Attention Mechanisms

Jonah Ramponi

March 2024

Contents

1 Background and Notation **2**

2 Introduction To Self Attention **4**
2.1 Scaled Dot Product Self Attention 4
2.2 Multi Head Self Attention 6

3 Memory Bandwidth Reduction via Key and Value **7**
3.1 Multi Query Attention 7
3.2 Grouped Query Attention 7
3.3 Conversions from Multi Head Attention 7

4 Adaptations to the Attention Matrix **8**
4.1 Sliding Window Attention 8
4.2 Sparse Attention 10

5 Inference **12**
5.1 The KV Cache 12
5.2 Flash Attention 13

1 Background and Notation

Suppose you give an LLM the input

input: *"What is the capital of France?"*

The first thing the LLM will do is split this input into tokens. A token is just some combinations of characters. You can see an example of the tokenization outputs for the question below.

"*What* *is* *the* *capital* *of* *France* *?*"[1]

In this example we have ($n = 7$) tokens. Importantly, from our model's point of view, our input size is defined by the number of tokens instead of words. A numerical representation (vector representation) of each token is now found. Finding this vector representation is called producing an embedding of the token. The token " What" might get tokenized as follows

$$\text{tokenizer}(\textit{What}) \rightarrow \begin{bmatrix} -0.4159 \\ -0.5147 \\ 0.5690 \\ \vdots \\ -0.2577 \\ 0.5710 \end{bmatrix} \tag{1}$$

The length of each of our embeddings, these vector outputs of our tokenizer, are the same regardless of the number of characters in our token. Let us denote this length d_{model}. So after we embed each token in our input sequence with our tokenizer we are left with

$$\textbf{Output after tokenization:} \begin{bmatrix} -0.415 \\ -0.514 \\ 0.569 \\ \vdots \\ -0.257 \\ 0.571 \end{bmatrix} \begin{bmatrix} -0.130 \\ -0.464 \\ 0.23 \\ \vdots \\ -0.154 \\ 0.192 \end{bmatrix}, \ldots, \begin{bmatrix} 0.127 \\ 0.453 \\ 0.110 \\ \vdots \\ -0.155 \\ 0.484 \end{bmatrix}$$

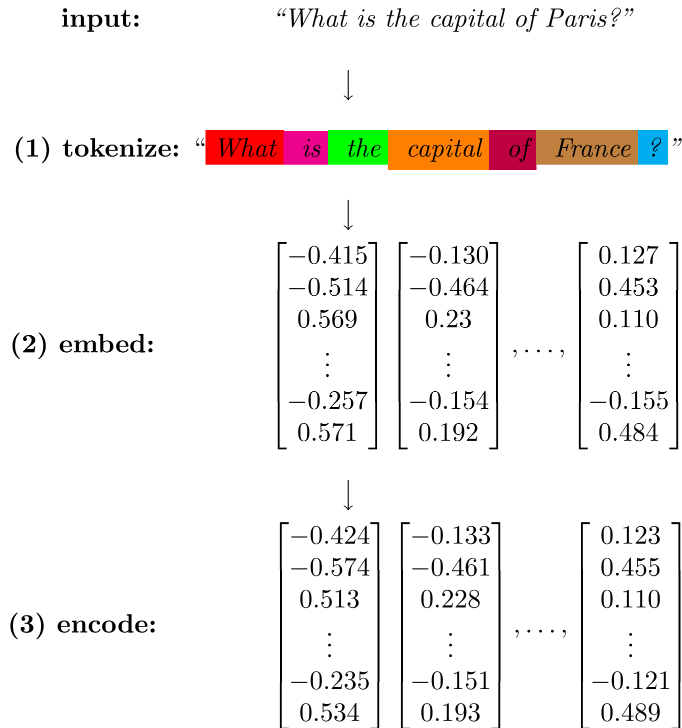
This output is now passed through a *positional encoder*. Broadly, this is useful to provide the model with information about the position of words or tokens within a sequence. You might wonder why we need to positionally encode each token. What does it even mean to positionally encode something? Why can't we just use the index of the item? These questions are for another document.

