LINFORMER: SELF-ATTENTION WITH LINEAR COMPLEXITY

ELIE BAKOUCH, YVES LECONTE

Abstract. The article Linformer [1] introduces a new architecture for the self-attention mechanism used in transformers. In its original form [2], its computational cost was of $\mathcal{O}(n^2)$ which is a serious bottleneck regarding large scale applications. The model Linformer is essentially based on the low-rank property of the self-attention matrix.

Contents

Introduction

Let us remind transformers map an input vectors $(x_1, ..., x_n)$ with $x_i \in \mathbb{R}^{\mathtt{d}}$ to an output vector $(y_1, ..., y_n)$ with $y_i \in \mathbb{R}^{\mathtt{d}}$ where d is the embedding dimension. We also remind the self-attention mechanism can be denoted as:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

with $X \in \mathbb{R}^{n\times d}$, $Q = XW^Q \in \mathbb{R}^{\mathtt{n}\times\mathtt{d}_\mathtt{k}}$, $K = XW^K \in \mathbb{R}^{\mathtt{n}\times\mathtt{d}_\mathtt{k}}$, $V = XW^V \in \mathbb{R}^{\mathtt{n}\times\mathtt{d}_\mathtt{v}}$ the query, key, and value matrices using $W^Q \in \mathbb{R}^{\mathtt{d}\times\mathtt{d}_\mathtt{k}}$, $W^K \in \mathbb{R}^{\mathtt{d}\times\mathtt{d}_\mathtt{k}}$, $W^V \in \mathbb{R}^{\mathtt{d}\times\mathtt{d}_\mathtt{v}}$ where d_k is the dimension of key and query vectors of each x_i, and d_v of the value vectors. We also remind that using a multi-headed architecture gives us the following equations derived from (1):

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{2}$$

where

$$\text{head}_i = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3}$$

with $Q = XW_i^Q$, $K = XW_i^K$, $V = XW_i^V$ the query, key, and value matrices of each head, and $W^O \in \mathbb{R}^{\mathtt{h}\mathtt{d}_\mathtt{v}\times\mathtt{d}}$ the projection allowing to reduce the output to its original dimension. Let us denote $P = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$, we have $P \in \mathbb{R}^{n\times n}$. The computation of the matrix P, and more precisely the computation of QK^T is the main driver of the paper as it is the reason why the self-attention mechanism requires $\mathcal{O}(n^2)$ operations.

Date: October 2023.

1. Linformer model

The model Linformer is based on two theorems that are stated in the given paper. The first theorem boils down to saying that we can approximate the matrix P using a low rank matrix $\tilde{P} \in \mathbb{R}^{\mathbb{n}\times\mathbb{n}}$ with $\text{rank}(\tilde{P}) \approx \log(n)$. Meaning that we can perform SVD and use a low rank approximation of P costing $O(nk)$ operations instead of $\mathcal{O}(n^2)$ with $k \sim log(n)$. There are two flaws to this approach, the first one being that performing SVD on each attention matrix is costly, and the second one is that here k depends on n, thus there is little use to this result on its own and we are far from having a linear complexity result. The second theorem states that when $k = \mathcal{O}(d_k/\epsilon)$, which is independent of n, then we can approximate 1 with

$$\text{head}_i = \text{softmax}\left(\frac{Q(E_iK)^T}{\sqrt{d_k}}\right)F_iV \tag{4}$$

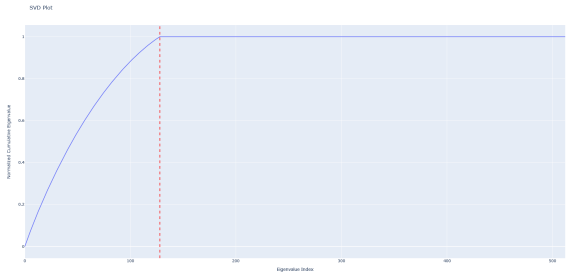
in a multihead-attention setting, using $E_i, F_i \in \mathbb{R}^{\mathbb{k}\times\mathbb{n}}$.[1] In practice, for the sake of understanding what we are doing, in the proof $E_i = \delta R$ and $F_i = e^{-\delta}R$ where $R \in \mathbb{R}^{\mathbb{k}\times\mathbb{n}}$ with i.i.d. entries of $\mathcal{N}(0, 1/k)$ setting $k = 5\text{log}(nd_k)/(\epsilon^2 - \epsilon^3)$ and δ a "small constant". Thus, the yielded result implies that k still depends on n, which isn't beneficial. However, is put under the carpet that using the low-rank property, it is possible to prove that the choice of k can be made constant and independent of n. We find that heuristically, this result quite disturbing, as it means that we can approximate the self-attention matrix of size $n \times n$ using projection matrices of size $n \times k$ with k constant. We also notice that this method faces challenges in situations where the lengths of different inputs vary greatly. The issue is that we need to set a single, global sequence length for training our model. This becomes problematic when dealing with inputs of varying lengths, as it can affect the model's performance and efficiency.
From those last result, are yielded various efficiency techniques based on the choice of the projection matrices and their dimensions.

