x}_t$, $\mathbf{c}_{K, V}^t \in \mathbb{R}^{d_C}$, we have:

$$\begin{aligned} \mathbf{k}_s^{(i)} &= \begin{bmatrix} \underbrace{(\mathbf{W}_{\uparrow K}^{(i)} \mathbf{W}_{\downarrow K, V} \mathbf{x}_s)}_{d_h \times d_C} & \underbrace{\text{RoPE}(\mathbf{W}_{(R)K}^{(i)} \mathbf{W}_{\downarrow K, V} \mathbf{x}_s)}_{d_h^R \times d_C} \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} \mathbf{W}_{\uparrow K}^{(i)} \mathbf{c}_{K, V}^s & \text{RoPE}(\mathbf{W}_{(R)K}^{(i)} \mathbf{c}_{K, V}^s) \end{bmatrix}}_{(d_h + d_h^R) \times 1} \end{aligned} \quad (66)$$

now we look at the value $\mathbf{v}_t^C$, we have:

$$\begin{aligned}
\mathbf{v}_t^C &= \underbrace{\mathbf{W}_{\uparrow V}^{(i)}}_{d_h \times d_C} \underbrace{\mathbf{W}_{\downarrow K, V}}_{d_C \times d_{\text{model}}} \underbrace{\mathbf{x}_t}_{d_{\text{model}} \times 1} \\
&= \underbrace{\mathbf{W}_{\uparrow V}^{(i)}}_{d_h \times d_C} \underbrace{\mathbf{c}_{K, V}^t}_{d_C \times 1}
\end{aligned} \tag{67}$$

during inference, we have the following steps:

- for the t -th token, we compute:

$$\mathbf{q}_t^{(i)} = \left[\mathbf{W}_{\uparrow Q}^{(i)} \mathbf{c}_Q^t \quad \text{RoPE}(\mathbf{W}_{(R)Q}^{(i)} \mathbf{c}_Q^t) \right] \tag{68}$$

here, nothing needs to be cached.

- and for each of the s -th token, $\forall s \in \{1, 2, \dots, t-1\}$, we have:

$$\mathbf{k}_s^{(i)} = \left[\mathbf{W}_{\uparrow K}^{(i)} \mathbf{c}_{K, V}^s \quad \text{RoPE}(\mathbf{W}_{(R)K}^{(i)} \mathbf{c}_{K, V}^s) \right] \tag{69}$$

the things in the cache are the red parts, i.e.,

- $\mathbf{c}_{K, V}^t = \mathbf{W}_{\downarrow K, V} \mathbf{h}_t$
- $\mathbf{k}_{(R)t}^{(i)} = \text{RoPE}(\mathbf{W}_{(R)K}^{(i)} \mathbf{c}_{K, V}^t)$

3.5 Lightning Indexer

As the number of tokens t increases, it became too computationally expensive to compute all the attention weights for each token $j \leq t$ with the current token t , therefore, we need a **lightening indexer** to handle this.

Deepseek [4] computes the current query $\mathbf{q}_t$ and the previous key $\mathbf{k}_s$ to decide which query-key pair to use for the attention calculation. Now since we have H^I to indicate the number of index head. (in the paper, $H^I = 64$). Therefore, we have a set of query vectors per head, i.e., $\{\mathbf{q}_{t,j}^I\}_{j=1}^{H^I}$, and a set of key vectors per head, i.e., $\{\mathbf{k}_{s,j}^I\}_{j=1}^{H^I}$. However, to save computation, the key vector is shared across all the index heads, i.e., $\mathbf{k}_{s,j}^I$ is the same for all $j = 1, 2, \dots, H^I$. So we can simply use $\mathbf{k}_s^I$ for all the index heads.

$$I_{t,s} = \sum_{j=1}^{H^I} w_{t,j}^I \cdot \text{ReLU}(\mathbf{q}_{t,j}^I \cdot \mathbf{k}_s^I) \tag{70}$$

which can be thought as the average attention (across all the index heads) between the current query $\mathbf{q}_t$ and the previous key $\mathbf{k}_s$.