The only thing that matters for now, is that each of our numerical representations (vectors) are slightly altered. For the numerical representation of the token " What" that we get from our embedding model, it might look something like:

$$\text{positional encoder}\left(\begin{bmatrix} -0.415 \\ -0.514 \\ \vdots \\ -0.257 \\ 0.571 \end{bmatrix}\right) = \begin{bmatrix} -0.424 \\ -0.574 \\ \vdots \\ -0.235 \\ 0.534 \end{bmatrix} \tag{2}$$

Importantly, the positional encoder does not alter the length of our vector, d_{model}. It simply tweaks the values slightly. So far, our transformations look like:

[1] This tokenization was produced using cl100k_base, the tokenizer used in GPT-3.5-turbo and GPT-4.



We're now very close to being able to introduce attention. One last thing remains, at this point we will transform the output of our positional encoding to a matrix M as follows

$$M = \begin{pmatrix} -0.424 & -0.574 & 0.513 & \dots & -0.235 & 0.534 \\ -0.133 & 0.461 & 0.228 & \dots & -0.151 & 0.193 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0.123 & 0.455 & 0.110 & \dots & -0.121 & 0.489 \end{pmatrix} = \text{positional encoding} \begin{pmatrix} \text{What} \\ \text{is} \\ \vdots \\ ? \end{pmatrix} \tag{3}$$

The top row is the first vector output of our positional encoding. The second row is the second, and so on. If we had n tokens in our input sequence, then matrix M would have n rows. The dimensions of M are as follows

$$M = \Big(\text{number of tokens in input} \times \text{length of embedding}\Big) = \Big(n \times d_{\text{model}}\Big). \tag{4}$$

2 Introduction To Self Attention

At a high level, self-attention aims to evaluate the importance of each element in a sequence with respect to all other elements and use this to compute a representation of the sequence. All it really does is compute a weighted average of input vectors to produce output vectors. Mathematically, for an input sequence of vectors $x = (\vec{x}_1, \ldots, \vec{x}_n)$ it will return some sequence of vectors, $y = (\vec{y}_1, \ldots, \vec{y}_m)$ such that

$$y_i = \sum_{j=1}^{n} w_{ij} \cdot x_j, \qquad \forall 1 \leq i \leq m. \tag{5}$$

for some mapping w_{ij}. The challenge is in figuring out how we should define our mapping w_{ij}. Let's look at the first way w_{ij} was defined, introduced in Attention is All You Need [7].

2.1 Scaled Dot Product Self Attention

To compute scaled dot product self attention, we will use the matrix M with rows corresponding to the positionally encoded vectors. M has dimensions $(n \times d_{\text{model}})$.

We begin by producing query, key and value matrices, analogous to how a search engine maps a user query to relevant items in its database[2]. We will make 3 copies of our matrix M. These become the matrices Q, K and V. Each of these has dimension $(n \times d_{\text{model}})$. We let d_k denote the dimensions of the keys, which in this case is d_{model}. We are ready to define attention as

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V. \tag{6}$$

```
1 def attention(Q, K, V):
2     dk = K.size(-1)
3     scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(dk)
4     attn_weights = torch.nn.functional.softmax(scores, dim=-1)
5     return torch.matmul(attn_weights, V)
```

Our matrix QK^T of dimension $(n \times d_{\text{model}}) \times (n \times d_{\text{model}})^T = (n \times n)$. After we re-scale by $\sqrt{d_k}$, this matrix is referred to as the *attention matrix*.

Why do we divide by $\sqrt{d_k}$? This was introduced to counteract the effect of having the dot products grow large in magnitude for large dimensional inputs $d_k >> 1$. In cases where the dot product grew large in size, it was suspect suspected that application of the softmax function was returning extremely small gradients which in turn lead to the vanishing gradients problem.

We multiply the softmax of the attention matrix with each row of V. This re-scales each row of the output matrix to sum to one. The equation for softmax applied to a matrix X is as follows[3]

$$\text{softmax}(X)_{ij} = \frac{e^{X_{ij}}}{\sum_{k=1}^{n} e^{X_{ik}}}. \tag{7}$$

```
1 def softmax(X):
2     exp_X = torch.exp(X)
3     denom = exp_X.sum(dim=-1, keepdim=True)
4     return exp_X / denom
```

[2] This stack exchange post contains some great insight into the idea behind the Q, K and V matrices.
[3] A version of softmax with better stability is discussed in Section (5.2).

Why use softmax? The dot product of Q and K^T gives us a value anywhere between negative and positive infinity. Application of softmax ensures our outputs are more stable. Otherwise, large elements in Q or K^T would grow even larger, dominating the attention mechanism which may cause convergence issues.

