Compressive Performers in Language Modelling

Anjali Ragupathi, Siddharth Shanmuganathan, Manu Madhavan
Department of Computer Science and Engineering
Amrita School of Engineering - Coimbatore
Amrita Vishwa Vidyapeetham, India
anjaliragupathi99@gmail.com
siddhsham@gmail.com
m_manu@cb.amrita.edu

Abstract

This work introduces the Compressive Performer, a hybrid Transformer variant based on two existing model architectures: the Performer, which reduces the memory requirement and processing time of the Transformer to linear complexity, and the Compressive Transformer, which retains contextual dependencies over a long range by compressing old activations instead of discarding them. Experiments in language modelling at the character level, the word level, and the sub-word level demonstrate that the Compressive Performer shows improved perplexity scores on the enwik-8 dataset, compared to its base models. This work also compares convolutional compression with autoencoder compression, determining that both show similar perplexity scores.

1 Introduction

Language models were initially based on the relative frequencies of words occurring in a corpus. Traditionally, statistical models like n-grams (Jurafsky and Martin, 2000) and temporal neural models like RNNs (Rumelhart et al., 1986; Hochreiter and Schmidhuber, 1997) were preferred to simulate the structure of natural language. However, they had many deficits like the inability to generalize over unseen words (for n-grams) or to capture information from prior parts of the sentence while taking less time to train(for RNNs).

Recent state-of-the-art deep learning models such as the Transformer (Vaswani et al., 2017) and its subsequent incarnations were created to mitigate these issues. They relied on parallelization techniques that efficiently utilised multiple GPUs - unlike RNNs, which could only use one at a time. Their core feature was "attention", which used $O(n^2)$ operations to find correlations between token pairs for a sequence of n tokens. Although attention had the advantage of large context windows, it also placed constraints on the performance of the Transformer in environments with small memories and few CPUs/GPUs. Since a quadratic number of operations was required ($n \times n$), the Transformer took longer to train. Additionally, each layer of the Transformer stored the output of its activation function in the form of an $n \times n$ matrix, which made the space complexity quadratic as well.

There have been many attempts to mitigate either the space complexity or the time complexity of Transformers using techniques that would reduce the number of computations (Vyas et al., 2020), the amount of data being processed (Li et al., 2016; Panahi et al., 2019), or the amount of information being stored in memory (Child et al., 2019; Katharopoulos et al., 2020; Lan et al., 2019). These were explored in a survey conducted by Tay et al. (2020). Drawbacks of these methods included loss of information due to sparse representations of data (Child et al., 2019), as well as the

uncertainty around the optimal kernel function to be used (Katharopoulos et al., 2020). While models like the Reformer (Kitaev et al., 2020) were effective in reducing the space complexity of the model, they could not adequately capture context over long-range sequences.

Conversely, the Transformer-XL model (Dai et al., 2019) aimed to preserve dependency across long sequences. It introduced relative positional encoding and applied a recurrence relation over segments of data, to extend the length of context dependency learnt by the model.

The success of Transformers sparked the development of two important variants - Compressive Transformer (Rae et al., 2019) and Performers (Choromanski et al., 2020; Likhosherstov et al., 2020). The former was developed as an extension of the TransformerXL which distinguished between long-term and short-term memories. Unlike its predecessor, it compressed old activations instead of discarding them, which allowed extended amounts of context to be preserved with minimal information loss.

In the Performer, a method called FAVOR - Fast Attention Via Positive Orthogonal Random Features - was proposed to scale attention linearly. The attention matrix was decomposed into random features of lower dimensionality, allowing information to be encoded to take up less space than a complete attention matrix. An extension to the algorithm also demonstrated that it was not necessary to construct the complete attention matrix, as the random features could be rearranged to form an approximation that had lower space and time complexities.

Based on the advantages demonstrated by the above models, this work proposes the Compressive Performer-a hybrid model which combines the linear space and time complexity of the Performer with the long-range context-sensitivity of the Compressive Transformer. The empirical performance of this integrated model is compared with the original models on which it is based. The goal is to preserve the low space complexity shown by the Performer without losing any valuable contextual information over a long-range sequence. To demonstrate the consistency of the model over different use cases, this work also involves tokenization of the dataset at three different levels of natural language - word, sub-word, and character.