- Note that H^I is smaller than number of heads used in the main attention, i.e., h . $H^I = 64$ in the paper.

- The dimensions of both $\mathbf{q}_{t,j}^I$ and $\mathbf{k}_s^I$ are 128.
- $w_{t,j}^I$: A learnable scalar weight. This is particularly important as it's learnable parameter!
- Using ReLU activation instead of softmax is beneficial for throughput optimization (FP8 implementation).

Since lower precision is used for both $\mathbf{q}_{t,j}^I$ and $\mathbf{k}_s^I$, therefore, an orthogonal projection is used to project the query and key vectors to prevent them to be too concentrated at some values, for example, $\mathbf{q}_{t,j}^I = [0.01 \ 0.98 \ \dots \ 0.01]^\top$.

Knowing that by applying an orthogonal projection $\mathbf{U}$, between $\mathbf{q}$ and $\mathbf{k}$, their dot product is preserved:

$$(\mathbf{U}\mathbf{q})^\top(\mathbf{U}\mathbf{k}) = \mathbf{q}^\top\mathbf{k} \quad (71)$$

but the projection $\mathbf{U}$ can make both $\mathbf{q}$ and $\mathbf{k}$ to have the values more spread out separately.

4 Flash Attention

this is from the paper [6]:

4.1 traditional softmax algorithm

The algorithm for computing softmax is as follows, given there is a logit set $\{x_i\}_{i=1}^N$, where each $x_i \in \mathbb{R}$. We need three separate loops to compute the softmax output:

1. loop 1: compute the maximum value in the logit set:

$$\begin{aligned} m_0 &= -\infty \\ \text{for } 1 \leq i \leq N \text{ do} \\ &\quad m_i = \max(m_{i-1}, x_i) \\ \text{return } m_N \end{aligned} \quad (72)$$

2. loop 2: compute the sum of the exponential of the logits after we obtained m_N which is the maximum value in the logit set:

$$\begin{aligned} d_0 &= 0 \\ \text{for } 1 \leq i \leq N \text{ do} \\ &\quad d_i = d_{i-1} + \exp^{x_i - m_N} \\ \text{return } d_N \end{aligned} \quad (73)$$

3. loop 3: normalize the logits:

$$\begin{aligned} \text{for } 1 \leq i \leq N \text{ do} \\ &\quad a_i = \frac{\exp^{x_i - m_N}}{d_N} \\ \text{return } \{a_i\}_{i=1}^N \end{aligned} \quad (74)$$

4.2 online softmax algorithm

now looking at the second loop, in essence, at the i -th iteration, it is computing the sum of the exponential of the logits from 1 to i :

$$\begin{aligned}
 d_i &= \sum_{j=1}^i \exp^{x_j - m_N} \\
 &= \underbrace{\sum_{j=1}^{i-1} \exp^{x_j - m_N}}_{d_{i-1}} + \exp^{x_i - m_N} \\
 &= d_{i-1} + \exp^{x_i - m_N}
 \end{aligned} \tag{75}$$

with the initial condition $d_0 = 0$, the above can also be expressed as:

$$d_0 = 0 \quad \text{and} \quad d_i = d_{i-1} + \exp^{x_i - m_N} \tag{76}$$

However, the above cannot be combined with the first loop, because of the m_N term. Therefore, instead of subtracting a “global” maximum value m_N , we subtract a “intermediate” maximum value m_i up to the i -th iteration:

$$d'_i = \sum_{j=1}^i \exp^{x_j - m_i} \tag{77}$$

at the completion of the entire loop, we have $d'_N = d_N$, even though $d_i \neq d'_i \quad \forall i \neq N$ in general. However, if we do the same trick as Eq.(77), however, the first term is not equal to $d'_{i-1} = \sum_{j=1}^{i-1} \exp^{x_j - m_{i-1}}$, i.e., you cannot write the sum as, (a term only contain sum up to $i - 1$ terms d_{i-1}) + (a term only contain x_i term).