2. Experiments around Linformer

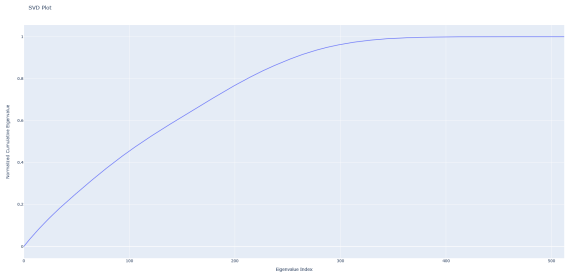
We first decided to conduct a similar experiment to the first experiment of the article, testing the low-rank property of matrix P. Additionally, we looked at how things change before and after using the softmax function. We did this by performing a singular value decomposition. Then, we checked where the normalized cumulative sum is roughly 1. What we found is that the non-linearity in the softmax function affects the rank of the matrix. Matrices that were low rank don't stay so low rank anymore after applying softmax as we can see in Figure 1.

Then we decided to implement the model Linformer to then perform some memory and speed testing applying the model Linformer with a varying parameter k ranging from 1 to the sequence length (n) as we should expect both memory consumption and speed execution to be far lower from the speed and time complexity of a regular transformer model when considering a value of k that is far lower than n. To do so, we also implemented a standard transformer model. The input given to the models are random tensors, to focus on the theoretical performances of the models and to avoid any additional granularity. We can see in Figure 2 that indeed the memory consumption of Linformer is lower than the one of a standard transformer model when k is lower than n, but slowly catches up with the memory consumption of the standard model up to being equal to it when k=n. It is interesting to notice that memory usage is somewhat constant from k=1 to k=80. Thus if our analysis only relied on this metric, it would imply that there would be no need to loose more information by compressing the data onto a $n \times 1$ matrix and that we should instead consider a matrix of size $n \times 80$ or more depending on our limitations.

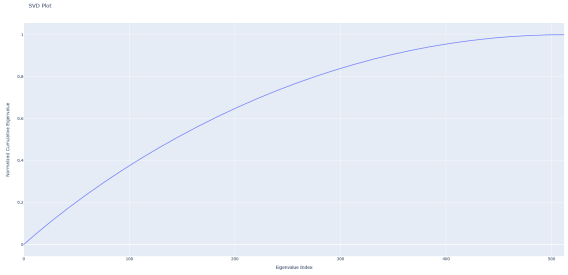
[1] There is a confusion in the article as the given dimension of E_i and F_i was $\mathbb{R}^{\mathbb{k}\times\mathbb{n}}$ in the proof as define as $E_i = \delta R$ and $F_i = e^{-\delta}R$ where $R \in \mathbb{R}^{\mathbb{k}\times\mathbb{n}}$, while being $R \in \mathbb{R}^{\mathbb{n}\times\mathbb{k}}$ in the theorem.



