

Lecture 6: Attention Mechanism

Instructors: *Alex Andoni, Edo Liberty*Scribes: *Sean Kim*

1 Overview of Transformers and LLMs

Learning is the method of generating a useful function that maps inputs to outputs. In language modeling, the input is usually a sequence $x = (x_1, \dots, x_n)$ of n tokens, and the output is usually a probability distribution $y = (y_1, \dots, y_m)$ over a token vocabulary of size m . The output y represents the predicted probabilities of each token coming next in the sequence (at position $n + 1$). Generating the next token in the sequence simply amounts to sampling from this probability distribution.

The Transformer [2] is the most popular architecture in use today for language modeling and other tasks. It is easily optimizable (thanks to residual connections and layer normalization), runs very efficiently on GPUs, and uses attention as its sole communication mechanism amongst input tokens (instead of recurrence). It is worth noting that these ideas existed prior to the introduction of the Transformer (e.g., attention existed for multiple years before it), but the Transformer was a successful combination of these ideas within a single architecture.

The Transformer consists of a stack of Transformer blocks, each of which performs the operations:

$$\begin{aligned}x' &= x + \text{Attention}(\text{LayerNorm}(x)), \\y &= x' + \text{MLP}(\text{LayerNorm}(x')).\end{aligned}$$

where x is the input to the block, y is its output, and its components are:

- **Layer normalization (LayerNorm):** Normalizes across the channel (feature) dimension, which helps stabilize activations during training.
- **Attention:** The communication mechanism described in Section 2.
- **Feed-forward network (MLP):** A simple fully connected multilayer perceptron applied independently to each token representation.
- **Residual connection (+):** Adds the input of a sublayer to its output, providing a direct path for gradient flow and helping mitigate the vanishing-gradient problem [1].

To better understand the utility of residual connection, one toy example is to consider the operation $a + b = c$. If $\frac{\partial L}{\partial c} = d$, then the gradient flowing directly from c to a is $\frac{\partial L}{\partial a} = \frac{\partial L}{\partial c} \frac{\partial c}{\partial a} = d$, since $\frac{\partial c}{\partial a} = 1$.

More generally, consider a residual connection $o = w_t \sigma(w_{t-1} \sigma(\dots \sigma(w_1 a))) + a$. Even if the gradient through the w 's and σ 's becomes very small, the addition provides a direct gradient path from o to a with derivative 1. Since gradient accumulation is additive, gradients can flow through the residual pathway unimpeded.

We are now ready to describe the basic data flow in a typical Transformer:¹

¹Although there have been many modifications and additional architectural details, the core structure described here remains largely the same.

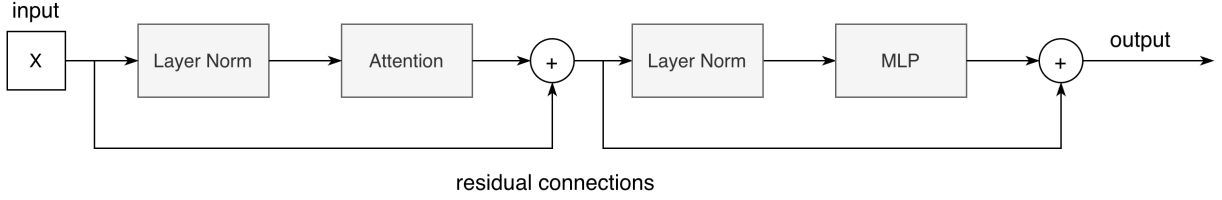


Figure 1: Diagram of the data flow in a typical pre-norm [4] Transformer block. Pre-norm applies layer normalization before each sublayer (attention or MLP), whereas post-norm applies it after each sublayer.

1. A batch of B input token sequences, each of length T , is fed into the Transformer as a $B \times T$ matrix.
2. Every token indexes into a token embedding lookup table, obtaining an embedding vector of size d_{model} (the model’s channel dimension). This gives a $B \times T \times d_{\text{model}}$ tensor.
3. Every position in the sequence indexes into a positional encoding lookup table, obtaining positional information that is added to the corresponding token embedding (using broadcasting).² The resulting tensor remains of size $B \times T \times d_{\text{model}}$.
4. The token representations are passed through a stack of Transformer blocks.
5. A linear projection layer projects each token representation from size d_{model} to the vocabulary size d_{vocab} . This gives a $B \times T \times d_{\text{vocab}}$ tensor, whose values are called the *logits* (the model’s raw, unnormalized scores).
6. Softmax is applied independently to each vocabulary-sized logit vector, producing a probability distribution over the vocabulary for each token position.