$$d'_i = \underbrace{\sum_{j=1}^{i-1} \exp^{x_j - m_i}}_{\neq d'_{i-1}} + \exp^{x_i - m_i} \tag{78}$$

but we can make it so, by adding and subtracting a dummy variable:

$$\begin{aligned}
d'_i &= \sum_{j=1}^{i-1} \exp^{x_j - m_i} + \exp^{x_i - m_i} \\
&= \left(\sum_{j=1}^{i-1} \exp^{x_j - m_i} \right) \underbrace{\exp^{m_{i-1}} \exp^{-m_{i-1}}}_{\text{dummy variable}} + \exp^{x_i - m_i} \quad \text{perform } \times \exp^{m_{i-1}} \exp^{-m_{i-1}} \\
&= \left(\sum_{j=1}^{i-1} \exp^{x_j - \textcolor{red}{m}_{i-1}} \right) \exp^{m_{i-1}} \exp^{-\textcolor{blue}{m}_i} + \exp^{x_i - m_i} \quad \text{sawp the red and blue terms} \\
&= \underbrace{\left(\sum_{j=1}^{i-1} \exp^{x_j - \textcolor{red}{m}_{i-1}} \right)}_{d'_{i-1}} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i} \\
&= d'_{i-1} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i}
\end{aligned} \tag{79}$$

therefore, we can have the algorithm to be:

1. loop 1: compute the maximum value in the logit set:

$$\begin{aligned}
&m_0 = -\infty \\
&d_0 = 0 \\
&\mathbf{for } 1 \leq i \leq N \mathbf{ do} \\
&\quad m_i = \max(m_{i-1}, x_i) \\
&\quad d_i = d_{i-1} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i} \\
&\mathbf{return } m_N, d_N
\end{aligned} \tag{80}$$

we also changed $d'_i \rightarrow d_i$ to simplify the notation.

2. loop 2: same as loop 3 in the traditional softmax algorithm:

$$\begin{aligned}
&\mathbf{for } 1 \leq i \leq N \mathbf{ do} \\
&\quad a_i = \frac{\exp^{x_i - m_N}}{d_N} \\
&\mathbf{return } \{a_i\}_{i=1}^N
\end{aligned} \tag{81}$$

since we are not perform “reduce” operation, such as sum, max, etc., as in here, we compute each a_i individually, we cannot combine it with the first loop.

4.3 online attention

looking at the single $\mathbf{q}$ row matrix form, where we have (we omit $1/\sqrt{d_k}$ for notational clarity):

$$\begin{aligned}
\mathbf{qK}^\top &= [\mathbf{qk}_1^\top \quad \dots \quad \mathbf{qk}_m^\top] \\
\Rightarrow \mathbf{o} &= \text{softmax}(\mathbf{qK}^\top) \mathbf{V} \\
&= [\text{softmax}([\mathbf{qk}_1^\top \quad \dots \quad \mathbf{qk}_m^\top])] \begin{bmatrix} - & \mathbf{v}_1 & - \\ - & \dots & - \\ - & \mathbf{v}_m & - \end{bmatrix} \\
&= \sum_{i=1}^N \frac{\exp[\mathbf{qk}_i^\top]}{\sum_j \exp[\mathbf{qk}_j^\top]} \mathbf{v}_i \\
&= \sum_{i=1}^N a_i \mathbf{v}_i
\end{aligned} \tag{82}$$

taking a particular row of the above, $\mathbf{x} = \mathbf{qK}^\top$, where $x_i = \mathbf{qk}_i^\top$ is an element of $\mathbf{x}$:

1. loop 1: compute the maximum value in the logit set:

$$\begin{aligned}
m_0 &= -\infty \\
d_0 &= 0 \\
\text{for } 1 \leq i \leq N \text{ do} \\
&\quad x_i = q_i \mathbf{k}_i^\top \\
&\quad m_i = \max(m_{i-1}, x_i) \\
&\quad d_i = d_{i-1} \exp^{m_{i-1}-m_i} + \exp^{x_i-m_i} \\
\text{return } m_N, d_N
\end{aligned} \tag{83}$$