(A) Low rank matrice before softmax



(B) Low rank matrice after softmax



(C) High rank matrice

FIGURE 1. Normalized cumulative sum of the SVD and softmax impact

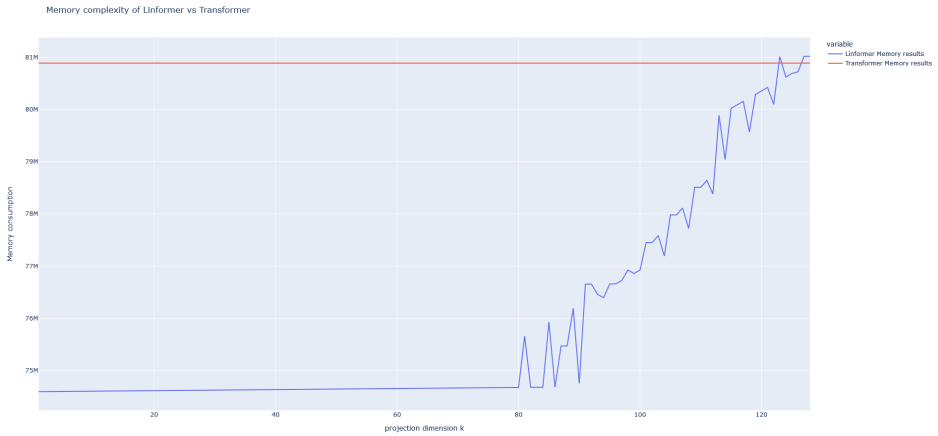


FIGURE 2. Plot representing the evolution the memory allocation of both Linformer and Transformer depending on the value of k.

We also tried to perform a similar testing for speed execution, but we didn't get the expected results, see Figure 3. Indeed, we expected that as in Figure 2 Linformer performance gains of having a lower dimension than a standard transformer would slowly disappear and meet the performances of a standard transformer when k goes to n. This could be explained by a lot of factors, but the main one is that as we are dealing with milliseconds, any extra operation performed when executing Linformer has more impact than the extra computational gain from having an $n \times k$ matrix P to multiply instead of an $n \times n$ matrix, leading to an intrinsic additional computational cost with every execution of Linformer compared to a transformer. However, we still see that clearly Linformer stays more efficient in terms of execution time compared to a transformer. We also would like to mention that the performances

were drastically worse when running the code on collab compared to running it locally and actually showed that Linformer was more time hungry than transformer, which was not the case on a local GPU.

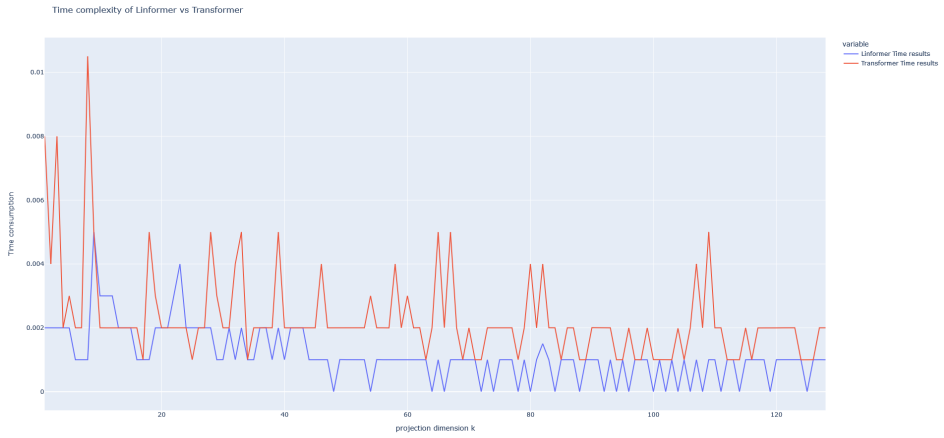


FIGURE 3. Plot representing the evolution execution speed of both Linformer and Transformer depending on the value of k.

Following the previous experiments using random data inputs, we decided to use the IMDB dataset as inputs to compare the results of both experiments types (memory and speed) while training on this real-world dataset. We can see that interestingly even though the memory comparison remains similar as before, the speed comparison is much more insightful as it appears that up to k=109 while n = 256, we do have a gain in speed by using Linformer compared to transformer. The fact that after that threshold the model Linformer is more time consuming could be explained, as before, because the time complexity taken performing the two projections has more impact than the computational gain resulting from lower dimensional matrix products.