Earlier on in Equation (5) we described attention as

$$y_i = \sum_{j=1}^{n} w_{ij} \cdot x_j, \qquad \forall 1 \leq i \leq m. \tag{8}$$

Well, our *attention matrix* after softmax has been applied is simply w with $(i,j)th$ element w_{ij}. The output y_i is just the weighted sum using w on the value vectors, $v = (\vec{v}_1, \ldots, \vec{v}_n)$. It may be clearer to visualize the output as

$$\vec{y} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$$

The attention matrix is a nice thing to visualize. For our toy example, it might look like

$w =$ (9)

	What	*is*	*the*	*capital*	*of*	*France*	*?*
What	0.71	0.12	0.32	0.29	0.23	0.03	0.49
is	0.12	0.65	0.04	0.37	0.27	0.15	0.13
the	0.32	0.04	0.68	0.21	0.11	0.36	0.22
capital	0.29	0.37	0.21	0.59	0.12	0.39	0.41
of	0.23	0.27	0.11	0.12	0.67	0.20	0.15
France	0.03	0.15	0.36	0.39	0.20	0.81	0.12
?	0.49	0.13	0.22	0.41	0.15	0.12	0.70

What can we notice about our attention matrix?

- It is symmetric. That is, $w = w^T$. This is to be expected, as remember it was produced by computing QK^T where Q and K are identical.
- The largest values are often times found on the leading diagonal. You can think of the values in the matrix as some measure of how important one token is to another. Typically, we try to ensure that each token pays attention to itself to some extent.
- Every cell is filled. This is because in this attention approach, every token attends to every other token. This is often referred to as *full* n^2 *attention*. In Section (4) you will see other ways of defining this attention matrix.

2.2 Multi Head Self Attention

It's important to acknowledge that there may not exist a single perfect representation of the attention matrix. Multi Head Self Attention allows us to produce many different representations of the attention matrix. Each individual attention mechanism is referred to as a "head". Each head learns slightly different representations of the input sequence, which the original researchers found prompted the best output [7].

Firstly, we're going to introduce some new matrices. These will be defined as

$$Q = (n \times d_q), \quad K = (n \times d_k), \quad V = (n \times d_v)$$

These matrices will be obtained by linearly transforming the original matrix M, using weight matrices W^Q, W^K and W^V respectively:

$$\begin{aligned} Q &= M \times W^Q, \\ K &= M \times W^K, \\ V &= M \times W^V. \end{aligned}$$

Each of these matrices has d_{model} rows, and remember that M has d_{model} columns. We have control over parameters d_q, d_k, d_v. In the original research they took $d_q = d_k = d_v = d_{\text{model}}/8 = 64$[7].

We're going to use a different set of weight matrices W^Q, W^K and W^V for each head. If we have H heads, we will refer to the set of weight matrices of the h_{th} head as $\{W_h^Q, W_h^K, W_h^V\}$. For a given head, h, the output of the attention mechanism is

$$h_i = \text{attention}(M \cdot W_h^Q, M \cdot W_h^K, M \cdot W_h^V) \tag{10}$$

In Section (3) you will see how we can make the computation of Multi Head Attention more efficient. The overall output of the process is then simply

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W^O. \tag{11}$$

Concat() simply concatenates our output matrices. The output matrix of size $(n \times d_v)$ for each head is simply our matrices stacked on top of one another like so

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_h) = \begin{pmatrix} \text{head}_{1_{11}} & \ldots & \text{head}_{1_{1d_v}} & \ldots & \text{head}_{H_{11}} & \ldots & \text{head}_{H_{1d_v}} \\ \text{head}_{1_{21}} & \ldots & \text{head}_{1_{2d_v}} & \ldots & \text{head}_{H_{21}} & \ldots & \text{head}_{H_{2d_v}} \\ \vdots & \ddots & \vdots & \ldots & \vdots & \ddots & \vdots \\ \text{head}_{1_{n1}} & \ldots & \text{head}_{1_{nd_v}} & \ldots & \text{head}_{H_{n1}} & \ldots & \text{head}_{H_{nd_v}} \end{pmatrix}$$

This output has dimension $(n \times Hd_v)$. We still have n rows, however now we have h different representations of d_v. Our output, W^O, is another trainable weight matrix which has dimensions $W^O = (Hd_v \times d_{\text{model}})$. Therefore, the multiplication of $\text{Concat}(\text{head}_1, \ldots, \text{head}_H)$ and W^O results in a matrix with dimension $(n \times d_{\text{model}})$.