2. loop 2: adding $\mathbf{o} = \sum_{i=1}^N a_i \mathbf{v}_i$ to the second loop:

$$\begin{aligned}
\mathbf{o}_0 &= \mathbf{0} \\
\text{for } 1 \leq i \leq N \text{ do} \\
&\quad a_i = \frac{\exp^{x_i-m_N}}{d_N} \\
&\quad \mathbf{o}_i = \mathbf{o}_{i-1} + a_i \mathbf{v}_i
\end{aligned} \tag{84}$$

$$\begin{aligned}
\mathbf{o}_0 &= \mathbf{0} \\
\text{for } 1 \leq i \leq N \text{ do} \\
&\quad \mathbf{o}_i = \mathbf{o}_{i-1} + \frac{\exp^{x_i-m_N}}{d_N} \mathbf{v}_i
\end{aligned}$$

note that we can apply the same trick as Eq.(77) to the second loop:

$$\begin{aligned}
\mathbf{o}_N &= \sum_{j=1}^N \frac{\exp^{x_j-m_N}}{d'_N} \mathbf{v}_j \\
\mathbf{o}'_i &= \sum_{j=1}^i \frac{\exp^{x_j-m_i}}{d'_i} \mathbf{v}_j
\end{aligned} \tag{85}$$

where we have $\mathbf{o}_N = \mathbf{o}'_N$. Apply the same track as Eq.(77), we have:

$$\begin{aligned}
\mathbf{o}'_i &= \sum_{j=1}^i \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j \\
&= \left(\sum_{j=1}^{i-1} \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j \right) + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i \\
&= \underbrace{\left(\sum_{j=1}^{i-1} \frac{\exp^{x_j - m_i}}{d'_i} \mathbf{v}_j \right)}_{\neq \mathbf{o}_{i-1}} \frac{d'_{i-1}}{d'_i} \frac{\exp^{m_{i-1}}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i \\
&= \underbrace{\left(\sum_{j=1}^{i-1} \frac{\exp^{x_j - m_{i-1}}}{d'_{i-1}} \mathbf{v}_j \right)}_{= \mathbf{o}_{i-1}} \frac{d'_{i-1}}{d'_i} \frac{\exp^{m_i}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i \\
&= \mathbf{o}_{i-1} \frac{d'_{i-1}}{d'_i} \frac{\exp^{m_i}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d'_i} \mathbf{v}_i
\end{aligned} \tag{86}$$

4.3.1 final algorithm

everything comes down to just a single loop:

$$\begin{aligned}
m_0 &= -\infty \\
d_0 &= 0 \\
\mathbf{o}_0 &= \mathbf{0} \\
\text{for } 1 \leq i \leq N \text{ do} \\
&\quad x_i = q_i \mathbf{k}_i^\top \\
&\quad m_i = \max(m_{i-1}, x_i) \\
&\quad d_i = d_{i-1} \exp^{m_{i-1} - m_i} + \exp^{x_i - m_i} \\
&\quad \mathbf{o}_i = \mathbf{o}_{i-1} \frac{d_{i-1}}{d_i} \frac{\exp^{m_i}}{\exp^{m_{i-1}}} + \frac{\exp^{x_i - m_i}}{d_i} \mathbf{v}_i \\
\text{return } \mathbf{o}_N
\end{aligned} \tag{87}$$

again, we changed $d'_i \rightarrow d_i$ and $\mathbf{o}'_i \rightarrow \mathbf{o}_i$ to simplify the notation.

5 Manifold-Constrained Hyper-Connections (mHC)

imagine that at layer l , we have multiple streams of input vectors, $\mathbf{x}_l^{(1)}, \mathbf{x}_l^{(2)}, \mathbf{x}_l^{(3)}, \mathbf{x}_l^{(4)}$, we can aggregate them together to form a single vector, via a weighted sum, where each $\mathbf{x}_l^{(i)} \in \mathbb{R}^d$:

$$\mathbf{X}_l = \begin{bmatrix} - & \mathbf{x}_l^{(1)} & - \\ - & \mathbf{x}_l^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{x}_l^{(n)} & - \end{bmatrix} \quad (88)$$