2 Causal Multi-Head Self-Attention

The following is the data flow in a typical causal multi-head self-attention³ block:

1. The input to the attention block is a tensor X of size $B \times T \times d_{\text{model}}$. Recall that B is the batch size (number of input sequences processed in parallel), T is the sequence length (number of tokens in each input sequence), and d_{model} is the model’s channel dimension.
2. The number of attention heads h is a chosen hyperparameter, where $d_{\text{model}} \bmod h = 0$. For each head $i = 1$ to h , the input is linearly projected using learned weight matrices

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v},$$

²In modern Transformer LLMs, the addition of learned absolute positional embeddings is often replaced by Rotary Position Embeddings (RoPE). RoPE applies position-dependent geometric rotations to the query and key vectors. The dot product between a query at position i and key at position j then depends on their relative positional displacement, which incorporates positional information into the attention scores.

³We call it “self”-attention because the queries, keys, and values all come from the same input sequence. In contrast, in “cross”-attention, the queries come from one source while the keys and values come from another. For example, encoder-decoder architectures used in machine translation source the keys and values from the encoder outputs (e.g., “Je t’aime”), while the queries come from the decoder representations (e.g., in producing “I love you”). This allows each decoder token to attend to the encoded input sequence as it generates the output.

producing

$$Q_i = XW_i^Q, \quad K_i = XW_i^K, \quad V_i = XW_i^V.$$

Thus, Q_i and K_i have shape $B \times T \times d_k$, and V_i has shape $B \times T \times d_v$.

3. For each head $i = 1$ to h , compute

$$Q_i K_i^\top,$$

which has shape $B \times T \times T$. The (j, k) -th entry is the dot product between the query of token j and the key of token k . Intuitively, the query asks “what am I looking for?”, the key responds “here is what information I can give you”, and the dot product is a scalar measure of their affinity, which we call *attention score*.

4. Scale the attention scores by $1/\sqrt{d_k}$:

$$\frac{Q_i K_i^\top}{\sqrt{d_k}}.$$

This prevents the magnitude of the dot products from growing too large as d_k increases, which can saturate the softmax and cause gradients to vanish.

To understand the scaling factor, consider vectors q and k of size d_k whose components are independent random variables with zero mean and unit variance. Their dot product is a sum of d_k independent products, each with zero mean and unit variance. Thus, $q \cdot k$ has variance d_k and standard deviation $\sqrt{d_k}$. Dividing by $\sqrt{d_k}$ normalizes this standard deviation back to 1, keeping the scale roughly constant as d_k increases [2].

5. Apply a causal mask by setting entries corresponding to future tokens to $-\infty$. In particular, the score between query position j and key position k is masked whenever $k > j$. This prevents a token from attending to future tokens.
6. Apply softmax across the key dimension:

$$A_i = \text{softmax} \left(\frac{Q_i K_i^\top}{\sqrt{d_k}} \right),$$

which converts the attention scores into *attention weights* that sum to 1 for each query. Exponentiation within the softmax maps attention scores of $-\infty$ to 0, thereby giving masked positions zero attention weight (which preserves the autoregressive property).

7. Compute a weighted sum of the value vectors, where weights are the attention weights:

$$H_i = A_i V_i.$$

Each token’s output can therefore be interpreted as a weighted summary of the value vectors, where weights are determined by corresponding query-key affinities. Each H_i has shape $B \times T \times d_v$.

8. Concatenate the outputs of all h attention heads:

$$H = \text{Concat}(H_1, \dots, H_h),$$

giving a tensor of shape $B \times T \times (hd_v)$.

9. A learned output projection

$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

gives an output tensor of shape $B \times T \times d_{\text{model}}$.