3 Memory Bandwidth Reduction via Key and Value

3.1 Multi Query Attention

Multi Query Attention (MQA) [6] using the same K and V matrices for each head in our multi head self attention mechanism. For a given head, h, $1 \leq h \leq H$, the attention mechanism is calculated as

$$h_i = \text{attention}(M \cdot W_h^Q, M \cdot W^K, M \cdot W^V). \tag{12}$$

For each of our H heads, the only difference in the weight matrices is in W_h^Q. Each of these W_h has dimension $(n \times d_q)$. The attention output for each head i is given by

$$\text{attention}(Q_h, K, V) = \text{softmax}\Big(\frac{Q_h \cdot K^T}{\sqrt{d_k}}\Big) \cdot V \tag{13}$$

As before, we simply concatenate our attention outputs and multiply by W^O, which is defined as before.

3.2 Grouped Query Attention

Grouped Query Attention (GQA) [1] is very similar to MQA. The difference is that instead of using just one set of K, V values for attention calculations it uses G different sets of K, V values. If we have H heads, GQA is equivalent to MHA if $G = H$ and equivalent to MQA if $G = 1$. Suppose we want to use G groups. We would

1. Allocate each of our H heads into one of the G groups. It would likely make sense to pick G such that $G \mod H \equiv 0$. Though this is not a requirement.
2. For each head in a given group, we calculate attention outputs as

$$\begin{aligned} \text{attention}(h) &= \text{attention}(M \cdot W_h^Q, M \cdot W_g^K, M \cdot W_g^V) & (14) \\ &= \text{softmax}\Big(\frac{Q_h \cdot K_g^T}{\sqrt{d_k}}\Big) \cdot V_g & (15) \end{aligned}$$

The query matrices will be shared by all groups under a given head, and the key and value matrices will be used for all attention calculations within a given group.

3.3 Conversions from Multi Head Attention

A natural question might be how one could take a model which uses multi-head attention and convert it to model using multi query attention or grouped query attention. To convert to multi query attention, we want to find a single representative matrix for both K and V from our set of H different heads. We achieve this via mean pooling. For instance for K,

$$\text{mean pooling}(K_1, \ldots, K_h) \to K'. \tag{16}$$

We need to decide the size of our mean pooling window, w. Our process then involves

1. Divide each of the input matrices $(K_1, \ldots, K_H)$ into non-overlapping $w \times w$ regions,
2. Compute the average value within each $w \times w$ region for each input matrix $(K_1, \ldots, K_H)$,
3. Compute the mean of the corresponding regions across all H input matrices and set this to the corresponding values in our final matrix K'.

We now have our matrix K'. It is required at this stage to pre-train for a small portion of the original training steps. The process is nearly identical for grouped query attention. However this time we mean pool over each group of matrices (instead of the whole set). The matrices within a given group are simply dictated by how we chose to assign our G groups to the original H heads.

4 Adaptations to the Attention Matrix

4.1 Sliding Window Attention

Sliding Window Attention [2] reduces the number of calculations we are doing when computing self attention. Previously, to compute attention we took our input matrix of positional encodings M, and made copies named Q, K and V. We used these copies to compute

$$\text{attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^T}{\sqrt{d_k}}\Big)V. \tag{17}$$

For now, let's ignore the re-scaling by $\sqrt{d_k}$ and just look at the computation of QK^T. This computation looks like

$$Q \times K^T = \begin{pmatrix} Q_{11} & Q_{12} & \cdots & Q_{1d} \\ Q_{21} & Q_{22} & \cdots & Q_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{n1} & Q_{n2} & \cdots & Q_{nd} \end{pmatrix} \times \begin{pmatrix} K_{11} & K_{21} & \cdots & K_{n1} \\ K_{12} & K_{22} & \cdots & K_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ K_{1d} & K_{2d} & \cdots & K_{nd} \end{pmatrix} \tag{18}$$

Our goal is to simplify this computation. Instead of letting each token attend to all of the other tokens, we will define a window size w. The token we are calculating attention values for will then only get to look at the tokens $\frac{1}{2}w$ either side of it. For our example, we could consider a sliding window of size 2 which will look 1 token to either side of the current token. Only the values shaded in olive will be calculated.