1. Aggregation Streams:

now we need a parameter vector to aggregate the streams:

$$\mathcal{H}_l^{\text{pre}} = \begin{bmatrix} \alpha_l^{(1)} & \alpha_l^{(2)} & \dots & \alpha_l^{(n)} \end{bmatrix} \in \mathbb{R}^{1 \times n} \quad (89)$$

We can have the pre-aggregated vector:

$$\begin{aligned} \mathbf{x}_l^{\text{pre}} &= \mathcal{H}_l^{\text{pre}} \mathbf{X}_l \\ &= \begin{bmatrix} \alpha_l^{(1)} & \alpha_l^{(2)} & \dots & \alpha_l^{(n)} \end{bmatrix} \begin{bmatrix} - & \mathbf{x}_l^{(1)} & - \\ - & \mathbf{x}_l^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & \mathbf{x}_l^{(n)} & - \end{bmatrix} \end{aligned} \quad (90)$$

where $\mathbf{x}_l^{\text{pre}} \in \mathbb{R}^{1 \times d}$, i.e., a row vector. this is the same as computing the sum of the streams, i.e.,

$$\mathbf{x}_l^{\text{pre}} = \sum_{i=1}^n \alpha_l^{(i)} \mathbf{x}_l^{(i)} \quad (91)$$

2. neural network:

$$\mathcal{F}_{W_l}(\mathbf{x}_l^{\text{pre}}) = [\bar{z}_{l,1} \quad \bar{z}_{l,2} \quad \dots \quad \bar{z}_{l,d}] \quad (92)$$

this is the main neural network that we need, and its still only performed on a single stream, as it is the most expensive operation.

3. Expand Streams:

We need another set of parameters to expand the stream:

$$\mathcal{H}_l^{\text{post}} = \begin{bmatrix} \beta_l^{(1)} & \beta_l^{(2)} & \dots & \beta_l^{(n)} \end{bmatrix}^{\top} \in \mathbb{R}^{n \times 1} \quad (93)$$

then the expanded streams are:

$$\begin{aligned}
\mathbf{Z}_l &= \mathcal{H}_l^{\text{post}} \mathcal{F}_{W_l}(\mathbf{x}_l^{\text{pre}}) \\
&= \begin{bmatrix} \beta_l^{(1)} \\ \beta_l^{(2)} \\ \vdots \\ \beta_l^{(n)} \end{bmatrix} \begin{bmatrix} \bar{z}_{l,1} & \bar{z}_{l,2} & \dots & \bar{z}_{l,d} \end{bmatrix}
\end{aligned} \tag{94}$$

4. Mix Streams:

here we have another set of parameters to mix the streams $\mathcal{H}_l^{\text{res}} \in \mathbb{R}^{n \times n}$:

$$\begin{aligned}
\mathbf{H}_l &= \mathcal{H}_l^{\text{res}} \mathbf{X}_l \\
\text{where } \mathbf{H}_l &\in \mathbb{R}^{n \times d}
\end{aligned} \tag{95}$$

then the mixed streams are:

$$\mathbf{x}_{l+1} = \mathbf{H}_l + \mathbf{Z}_l \tag{96}$$

Putting them all together, we have:

$$\mathbf{x}_{l+1} = \mathcal{H}_l^{\text{res}} \mathbf{x}_l + \mathcal{H}_l^{\text{post}}(\mathcal{F}_{W_l}(\mathcal{H}_l^{\text{pre}} \mathbf{x}_l)) \tag{97}$$

Comparison with the traditional residual network

The classic residual (skip-connection) update is:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}_{W_l}(\mathbf{x}_l)$$

Setting $\mathcal{H}_l^{\text{pre}} = \mathcal{H}_l^{\text{post}} = \mathbf{I}$ and $\mathcal{H}_l^{\text{res}} = \mathbf{I}$ recovers the standard residual block exactly.