2 Proposed Model

This section briefly describes the FAVOR+ algorithm implemented in the Performer (Choromanski et al., 2020), along with the two compression techniques used in the proposed model. Further, the integration of compression with FAVOR is discussed in Section 2.2.

2.1 FAVOR+ - Fast Attention

Fast attention was developed as an alternative to multiplicative attention / scaled dot-product attention(Luong et al., 2015) and Bahdanau's attention (Bahdanau et al., 2014). To bring down the space complexity for the model, the complete attention matrix is never constructed. Instead, the softmax function is approximated by using suitable kernel functions which make use of "random orthogonal features". Using this approximation, the original Q and K matrices cannot be reconstructed, but similar matrices of the same dimension ($L \times d$, where L is the input sequence length and d is the inner dimensionality) can be approximated.

The original attention formula is defined in Equation 1, while its corresponding FAVOR approximation is defined in Equation 2.

$$Attn(Q, K, V) = softmax(\frac{Q \cdot K^T}{\sqrt{d_k}}) \cdot V \quad (1)$$

where
$Attn$ = Attention Function

Q = Query Matrix
K = Key Matrix
V = Value Matrix
$\sqrt{d_k}$ = Scaling factor of dimension k

$$Attn(Q, K, V) = Q' \cdot ((K')^T \cdot V) \quad (2)$$

where
$Attn$ = Attention Function
Q' = Approximated Query Matrix
K' = Approximated Key Matrix
V = Original Value Matrix

2.2 Compression algorithm integrated with FAVOR

The algorithm used in the Compressive Performer is based on the original paper on the Compressive Transformer (Rae et al., 2019). It uses a FIFO queue as the data structure in which the attention weights at each memory layer are stored (Wang, 2020). This allows a time-independent Transformer to access "memories" in a definite temporal order. The queue is split into the short-term memory (*mem*) and the long-term memory (*cmem* for compressive memory). At each time step, after the attention weights have been calculated, they are pushed into the FIFO queue's short-term memory. Once this section is filled, the short-term memory data is compressed and pushed into the long-term memory, and the complete memory queue is updated. During backpropagation, the gradients used for the compressive network itself are not propagated into the main network and are instead used to improve reconstruction loss. However, since the memory layers are generally hidden layers, the stored activations are used to update the weight matrices on subsequent passes through the network, thus preserving context.

The proposed work compares two compression algorithms. The first is basic convolutional compression, which is generally used in images. It uses convolutional layers to extract features from data, and pooling layers to reduce the dimensions of data by discarding unnecessary features (as in Equation 3). When text data is represented in the form of vectors or tensors, the underlying numerical representation of language structure can also be compressed in a way similar to images (Mahoney, 2000; Cox, 2016; Goyal et al., 2018). The second type of compression is autoencoder compression. An autoencoder is an artificial neural network that can learn an encoding for data by training itself to ignore "noise". This encoding is a projection of the input data in latent space, typically of lower dimensionality than the original data. The proposed work uses convolutional autoencoders, which compress data with some loss of information. Hence, Mean Squared Error loss is used as the evaluation metric for reconstructing the original input from its encoded counterpart.

$$dim' = (dim - f)/(s + 1) \quad (3)$$

where dim' is the new length or width of image, dim is the original length or width of image, f is the filter dimension and s is the stride length.

While integrating the compression algorithm with FAVOR, no complete attention matrices are constructed, as in ordinary Performers. Pre-normalization is used at each layer instead of post-normalization, as described by Nguyen and Salazar (2019). Since backward propagation for the computation of attention reconstruction loss is nearly identical to the one described by Rae et al. (2019), its discussion remains beyond the immediate scope of this paper.