$$w = \begin{pmatrix} & \textit{What} & \textit{is} & \textit{the} & \textit{capital} & \textit{of} & \textit{France} & \textit{?} \\ \textit{What} & 0.71 & 0.12 & 0.32 & 0.29 & 0.23 & 0.03 & 0.49 \\ \textit{is} & 0.12 & 0.65 & 0.04 & 0.37 & 0.27 & 0.15 & 0.13 \\ \textit{the} & 0.32 & 0.04 & 0.68 & 0.21 & 0.11 & 0.36 & 0.22 \\ \textit{capital} & 0.29 & 0.37 & 0.21 & 0.59 & 0.12 & 0.39 & 0.41 \\ \textit{of} & 0.23 & 0.27 & 0.11 & 0.12 & 0.67 & 0.20 & 0.15 \\ \textit{France} & 0.03 & 0.15 & 0.36 & 0.39 & 0.20 & 0.81 & 0.12 \\ \textit{?} & 0.49 & 0.13 & 0.22 & 0.41 & 0.15 & 0.12 & 0.70 \end{pmatrix} \tag{19}$$

This greatly reduces the cost of the computation of $Q \times K^T$, as our computation will now look like

$$Q \times K^T = \begin{pmatrix} Q_{11} & Q_{12} & & \\ Q_{21} & Q_{22} & \cdots & \\ & \vdots & \ddots & \vdots \\ & & \cdots & Q_{nd} \end{pmatrix} \times \begin{pmatrix} K_{11} & K_{21} & & \\ K_{12} & K_{22} & \cdots & \\ & \vdots & \ddots & \vdots \\ & & \cdots & K_{nd} \end{pmatrix} \tag{20}$$

However, the original authors encountered a problem in training. The authors found that this approach is not flexible enough to learn to complete specific tasks. They solved this problem through the introduction of *global attention*. This will give a few of our tokens some special properties:

- A token with a global attention attends to all other tokens in the sequence
- All tokens in the sequence attend to every token with a global attention.

The local attention (sliding window attention) is primarily used to build contextual representations, while the global attention allows the model to build full sequence representations for prediction [2].

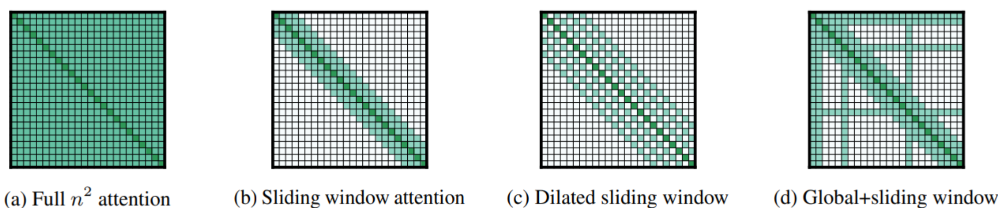
We will require two sets of our projection matrices. Firstly, projections to compute attention scores for our sliding window approach $\{Q_s, K_s, V_s\}$ and secondly attention scores for the global attention $\{Q_g, K_g, V_g\}$. These are initialized to the same values.

We first calculate local attention weights using $\{Q_s, K_s, V_s\}$. This gives us an attention output, which is then combined with the output using the global attention weights. The global weights are written on top of the output attention weight matrix calculated by the local attention calculation.

Dilated Sliding Window Attention is another approach to achieve a similar result. This time, instead of simply taking the $\frac{1}{2}w$ tokens either side of a given w we will introduce some gaps of size d. This is referred to as the dilation. Using $w = 2, d = 1$ in our example we would have an attention matrix which looks like

$$w = \left(\begin{array}{c|ccccccc} & \textit{What} & \textit{is} & \textit{the} & \textit{capital} & \textit{of} & \textit{France} & \textit{?} \\ \textit{What} & 0.71 & 0.12 & 0.32 & 0.29 & 0.23 & 0.03 & 0.49 \\ \textit{is} & 0.12 & 0.65 & 0.04 & 0.37 & 0.27 & 0.15 & 0.13 \\ \textit{the} & 0.32 & 0.04 & 0.68 & 0.21 & 0.11 & 0.36 & 0.22 \\ \textit{capital} & 0.29 & 0.37 & 0.21 & 0.59 & 0.12 & 0.39 & 0.41 \\ \textit{of} & 0.23 & 0.27 & 0.11 & 0.12 & 0.67 & 0.20 & 0.15 \\ \textit{France} & 0.03 & 0.15 & 0.36 & 0.39 & 0.20 & 0.81 & 0.12 \\ \textit{?} & 0.49 & 0.13 & 0.22 & 0.41 & 0.15 & 0.12 & 0.70 \end{array} \right) \tag{21}$$

The authors provide a nice visual of how this looks generally, which you can see in Figure (1). The authors note they use dilated sliding window attention with small window sizes for lower layers, and larger window sizes for higher layers. They do not introduce dilation for lower layers, however for higher layers a small amount of increasing dilation was introduced on 2 heads.