The intuition for multi-head attention, as opposed to a single attention head, is best understood with a toy example. Suppose a token wants to attend to two different tokens for two different reasons: token A is useful for syntax, and token B is useful for semantic meaning. With a single head, the query-key affinity mechanism may result in equal attention scores for the two tokens, thereby producing attention weights of 0.5 for each token after softmax. The weighted sum then becomes $0.5v_A + 0.5v_B$, which mixes the information from tokens A and B into a single head output.

Multi-head attention provides separate representations for these relationships. For example, head 1 could attend strongly to token A (without attending to token B), while head 2 attends strongly to token B (without attending to token A). Because each head has its own learned projection matrices W_i^Q , W_i^K , and W_i^V , each head generates distinct queries (look for different things), keys (respond to queries differently), values (reveal different information), and attention weights (gather information from others in different proportions). This allows different heads to look for different kinds of relationships, and their outputs remain separate until they are concatenated.

In summary, attention is a direct⁴ communication mechanism that allows tokens in the input sequence to share information with each other. If the attention is causal, a token can only gather information from itself or tokens that appear earlier in the sequence, which prevents it from gathering information from the future (which is what the model is trying to predict in an autoregressive setting). By making it multi-head, we allow for multiple distinct communication channels in parallel. The output of attention, relative to its input, can be seen as a contextualized version of the token representation that contains information about both itself and previous tokens that it attends to most.

3 Computational and Memory Costs of Attention

We now derive the computational cost of performing attention on a single input sequence of length n . Given n tokens and query (Q), key (K), and value (V) matrices of size $n \times d$, our goal is to compute the corresponding attention outputs. We first compute the matrix of attention scores

$$S := QK^\top,$$

which contains the dot product between each of the n query vectors and each of the n key vectors. There are n^2 such query-key pairs, and each dot product between two vectors of size d requires $\Theta(d)$ operations. Thus, computing the score matrix costs

$$\Theta(n^2d).$$

Our next step is to apply softmax on each row of the score matrix:

$$\text{softmax}(s)_j = \frac{e^{s_j}}{\sum_{k=1}^n e^{s_k}}, \quad j = 1, \dots, n.$$

Computing softmax over a row of n attention scores requires $\Theta(n)$ operations because we exponentiate each of the n scores in $\Theta(n)$, sum the n exponentials in $\Theta(n)$, and divide each of the n exponentials by that sum in $\Theta(n)$. Since there are n rows, applying softmax to the entire score matrix costs

$$\Theta(n^2).$$

⁴Unlike recurrence, which forces intermediate steps in communication between tokens that are far apart.

We then perform matrix multiplication with the value matrix V , which costs $\Theta(n^2d)$ by the same reasoning as the score matrix. Therefore, computing the attention outputs for an input sequence of size n has total computational cost

$$\Theta(n^2d),$$

which scales quadratically in the sequence length n .

Now, consider the memory and computational costs of attention when processing a new token x_{n+1} , given that we have already computed attention for the preceding tokens $(x_1, \dots, x_n)$. This situation occurs repeatedly during the generation phase of LLM inference, which proceeds as follows:

- 1: **Given** an input sequence $x = (x_1, \dots, x_n)$
- 2: **Given** an end-of-sequence token index i
- 3: **repeat**
- 4: Perform a forward pass on x to obtain the next-token probability distribution y
- 5: Sample x_{new} from y using the desired sampling method
- 6: Append x_{new} to x
- 7: **until** $x_{\text{new}} = i$
- 8: Decode and **return** the generated tokens $(x_{n+1}, \dots)$

Causal self-attention has the property that once attention outputs for tokens at positions 1 through i have been computed, adding a new token at position $i + 1$ does not change these outputs (since causal attention prevents earlier tokens from attending to the new token). That is, the only further computations needed are the query, key, and value for the new token and its corresponding attention output. The computational benefit of this property is amplified by the fact that, outside of the attention blocks, the Transformer processes each token independently. Thus, the entire forward pass remains unchanged for earlier tokens. We can take advantage of this fact during the generation phase of inference, since each newly generated token is appended to the sequence used in the previous forward pass.