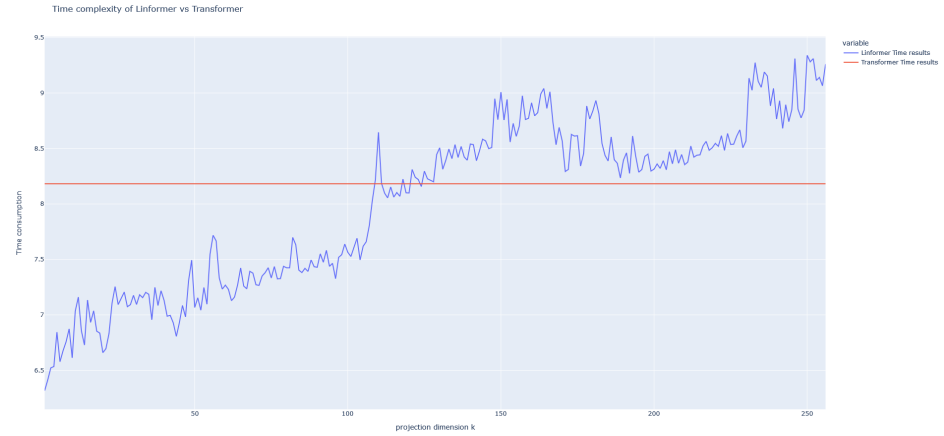


FIGURE 4. Plot representing the evolution of execution speed of both Linformer and Transformer, depending on the value of k, during training using IMDB dataset, adam optimizer with default parameters and learning rate of 0.001.

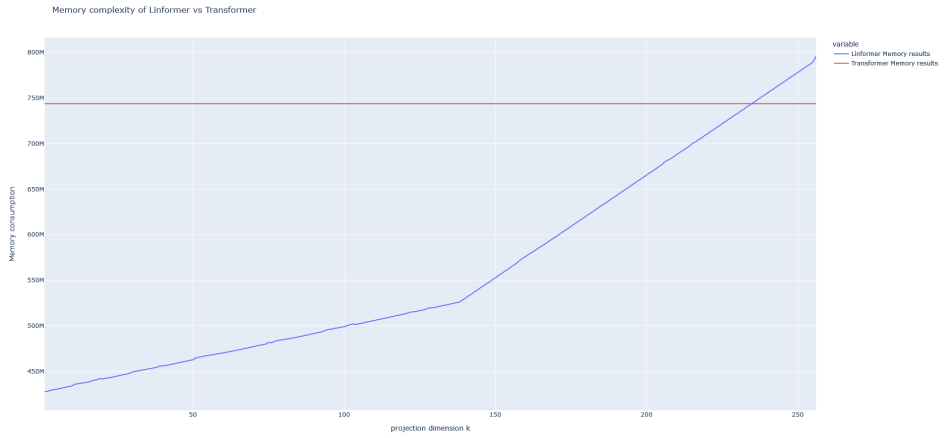


FIGURE 5. Plot representing the evolution of memory allocation of both Linformer and Transformer, depending on the value of k, during training using IMDB dataset, adam optimizer with default parameters and learning rate of 0.001.

We also notice looking at both training and validation losses of Linformer on the IMDB dataset that having a lower projection dimension k helps preventing over fitting, which can be explained as the compression happening here prevents the model from learning the noise from the training set.

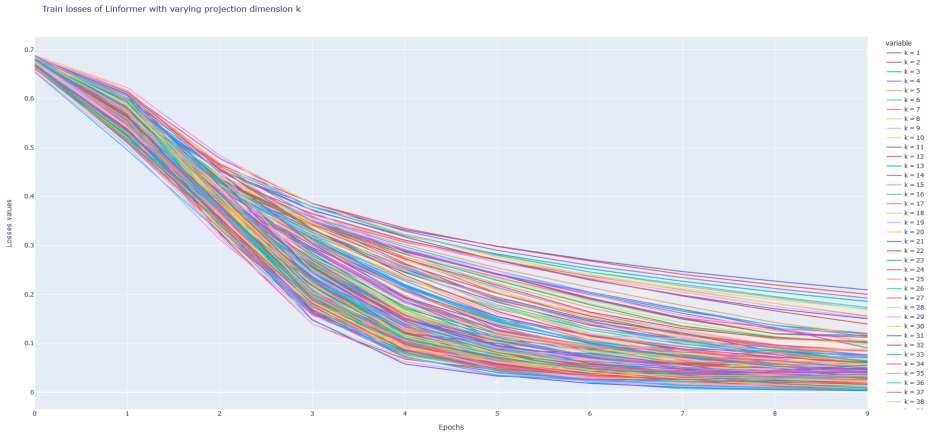


FIGURE 6. Plot representing the training losses of Linformer models with different projection values k, using IMDB dataset, adam optimizer with default parameters and learning rate of 0.001.