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

Figure 1: Attention Mechanisms, taken from [2].

4.2 Sparse Attention

Sparse Attention [3] introduces sparse factorizations on the attention matrix. To implement this we introduce a *connectivity pattern* $S = \{S_1, \dots, S_n\}$. Here, S_i denotes the set of indices of the input vectors to which the ith output vector attends. For instance, in regular n^2 attention every input vector attends to every output vector before it in the sequence. Remember that d_k is the inner dimension of our queries and keys. Sparse Attention is given as follows

$$\text{attention}(Q, K, V, S_i) = \text{softmax}\Big(\frac{(Q_{S_i})K_{S_i}^T}{\sqrt{d_k}}\Big)V_{S_i}. \tag{22}$$

Here, we have defined

$$Q_{S_i} = \big(W_q\vec{x}_j\big)_{j\in S_i}, \qquad K_{S_i} = \big(W_k\vec{x}_j\big)_{j\in S_i}, \qquad V_{S_i} = \big(W_v\vec{x}_j\big)_{j\in S_i}. \tag{23}$$

So how do we define the set of connectivity patterns S? Formally, we let $S_i = A_i^h$ for head h where $A_i^h \subset \{j : j \leq i\}$. It is still no clearer how we pick which indices we should take for a given S_i. The original authors consider two key criteria initially:

1. We should pick $|A_i^h| \propto n^{1/H}$ where H is our total number of heads. This choice is efficient as it ensures the size of the connectivity set scales well with H.
2. All input positions are connected to output positions across p steps of attention. For instance, for a pair $j \leq i$ we would like i to be able to attend to j through a path of locations with maximum length $p + 1$. This helps us propagate signals from input to output in a constant number of steps.

We now investigate two different approaches that satisfy this criteria, and allow us to implement sparse attention.

Strided Attention. We will define a factorized attention pattern in two heads. One head will attend to the previous l locations, while the other head will attend to every lth location. We call l the stride and it is chosen to be close to $\sqrt{n}$.

$$A_i^{(1)} = \{y, y+1, \dots, i\} \text{ for } t = \max(0, i-l), \tag{24}$$

$$A_i^{(2)} = \{j : (i-j) \mod l \equiv 0\}. \tag{25}$$

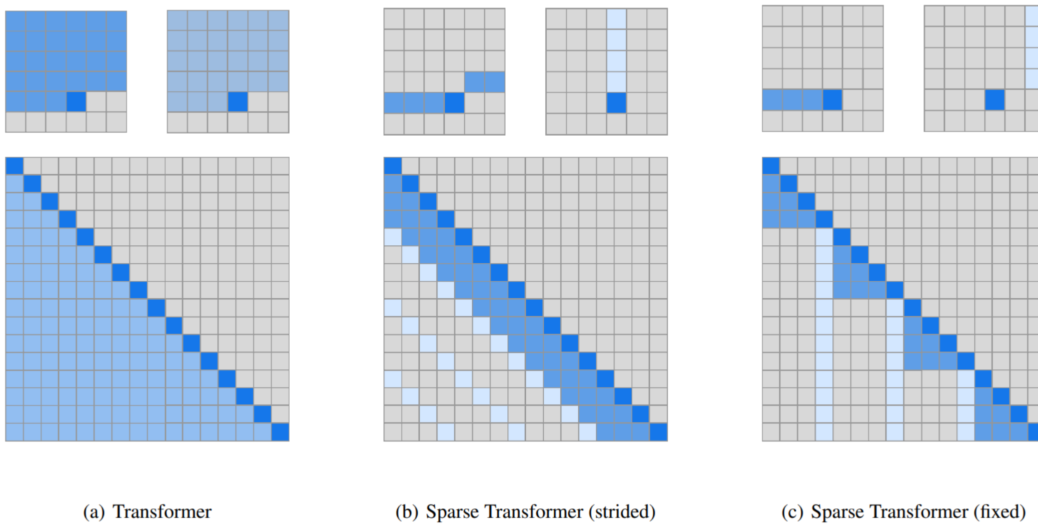
Here, $A_i^{(1)}$ simply takes the previous l locations. $A_i^{(2)}$ then takes every lth head from the first head where $i - j$ was divisible by l without remainder. This is particularly useful where you can align the structure of your input with the stride. For instance, with a piece of music. Where our input does not have a well defined structured, we use something different. In Figure (2), you can see $A_i^{(1)}$ responsible for the dark blue shading and $A_i^{(2)}$ responsible for the light blue.