Let $(x_1, \dots, x_n)$ be the input to the forward pass at iteration i , and let $(x_1, \dots, x_{n+1})$ be the input at iteration $i + 1$, where x_{n+1} is the output token sampled at iteration i . Because the forward pass for tokens $(x_1, \dots, x_n)$ remains unchanged, outputting $(y_1, \dots, y_n)$, we are only interested in computing y_{n+1} . Therefore, we forward only token x_{n+1} through the Transformer at iteration $i + 1$. Its computation is independent of the previous tokens except in the attention blocks, where it requires the keys and values of the previous tokens (to gather information from them) but not the queries (it need not respond to queries from any previous tokens, due to causality). We employ the strategy of storing the keys and values corresponding to all previous tokens $(x_1, \dots, x_n)$ in a cache, called the *KV cache* (key-value cache), and feeding the forward pass its contents as needed. This allows us to perform the forward pass at iteration $i + 1$ on only token x_{n+1} , at the memory cost of $\Theta(nd)$ per attention block (since for each of the n tokens, we store two vectors of size d).

As a result, given the cached keys and values for $(x_1, \dots, x_n)$, we can compute the attention output for the new token x_{n+1} at a computational cost of

$$\Theta(nd),$$

rather than recomputing attention over the entire sequence at a cost of $\Theta(n^2d)$. The initial forward pass over the input sequence, called the *prefill* phase, still incurs the full attention cost. Subsequent token-generation steps constitute the *decode* phase, during which the KV cache allows each attention block to operate in $\Theta(nd)$ time while incurring a memory cost of $\Theta(nd)$.

4 FlashAttention

We can further speed up attention by considering the hardware upon which we operate. A GPU broadly consists of:

- **Streaming Multiprocessor (SM):** Where most on-chip computation is performed. Each SM contains multiple tensor cores that perform matrix multiplications extremely quickly, along with other specialized computational units.
- **On-Chip Memory:** Small, fast memory located on the GPU chip, including registers and caches. Its capacity is much smaller than HBM, but it can be accessed much more quickly.
- **High-Bandwidth Memory (HBM):** The GPU’s large off-chip main memory. Its capacity is much larger than on-chip memory, but moving data between HBM and the chip is much slower.

Most deep learning operations are well-suited for the GPU and can be executed extremely quickly on-chip. Therefore, many deep learning workloads are memory-bandwidth constrained, and techniques that reduce data movement between HBM and the chip result in significant speedups, sometimes despite requiring additional computation in exchange.

We refer to this principle of maximizing the amount of computation done while data is on-chip as *data reuse*. To reason about it, consider an abstraction which we will call the *I/O model*, where we assume that on-chip computation is free, and equate the *I/O cost* to the amount of data transferred between HBM and cache. Let M denote the cache size, where $d \ll M \ll nd$, so that it can hold meaningful blocks of the attention matrices, but not the entire matrices (which is the typical regime in practice). Let d be the size of each key, query, and value vector, and n be the number of tokens in the sequence.

A naive implementation of computing the attention scores may be to process one query at a time. For each of the n queries, we must move the entire key matrix of size nd , resulting in a total I/O cost of $O(n^2d)$. Instead, we can employ a tiling strategy, where we partition the matrices into tiles (blocks) of size $B \times d$, where $B \approx \frac{M}{d}$ (tiling along the sequence dimension) up to a constant factor (enough to account for having to store both query and key blocks, as well as intermediate values). This means there are approximately n/B query, key, and value blocks. We will load a query block and a key block into cache, and compute all pairwise attention scores between them. Processing each of the $(n/B)^2$ query-key block pairs requires $O(Bd)$ data movement, and the I/O cost is:

$$O\left(\frac{n^2}{B^2}Bd\right) = O\left(\frac{n^2d}{B}\right) = O\left(\frac{n^2d^2}{M}\right),$$

where $\frac{d}{M} \ll 1$. In other words, the tiling strategy allows us to reduce the I/O cost by increasing M .