In Algorithm 1, the batch size is represented by b, the embedding dimensions by d, the length of the memory by l_{mem}, and the length of the compressed memory by l_{cmem}. The variable $h^{(i)}$ represents the hidden state at layer i. At time step 0, the memory queue is initialized to be empty. At each succeeding time

step, the subsequent input is extracted using random sampling. It is then passed to the embedding layer, where the embedding weights are calculated and relative positional encoding is applied to get the first hidden state (Dai et al., 2019). The long-term compressed memory and the short-term uncompressed memory are concatenated with the generated hidden state to form the queue. By applying a suitable projection function (for example, a simple linear function or multi-layer perceptron), the input is converted into the query, key, and value matrices. The FAVOR+ algorithm is then applied to these matrices as in Equation 2. Here, matrix associativity is applied to rearrange the three matrices, ensuring that the output of the attention function has linear complexity. Residual connections are made to prepare the model for the backward pass of training. In the compression phase, the oldest memories are extracted and compressed by the specified compression ratio c. The current hidden state is pushed into the short-term memory, while the compressed memories are pushed into the long-term memory. Per convention, normalization is applied to the output activations to speed up training time by making the gradients of the network stable. The next hidden state is generated and propagated to the following layers.

3 Experiments

3.1 Pre-processing and Tokenization

Raw data (a subset of the enwik8 dataset - originally curated by Mahoney (2006)) was read from its source file and pre-processed. The training and evaluation phases were tested at all three levels of tokenization. The number of bytes read from the input file had to be decreased from the character level model having the highest number (95 MB), to the word level having the lowest (25 MB). This was done because the large vocabulary of word level models would impede execution by creating many parameters, thus resulting in memory mismanagement.

Pre-processing included removing XHTML tags, delimiting sentences, and tokenizing them based on the level of natural language considered. At the character level, it was enough to split the data into separate characters. At the sub-word level, it was essential to identify the most common morphemes. While stemming can be used for this purpose, it remains inefficient when quick processing is desired. A Byte-Pair Encoding Tokenizer from HuggingFace was, thus, used to build common subwords based on the frequency of co-occurrence of characters. At the word level, a simple word tokenizer from the NLTK library was incorporated; it was complemented by a multi-word tokenizer to add delimiters to the vocabulary.

Post-tokenization, a vocabulary of the appropriate size was built and passed as a parameter to the learning algorithm. At the character and sub-word levels, the data was split into training, validation, and testing using a 90:5:5 split. For the word level model, the data was split using an 80:10:10 split, as per convention.

3.2 Training and Tuning

The model was trained using Kaggle Kernels on a Tesla P100 GPU with 16 GB of memory. At each iteration, gradient clipping was done to prevent exploding gradients. Perplexity (Equation 5) was calculated as an exponent of validation cross-entropy loss (Equation 4). Further discussion on suitable metrics for language models can be found in the article by Huyen (2021).

Another mechanism that was implemented was a learning rate scheduler (Rath, 2021). The implementation scaled down the learning rate by a factor of 0.5 if it observed no improvement in the validation loss for at least 5 epochs in a row. Early stopping (Rath, 2021) was

Algorithm 1 Proposed Algorithm: Compressive Performer Main Forward Propagation

At time t_0

$mem_0 \leftarrow 0$ ▷ Shape of memory is b x l_{mem} x d
$cmem_0 \leftarrow 0$ ▷ Shape of compressed memory is b x l_{cmem} x d
for t in 1, 2, ... $n_{timesteps}$ **do**
  $h^{(1)} \leftarrow xW_{emb}$ ▷ W_{emb} is the weight matrix for embeddings, x is the input tensor
  for i in 1, 2, ... n_{layers} **do** ▷ n_{layers} represents the number of memory layers
    $fifo^{(i)} \leftarrow concat(cmem_t^{(i)}, mem_t^{(i)}, h^{(i)})$ ▷ Create FIFO queue with both memory queues and input
    $q^{(i)}, k^{(i)}, v^{(i)} \leftarrow projection(fifo^{(i)})$ ▷ Extract Q, K, V matrices
    $a_{favor}^{(i)} \leftarrow FastAttention(q^{(i)}, k^{(i)}, v^{(i)})$ ▷ Apply FAVOR on the queue
    $a^{(i)} \leftarrow a_{favor}^{(i)} + h^{(i)}$ ▷ Apply residual connection
    $old\_mem^{(i)} \leftarrow mem_t^{(i)}[: n_{oldest}]$ ▷ Extract oldest memories to be compressed (n_{oldest} x d)
    $new\_cmem^{(i)} \leftarrow f_c^{(i)}(old\_mem^{(i)})$ ▷ Compress oldest memories by factor c ($\frac{n_{oldest}}{c}$ x d)
    $mem_{t+1}^{(i)} \leftarrow concat(mem_t^{(i)}, h^{(i)})[-l_{mem} :]$ ▷ Push current input into short-term memory
    $cmem_{t+1}^{(i)} \leftarrow concat(cmem_t^{(i)}, new\_cmem^{(i)})[-l_{cmem} :]$ ▷ Push new compressions into long-term memory
    $h^{(i+1)} \leftarrow pre\_norm(linear(a^{(i)})) + pre\_norm(a^{(i)})$ ▷ Generate next hidden state
  end for
end for