Fixed Attention. Our goal with this approach is to allow specific cells to summarize the previous locations, and to propagate this information on to future cells.

$$A_i^{(1)} = \left\{ j : \left\lfloor \frac{j}{l} \right\rfloor = \left\lfloor \frac{i}{l} \right\rfloor \right\},$$

$$A_i^{(2)} = \left\{ j : j \mod l \in \{t, t+1, \dots, l\} \right\}, \quad \text{where } t = l - c \text{ and } c \text{ is a hyperparameter.}$$

These are best understood visually in my opinion. In Figure (2), $A_i^{(1)}$ is responsible for the dark blue shading and $A_i^{(2)}$ for the light blue shading. If we take stride, $l = 128$ and $c = 8$, then all positions greater than 128 can attend to positions $120 - 128$. The authors find choosing $c \in \{8, 16, 32\}$ worked well. Choice of c controls the number of vertical lines in Figure 2) (c).



(a) Transformer

(b) Sparse Transformer (strided)

(c) Sparse Transformer (fixed)

Figure 2: Types of Sparse Attention: visualizing the *connectivity* matrix.

5 Inference

5.1 The KV Cache

The computation of attention is costly. Remember that our decoder works in an auto-regressive fashion. For our given input "*What is the capital of France ?*"

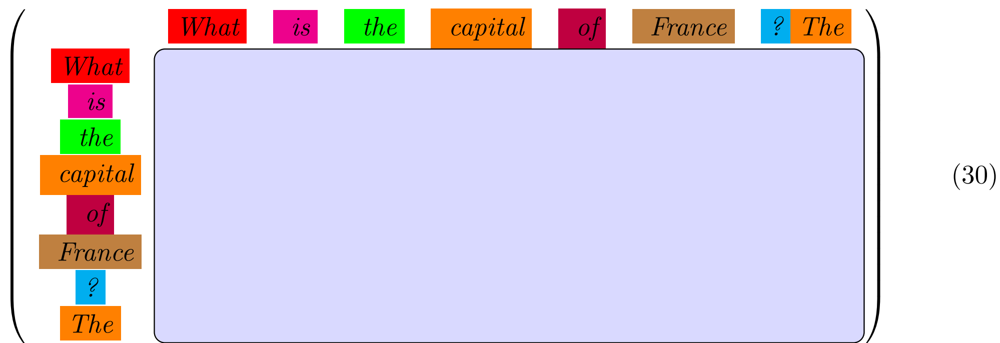
$$\text{Prediction } 1 = \text{The} \tag{26}$$

$$\text{Prediction } 2 = \text{The capital} \tag{27}$$

$$\vdots \tag{28}$$

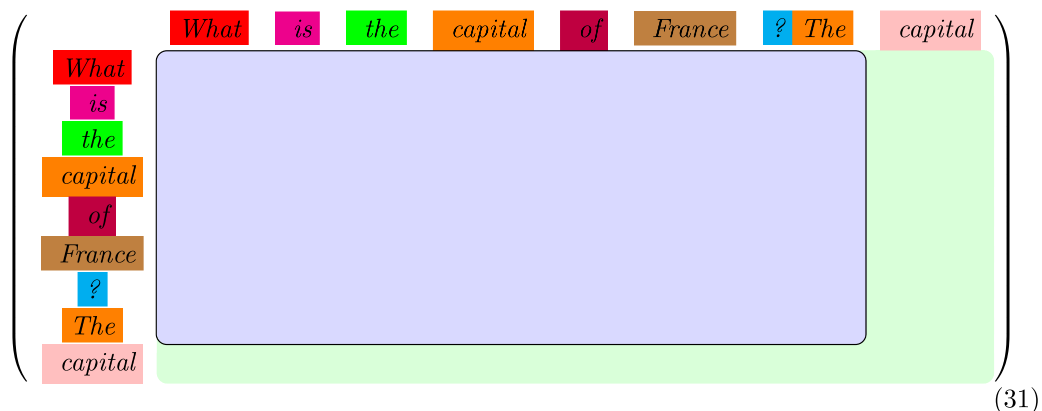
$$\text{Prediction } p = \text{The capital} (\dots) \text{ Paris.} \tag{29}$$