A naive implementation of softmax and value aggregation can further incur $O(n^2)$ and $O(n^2d)$ I/O costs respectively, which pushes the overall I/O cost to $O(n^2d)$. Instead, we will perform both operations alongside the attention score computation as follows:

- 1: **Given** query, key, and value matrices Q , K , and V
- 2: Divide Q , K , and V into blocks that fit in cache
- 3: **for** each query block Q_i **do**
- 4: Load Q_i into cache
- 5: Initialize running (unnormalized) value aggregation $N_i \leftarrow 0$
- 6: Initialize running normalization factor $Z_i \leftarrow 0$
- 7: **for** each key-value block (K_j, V_j) **do**
- 8: Load K_j and V_j into cache

9: Compute scaled attention scores

$$S_{ij} \leftarrow \frac{Q_i K_j^\top}{\sqrt{d_k}}$$

10: Exponentiate the scores and treat them as unnormalized weights

$$P_{ij} \leftarrow \exp(S_{ij})$$

11: Perform an unnormalized accumulation of values

$$N_i \leftarrow N_i + P_{ij} V_j$$

12: Accumulate the exponentiated scores (normalization factor)

$$Z_i \leftarrow Z_i + P_{ij} \mathbf{1}$$

13: Discard S_{ij} and P_{ij}

14: **end for**

15: Normalize to produce the final attention output⁵

$$O_i \leftarrow N_i \oslash Z_i$$

16: Write O_i to HBM

17: **end for**

18: **return** O

The channel dimension (of size d) is preserved when we split the matrices into blocks, so that each query block contains the full query vectors for a selected subset of tokens in the sequence. Since computing the attention output for each token's query requires query-key dot products with every token's key and a weighted aggregation of every token's value, we keep the query block in cache while iterating over all key-value block pairs.

With the standard implementation of attention, the normalization factor (the sum of all exponentiated attention scores) is not known until we've processed every token's key. This would require us to cache n attention scores before aggregating the values. To avoid this, we use a simple algebraic trick:

$$\left(\frac{a}{a + b + c + \dots} \right) v = av \left(\frac{1}{a + b + c + \dots} \right),$$

where v corresponds to the value of some token to accumulate, a corresponds to the exponentiated score for that token's key (for our fixed query), and $b, c, \dots$ are the exponentiated scores from subsequent tokens' keys. Instead of normalizing and then aggregating, we equivalently aggregate first and normalize at the end (using the normalization factor Z_i in our algorithm).

Because all operations are performed while the corresponding blocks are already in cache (without having to materialize the attention scores), the overall I/O cost remains

$$O\left(\frac{n^2 d^2}{M}\right).$$

⁵ $\oslash$ denotes elementwise division along the rows, using broadcasting.

One necessary extension to this tiling strategy is making the softmax computation numerically stable. When exponentiating the scores $P_{ij} = \exp(S_{ij})$, a large S_{ij} can cause overflow. FlashAttention prevents this by subtracting the row maximum:

$$\text{softmax}(s)_j = \frac{e^{s_j - m}}{\sum_k e^{s_k - m}}, \quad m = \max_k s_k,$$

which computes the equivalent softmax:

$$\frac{e^{s_j - m}}{\sum_k e^{s_k - m}} = \frac{e^{s_j} / e^m}{\sum_k e^{s_k} / e^m} = \frac{e^{s_j}}{\sum_k e^{s_k}}.$$

As a result, all shifted scores $s_j - m$ are guaranteed to be ≤ 0 , with a maximum of 0, and their exponentiated values are therefore ≤ 1 , with a maximum of 1.

As with the normalization factor, we do not know the maximum query-key dot product for a given query until we have iterated over all keys. However, we must exponentiate the scores (thereby risking overflow) before knowing the final row maximum in order to aggregate values as we go. We resolve this using a simple algebraic trick:

$$e^{s_j - m_{\text{new}}} = e^{s_j - m_{\text{old}}} e^{m_{\text{old}} - m_{\text{new}}}.$$

For each block, we first compute its attention scores and update the running maximum. We then exponentiate relative to the maximum known so far, avoiding overflow. At each block, if the running maximum increases from m_{old} to m_{new} , we rescale the previously accumulated weighted values and normalization factor by $e^{m_{\text{old}} - m_{\text{new}}}$. After the final key-value block has been processed, the running maximum is the true row maximum, and the accumulated quantities correspond to the numerically stable softmax.