5.1 issue with the implementation

looking at the residual connection, we have:

$$\begin{aligned}
\mathbf{x}_L &= \underbrace{\mathbf{x}_{L-1}}_{\text{don't apply recursion}} + \mathcal{F}_{W_{L-1}}(\mathbf{x}_{L-1}) \\
&= \underbrace{\mathbf{x}_{L-2} + \mathcal{F}_{W_{L-2}}(\mathbf{x}_{L-2})}_{\text{don't apply recursion}} + \mathcal{F}_{W_{L-1}}(\mathbf{x}_{L-1}) \\
&= \mathbf{x}_{L-3} + \mathcal{F}_{W_{L-3}}(\mathbf{x}_{L-3}) + \mathcal{F}_{W_{L-2}}(\mathbf{x}_{L-2}) + \mathcal{F}_{W_{L-1}}(\mathbf{x}_{L-1}) \\
&\quad \vdots \\
&= \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}_{W_i}(\mathbf{x}_i).
\end{aligned} \tag{98}$$

You see that the first term is the identity mapping, which prevents gradient vanishing and exploding, i.e., $\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = 1 + \text{some terms}$.

now we turn our attention to the hyper-connections:

$$\begin{aligned}
\mathbf{x}_L &= \mathcal{H}_{L-1}^{\text{res}} \mathbf{x}_{L-1} + \mathcal{H}_{L-1}^{\text{post}}(\mathcal{F}_{W_{L-1}}(\mathcal{H}_{L-1}^{\text{pre}} \mathbf{x}_{L-1})) \\
&= \mathcal{H}_{L-1}^{\text{res}} \underbrace{(\mathcal{H}_{L-2}^{\text{res}} \mathbf{x}_{L-2} + \mathcal{H}_{L-2}^{\text{post}}(\mathcal{F}_{W_{L-2}}(\mathcal{H}_{L-2}^{\text{pre}} \mathbf{x}_{L-2})))}_{\mathbf{x}_{L-1}} + \mathcal{H}_{L-1}^{\text{post}}(\mathcal{F}_{W_{L-1}}(\mathcal{H}_{L-1}^{\text{pre}} \mathbf{x}_{L-1})) \\
&= \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{res}} \mathbf{x}_{L-2} + \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{post}}(\mathcal{F}_{W_{L-2}}(\mathcal{H}_{L-2}^{\text{pre}} \mathbf{x}_{L-2})) + \mathcal{H}_{L-1}^{\text{post}}(\mathcal{F}_{W_{L-1}}(\mathcal{H}_{L-1}^{\text{pre}} \mathbf{x}_{L-1})) \\
&= \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{res}} \underbrace{(\mathcal{H}_{L-3}^{\text{res}} \mathbf{x}_{L-3} + \mathcal{H}_{L-3}^{\text{post}}(\mathcal{F}_{W_{L-3}}(\mathcal{H}_{L-3}^{\text{pre}} \mathbf{x}_{L-3})))}_{\mathbf{x}_{L-2}} \\
&\quad + \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{post}}(\mathcal{F}_{W_{L-2}}(\mathcal{H}_{L-2}^{\text{pre}} \mathbf{x}_{L-2})) + \mathcal{H}_{L-1}^{\text{post}}(\mathcal{F}_{W_{L-1}}(\mathcal{H}_{L-1}^{\text{pre}} \mathbf{x}_{L-1})) \\
&= \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{res}} \mathcal{H}_{L-3}^{\text{res}} \mathbf{x}_{L-3} \\
&\quad + \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{res}} \mathcal{H}_{L-3}^{\text{post}}(\mathcal{F}_{W_{L-3}}(\mathcal{H}_{L-3}^{\text{pre}} \mathbf{x}_{L-3})) + \mathcal{H}_{L-1}^{\text{res}} \mathcal{H}_{L-2}^{\text{post}}(\mathcal{F}_{W_{L-2}}(\mathcal{H}_{L-2}^{\text{pre}} \mathbf{x}_{L-2})) + \mathcal{H}_{L-1}^{\text{post}}(\mathcal{F}_{W_{L-1}}(\mathcal{H}_{L-1}^{\text{pre}} \mathbf{x}_{L-1})) \\
&\quad \vdots \\
&= \left(\prod_{i=1}^{L-l} \mathcal{H}_{L-i}^{\text{res}} \right) \mathbf{x}_l + \sum_{i=1}^{L-1} \left(\prod_{j=1}^{L-1-i} \mathcal{H}_{L-j}^{\text{res}} \right) \mathcal{H}_i^{\text{post}} \mathcal{F}_{W_i}(\mathcal{H}_i^{\text{pre}} \mathbf{x}_i).
\end{aligned} \tag{99}$$