To produce prediction 2, we will take the output from prediction 1. At each step, the model will also see our input sequence. Without any tricks, at every step, we're going to be re-computing values that have already been calculated. Our attention matrix used for our first prediction will have the following structure



(30)

When we compute the second prediction, the structure of our attention matrix looks very similar. Notice that the attention matrix after prediction one is actually contained within this matrix!



(31)

Remember, Q and K^T are just defined by our matrix M which contains one row per input token. Thus, Q and K^T are very similar between the first and second predictions - only one row / column has changed! By caching K for each prediction, we can make the computation of our attention matrix more efficient and by caching V, we make our attention mechanism output calculation more efficient.

5.2 Flash Attention

The goal of *Flash Attention* [5] is to compute the attention value with fewer high bandwidth memory read / writes. The approach has since been refined in *Flash Attention 2* [4].

We will split the attention inputs Q, K, V into blocks. Each block will be handled separately, and attention will therefore be computed with respect to each block. With the correct scaling, adding the outputs from each block we will give us the same attention value as we would get by computing everything all together.

Tiling. To compute attention, we multiply $Q \times K^T$, divide by $\sqrt{d_k}$ and then take the softmax. Keeping track of the scaling values in softmax is the key to making this technique work. The softmax for a vector $\vec{x} \in \mathbb{R}^{2n}$ is given by

$$m(x) := \max_i x_i, \quad f(x) := [e^{x_1 - m(x)}, \ldots, e^{x_b - m(x)}], \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}$$

This looks unfriendly, but is really just the notation for a more numerically stable softmax. What does that mean? Well, notice we are just applying regular softmax but with some shifting of each element of vector $\vec{x}$ by $\max(x)$ units. We can do this because $\text{softmax}(\vec{x}) = \text{softmax}(\vec{x} - c)$ for any scalar c.

In this case, we improve numerical stability by ensuring we do not take the exponential of very large numbers. This can lead to overflow issues. This simply means our number gets too big to store in the given datatype. By subtracting the largest element, we ensure the vector $\vec{x}$ only has non-positive entries. For example, in floating point 64, the maximum value we can represent is very large (10^{308}). However

$$e^x > 10^{308} \implies x > \ln(10^{308}) \implies x > 308 \times \ln(10) \implies x > 709 \tag{32}$$

Therefore, approximately any x larger than 709 will result in overflow issues. For instance, computing $\exp(709) = 8.22e + 307$ but $\exp(710) = inf$ in *numpy*. We certainly do not want our model to hit any overflow errors.

To compute softmax in blocks, we decompose our vector $\vec{x} \in \mathbb{R}^{2n}$ into two smaller vectors in $\mathbb{R}^n$.Let's look at the simple case of decomposing into two vectors. Denote these vectors $\vec{x}_1, \vec{x}_2$ each in $\mathbb{R}^n$. Our softmax calculation becomes

$$\begin{aligned} m(x) &= m([x_1 \quad x_2]) = \max(m(x_1), m(x_2)), \\ f(x) &= [e^{m(x_1)-m(x)} f(x_1) \quad e^{m(x_2)-m(x)} f(x_2)], \\ \ell(x) &= \ell([x_1 \quad x_2]) = [e^{m(x_1)-m(x)} \ell(x_1) \quad e^{m(x_2)-m(x)} \ell(x_2)], \\ \text{softmax}(x) &= \frac{f(x)}{\ell(x)}. \end{aligned}$$

Notice that we use $m(x_i) - m(x)$ as the normalization factor, as we do not know which group will contain the maximum value of $\vec{x}$. By keeping track of both $m(x)$ and $\ell(x)$ we will be able to accurately recombine the softmax outputs for each block, as will know how to rescale the softmax outputs.

Recomputation. We also do not wish to store all the intermediate values we calculate for every backward pass. Typically we require the attention matrix, QK^T, and the output after softmax, simply $\text{softmax}(QK^T)$ in each backward pass. However, by using our blocks of Q, K, V the whole attention matrix is not required to be loaded in during every backward pass.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

[3] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

[4] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[6] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.