also included as a mechanism to stop training when the validation loss showed no sign of improvement. This was done in an attempt to avoid overfitting, based on suggestions in the paper (Komatsuzaki, 2019). Finally, the Adam optimizer (Kingma and Ba, 2014) was used because of its advantages over stochastic gradient descent. The addition of these optimization techniques reduced training time significantly from a few hours to a minute.

Using various tunable parameters like compression ratio, batch size, memory size, and sequence length, the learning algorithm iteratively calculated a set of weights that would minimize the cross-entropy loss and the auxiliary MSE loss which is related to the reconstruction error after compression. In addition to these parameters, the highest batch size that showed the least error was found to be 16 batches. A model depth of 6 was also found to be better suited to the size of the dataset than a model depth of 8. Most hyperparameter values have been taken from the original paper on Compressive Transformers (Rae et al., 2019), such as sequence length and memory length being 768, compressed memory length being 1152, and compression ratio being 4.

$$Loss_{CE} = -\sum_{i=1}^{k} y_{o,i} \log_2 p_{o,i} \tag{4}$$

where
$Loss_{CE}$ = Cross Entropy Loss
k = number of classes
$y_{o,i}$ = binary value which tells if observation o was correctly categorized as class i (true value)
$p_{o,i}$ = predicted probability of observation o being in class i

$$Perplexity = 2^{Loss_{CE}} \tag{5}$$

4 Results and Discussion

The two baseline models that have been considered for comparison and inference are the Compressive Transformer (Rae et al., 2019) and the Performer (Choromanski et al., 2020). The Performer has been fitted with reversible layers as described in the paper by (Kitaev

et al., 2020), based on the implementation by (Wang, 2021).

Each model has been studied at the character level, sub-word level, and word level. Vocabulary size was found to have increased, with *n(character)* $<$ *n(sub-word)* $<$ *n(word)*. This was primarily because the set of characters in ASCII comes out to 256 tokens, whereas the number of unique words is significantly larger. Sub-words may share morphological stems of words.

4.1 Character Level

When the models were trained at a character level, they had access to 256 possible classes or categories. Since this number was significantly lower than the vocabulary size at the other two levels, the cross-entropy loss function mentioned in Equation 4 had to compute the summation over fewer classes. Hence, the training and validation losses were part of a lower range of values. However, there was significant volatility in the losses as seen in Figure 1a, because the model could not learn relationships between characters in a structurally or semantically cognizant manner. However, the model did learn which characters occur together frequently. The losses remained much lower than that of the Performer and the Compressive Transformer, despite starting at similar points; this showed that the Compressive Performer tended to have a much better record of achieving a relevant prediction compared to the baseline models, which could be attributed to its ability to learn complex context dependencies.

It was also noted that all the character level models were highly unstable, but their curves plateaued after a certain elbow point. Since character level models had not implemented early stopping, the graphs in Figure 1a indicate that this would be a good strategy to follow in subsequent implementations.