5.2 need $\mathcal{H}_l^{\text{res}}$ to be doubly stochastic

we no longer have the identity mapping, i.e., $\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} \neq 1$, as we now have:

$$\left(\prod_{i=1}^{L-l} \mathcal{H}_{L-i}^{\text{res}} \right) \mathbf{x}_l \tag{100}$$

However, if we can make sure that at each layer, $\mathcal{H}_l^{\text{res}}$ is a doubly stochastic matrix, i.e., $\mathcal{H}_l^{\text{res}} \in \mathbb{R}^{n \times n}$ and then the product of two doubly stochastic matrices is also a doubly stochastic matrix, one can be shown in the followig example of matrix multiplication:

$$\mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \tag{101}$$

you see that the row sum:

$$a_{11}b_{11} + a_{12}b_{21} + a_{11}b_{12} + a_{12}b_{22} = a_{11}(b_{11} + b_{12}) + a_{12}(b_{21} + b_{22}) = 1 \tag{102}$$

likewise, the column sum:

$$a_{11}b_{11} + a_{12}b_{21} + a_{21}b_{11} + a_{22}b_{21} = b_{11}(a_{11} + a_{21}) + b_{21}(a_{12} + a_{22}) = 1 \tag{103}$$

in addition, the $\|\mathcal{H}_l^{\text{res}}\|_2 = 1$, i.e., the maximum eigenvalue of $\mathcal{H}_l^{\text{res}}$ is 1.

This doubly stochastic property can be achieved by Sinkhorn-Knopp algorithm, which is to iteratively normalize the rows and columns of the matrix to sum to 1.

For DeepSeek, instead of learning the parameters $\mathcal{H}_l^{\text{res}}$, they learn the parameters $\mathcal{H}_l^{\text{pre}}$ and $\mathcal{H}_l^{\text{post}}$, we instead learn the parameters from the following, i.e., $\alpha_l^{\text{res}}, \mathbf{b}_l^{\text{res}}, \phi_l^{\text{res}}$:

$$\begin{aligned}
\tilde{\mathbf{x}}_l &= \text{RMSNorm}(\mathbf{x}_l) \\
\mathcal{H}_l^{\text{pre}} &= \sigma(\alpha_l^{\text{pre}} \cdot (\tilde{\mathbf{x}}_l \phi_l^{\text{pre}}) + \mathbf{b}_l^{\text{pre}}) \\
\mathcal{H}_l^{\text{post}} &= 2 \sigma(\alpha_l^{\text{post}} \cdot (\tilde{\mathbf{x}}_l \phi_l^{\text{post}}) + \mathbf{b}_l^{\text{post}}) \\
\mathcal{H}_l^{\text{res}} &= \text{Sinkhorn} - \text{Knopp}(\alpha_l^{\text{res}} \cdot \text{Mat}(\tilde{\mathbf{x}}_l \phi_l^{\text{res}}) + \mathbf{b}_l^{\text{res}})
\end{aligned} \tag{104}$$

References

- [1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [2] Ilya Sutskever, Oriol Vinyals, and Quoc V Le, “Sequence to sequence learning with neural networks,” *Advances in neural information processing systems*, vol. 27, 2014.
- [3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint arXiv:1409.0473*, 2014.
- [4] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al., “Deepseek-v3. 2: Pushing the frontier of open large language models,” *arXiv preprint arXiv:2512.02556*, 2025.
- [5] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu, “Roformer: Enhanced transformer with rotary position embedding,” *Neurocomputing*, vol. 568, pp. 127063, 2024.
- [6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” *Advances in neural information processing systems*, vol. 35, pp. 16344–16359, 2022.