3. LIMITATIONS OF LINFORMER

There are few limitations with the model Linformer. The first one being that the idea itself directly implies a trade-off between speed and precision, inherited from a low-rank approximation, even though the SVD isn't directly computed. Second, the idea that it is possible to find a value k that is solely independent from n doesn't really make sense as this would mean that we can compress any matrix P obtained of size $n \times n$ into a matrix of size $n \times k$ with a k constant and independent from n. From this second idea, which is the main problem with this model, we also notice that in the testings of the article of the inference time efficiency for a given projected dimension k=128, the performances

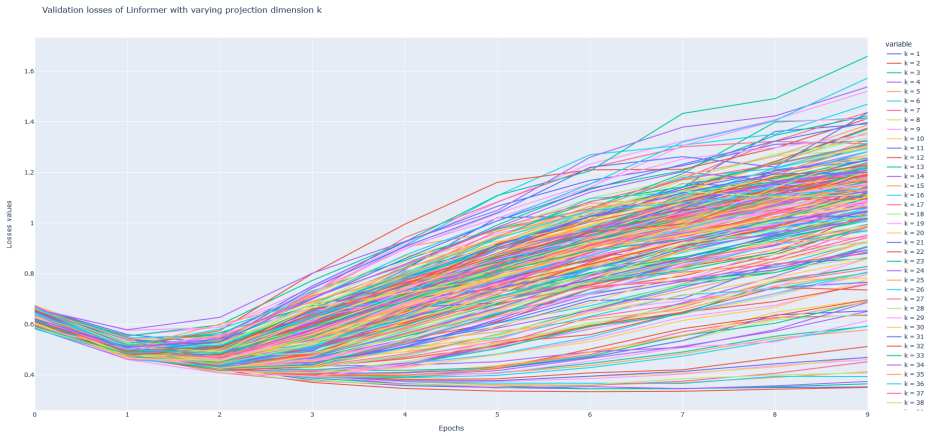


FIGURE 7. Plot representing the validation losses of Linformer models with different projection values k, using IMDB dataset, adam optimizer with default parameters and learning rate of 0.001.

drastically diminish when taking n=512 versus n=4096 or n=65536 as shown in Table 3, and the same goes for the memory saved. Which shows that the ratio n/k is indeed important, even if it was written above that it was not the case. We clearly see that we n goes big, we cannot keep a same constant k and cannot have the same performances that we have with a low value (512 or 1024) of n. We also notice that in the official implementation of the model in the GitHub repository xformers, the default value given to k is n//4, which is somewhat ironic as the whole point of the article is that k should be taken independently from n.

4. Beyond Linformer

In this section, we explore the question, "How can we make transformers work faster in real-world situations?". The Linformer [1] is one of many studies that try to deal with the high amount of calculations needed in the $\mathcal{O}(n^2)$ attention layer of transformers. Other studies, like [3], [4], and [5], try to solve this by making an approximation of the attention matrix. They often do this by looking at self-attention in a way that's similar to how kernel methods theory works. There are also solutions that remind us ideas from Convolutional Neural Networks (CNNs) and use sparse or local attention. You can see examples of this in [6] and [7]. Some recent models from Mistral AI, mentioned in [8] and [9], use a similar method called sliding attention window, first introduced in [10]. These methods are useful because they need less memory and fewer calculations (Floating Point Operations, or FLOPs) to work out the attention. But these methods are still not enough for practical use.

To understand why we face these challenges, we first need to know how modern GPUs (and TPUs, and other similar devices) are built and how they work best[2]. These GPUs are really good at doing lots of calculations quickly, especially matrix multiplication, which they are specially designed to do well. In a usual transformer model, these matrix multiplications are about 99.8% of all the calculations, but they only take up 60% of the time it takes to run the model. This difference makes us want to look more into which parts of the self-attention layer in transformers take the most time. The research in [12], which we'll talk about soon, gives us good information about this. Their paper, specifically Figure 8, shows us what parts of the self-attention process really take the most time.