FlashAttention [5] employs these strategies (tiling and numerically stable softmax) and achieves significant speedups through improved data reuse, despite performing more on-chip calculations. This is a form of *kernel fusion*,⁶ in which multiple operations that could otherwise be executed as separate GPU kernels are combined into a single kernel, avoiding intermediate reads and writes to HBM. Rather than separately computing the attention scores, applying softmax, and aggregating the values, FlashAttention performs these operations within the same kernel while the corresponding blocks remain on-chip. Attention-score blocks are therefore entirely consumed on-chip without ever materializing the $n \times n$ attention matrix in HBM.

5 KV-Cache-Efficient Attention

During the decode phase of autoregressive inference, we store the keys and values of all previous tokens in the KV cache. In standard multi-head attention (MHA), every attention head has its own key and value projections, which requires storing $nh(d_{\text{key}} + d_{\text{value}})$ values across a sequence of n cached tokens and h attention heads. Like most of deep learning, autoregressive decoding is typically bound by memory bandwidth. When generating each new token, these cached keys and values must be repeatedly loaded from memory, making the KV cache size consequential for inference speed.

In order to alleviate the memory capacity and bandwidth costs of KV cache, we can reduce the size of key and value representations. Intuitively, it makes sense that the information in keys and values can be

⁶Another example arises in PyTorch compilation. PyTorch’s eager mode typically dispatches operations as separate GPU kernels, potentially requiring intermediate results to be written to HBM and subsequently read back. For example, elementwise expressions consisting of multiple operations can instead be fused into a single kernel. `torch.compile()` can identify such opportunities and perform kernel fusion, reducing unnecessary data movement at the cost of initial compilation overhead.

compressed more so than queries. The key and value of a token advertise and provide information about itself, which has less need for multiple independent channels, whereas multiple independent queries not only aggregate different proportions of information from other tokens, but also look for different traits (thereby making much more critical use of independent channels).

Rather than computing distinct keys $K_1, \dots, K_h$ and values $V_1, \dots, V_h$ for the h attention heads, *Multi-query attention* (MQA) [3] computes a single K, V that is shared across all heads. This reduces the size and memory-bandwidth requirements of the KV cache by approximately a factor of h relative to standard multi-head attention (ignoring other parts of the model).

Sharing a single set of keys and values across all query heads nonetheless reduces the representational power and flexibility provided by multi-head attention and can degrade model quality. *Grouped-query attention* (GQA) [6] divides the query heads into groups and assigns each group its own shared key-value head. Given h query heads and g key-value heads (where $h \bmod g = 0$), each key-value head is shared by h/g query heads. In practice, GQA is commonly used in modern LLMs as it can optimize the tradeoff between the representational capacity of MHA and the inference efficiency of MQA.

Another approach is to reduce the dimensionality of the information stored in KV cache. *Multi-head latent attention* (MLA) [7] projects the hidden representation of each token into a lower-dimensional key-value latent representation, which is cached instead of the full keys and values. The required keys and values are recovered from this latent representation during inference.

Let h_t be the hidden representation of token t as input to an attention layer. MLA computes a compressed key-value representation

$$c_t^{KV} = W^{DKV} h_t,$$

where W^{DKV} projects onto a latent dimension d_c much smaller than the full multi-head key-value dimension. The keys and values used by attention can then be obtained through learned projections:

$$k_t = W^{UK} c_t^{KV}, \quad v_t = W^{UV} c_t^{KV},$$

which allows us to cache the much smaller c_t^{KV} and recover the full-dimensional keys and values during attention.⁷

References

- [1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” *Advances in Neural Information Processing Systems*, 2017.
- [3] N. Shazeer, “Fast Transformer Decoding: One Write-Head is All You Need,” *arXiv preprint arXiv:1911.02150*, 2019.
- [4] R. Xiong, Y. Yang, D. He, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, “On Layer Normalization in the Transformer Architecture,” *Proceedings of the 37th International Conference on Machine Learning*, 2020.

⁷Rotary position embeddings (RoPE) is applied to keys after their projection, so the information introduced by the position-dependent rotation cannot be absorbed into the same low-rank decomposition. MLA resolves this by caching this positional information separately, which is relatively negligible in size.

- [5] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” *Advances in Neural Information Processing Systems*, 2022.
- [6] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints,” *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.
- [7] DeepSeek-AI, “DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model,” *arXiv preprint arXiv:2405.04434*, 2024.