We find that the matrix multiplication FLOPs, which we have been trying to reduce, actually make up less than 30% of the total time it takes to run the program. This fact makes us think about

[2]We learned a lot about this from the great blog post [11]

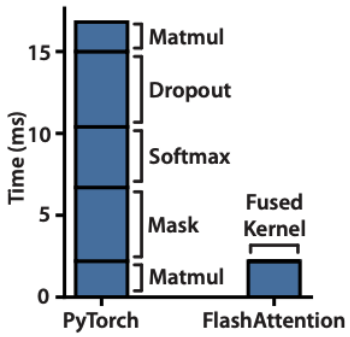


FIGURE 8. GPT-2 Attention computation (from [12])

optimizing our methods by considering how our hardware, like GPUs, actually works. This idea is at the heart of the studies in [12] and its more recent version [13]. These studies are now a key part of training and using all the big, recent models. Before we go into the main ideas of these papers[3], we need to understand how the memory in GPUs is organized. Basically, there are two parts to GPU memory:

- HBM (DRAM): This is the main memory in our GPU. When you get an "Out Of Memory" error in CUDA, it's usually because of this.
- SRAM: This has much faster access times but is smaller in size. It's typically used for cache.

Now, let's look at how the self-attention mechanism in transformers usually uses the HBM. Below is a simplified pseudo code to show its usage on the GPU:

Algorithm 1 Classical Attention Implementation

Load Q, K in HBM, compute $S = KQ^T$, write S to HBM
Read S from HBM, compute $P = \text{softmax}(S)$, write P
Load P and V from HBM, compute $O = PV$ write O in HBM

We notice that there are a lot of 'write/load to HBM" operations, which take up quite a bit of time. This happens because HBM has higher latency than SRAM, as we mentioned earlier. The innovation in [12] is to load the Q, K, V matrices in blocks to the SRAM and combine all the operations like softmax, dropout, and masking. This way, we can avoid the slow step of accessing memory too often. A challenge comes with the softmax operation, but this is addressed in [15] and [16] using an approach called online softmax. Another aspect of Flash attention is the use of gradient recomputation during the backward pass, but we won't go into details about that here.[4]

While this method is used in almost every big model, it's not a one-size-fits-all solution. It needs to be adjusted based on the specific hardware and model we're working with. Often, we end up in what's called the "overhead bound" regime. This means that the GPU spends a lot of time waiting for instructions from the CPU. One way to deal with this is by using cuda graphs. These are explained well in this blog post. Essentially, cuda graphs let us give the GPU more instructions on what to do next, so it doesn't have to wait as much. Other methods include rewriting the code in lower-level languages like C++ or Rust, which can make it run faster. Lastly, there are some "new" architectures to consider, like state space models. Notably, [17] and earlier [18] introduce methods to achieve linear complexity and good performance with these models. They also provide optimized code, making it easier to work with these state space models.

[3]For a really simple explanation of these ideas, check out the blog post [14]
[4]This is still a topic of ongoing research. For more information, check out this discussion

5. Contributions

Yves was responsible for the Introduction, the description of the Linformer model as well as the experiments around the implementation of Linformer including the testing of both memory and speed efficiency and the comparison against a transformer model and the limitation of Linformer. Yves added the experiments on real dataset IMDB after the first submission and presentation. Elie was responsible for the experiment around the low-rank assumption, as well as the last part beyond Linformer. Yves used ChatGPT to debug a part on the synchronization of the kernels while using cuda to run the efficiency experiment of Linformer against transformer. Elie utilized different instruct LLMs, including GPT-4 and Mixtral, to gain a deeper understanding of the topic and to improve the syntax of some sentences. Yves also used the previous work done in course to implement the transformer model. The experiment regarding the comparison of the low-rank property before and after applying softmax comes from Elie and partly reproduces the original experiment from the article Linformer. The experiment regarding memory usage and execution time depending on k comes from Yves.

References

[1] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

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

[3] Zhuoran Shen, Mingyuan Zhang, Haiyu Zhao, Shuai Yi, and Hongsheng Li. Efficient attention: Attention with linear complexities, 2020.

[4] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. *CoRR*, abs/2009.14794, 2020.

[5] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. *CoRR*, abs/2006.16236, 2020.

[6] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020.

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

[8] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[9] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

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

[11] Horace He. Making deep learning go brrrr from first principles. 2022.

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

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

[14] Aleksa Gordic. Eli5: Flashattention. 2023.

[15] Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax, 2018.

[16] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

[17] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2023.

[18] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces, 2022.