4.2 Sub-word Level

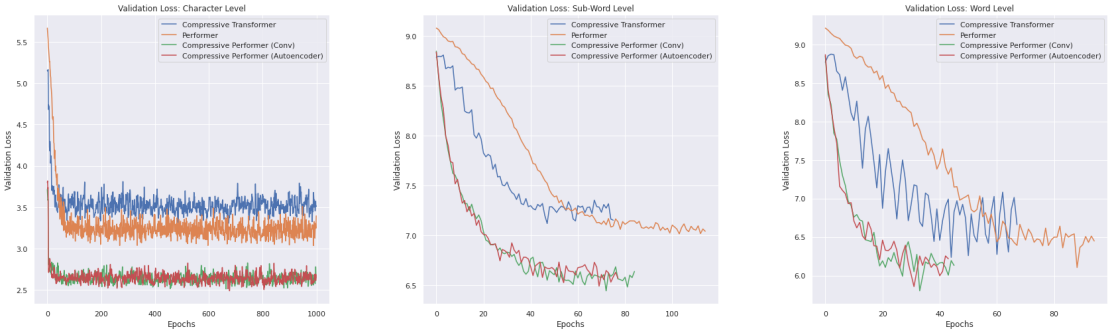
At the sub-word level, the training losses of both versions of the Compressive Performer were highly volatile, indicating that the number of classes was not easily determined. This was due to the numerous morpheme combinations that could potentially be predicted next. The model thus struggled to predict the best possible class consistently. Still, it can be seen in Figure 1b that the slope of the graphs was much steeper and decreased a lot more than for the baseline models. This showed that the Compressive Performer learned much faster and also more uniformly than both the Compressive Transformer and the Performer.

4.3 Word Level

At the word level, the loss graphs were much less volatile than at the sub-word level because of the relative ease of predicting a class. With a fixed-size vocabulary like that used in the word level model, the loss curves decreased with much less fluctuation. It can be seen in Figure 1c that the Performer still found predictions difficult because it could not preserve context dependencies as easily as the three compressive models.

4.4 Comparison and Inference

From the results, it can be seen that the proposed model could utilize compression algorithms without incurring large space overheads; thus, the space complexity of the Compressive Performer was closer to that of the Performer than to the Compressive Transformer. A comparison of both Compressive Performer variants with their baseline models is shown in Tables 1, 2 and 3. It is clear that the Compressive Performer has outperformed the baseline models at all three levels of experimentation, by achieving the lowest relative perplexity. Though the GPU utilization of the proposed model was greater than that of the Performer



(a) Character Level (b) Sub-word Level (c) Word Level

Figure 1: Cross-entropy Loss over Validation

Cross-entropy loss of Compressive Performer is much lower than the base models at all three levels. Differences in graph lengths in Figure (1b) and Figure (1c) are due to early stopping. Steep curves imply faster learning. High jitter implies greater uncertainty in predicting the next token (seen in all four models in Figure (1a) and in Compressive Transformer in Figure (1c))

Table 1: Performance Comparison of Models - Word Level

Model Name	Prediction Time(ms)	GPU RAM(GB)	Training Perplexity	Test Perplexity
Compressive Transformer	70.7	6.9	173.45	102.36
Performer	170	3.1	220.32	109.334
Compressive Performer (Conv)	**67.7**	**3.6**	**110.2**	**75.896**
Compressive Performer (Auto)	**57**	**3.6**	**111.69**	**75.141**

Table 2: Performance Comparison of Models - Sub-Word Level

Model Name	Prediction Time(ms)	GPU RAM(GB)	Training Perplexity	Test Perplexity
Compressive Transformer	32	6.9	209.69	179.28
Performer	140	3.2	230.93	171.316
Compressive Performer (Conv)	**55.375**	**3.5**	**128.4**	**127.82**
Compressive Performer (Auto)	**55.05**	**3.5**	**133.32**	**126.727**

Table 3: Performance Comparison of Models - Character Level

Model Name	Prediction Time(ms)	GPU RAM(GB)	Training Perplexity	Test Perplexity
Compressive Transformer	96	8.6	11.61	13.078
Performer	166	4.7	9.57	9.3378
Compressive Performer (Conv)	**51**	**7.2**	**6.26**	**6.4314**
Compressive Performer (Auto)	**51**	**6.8**	**6.295**	**6.025**

due to the inclusion of queues, it was considerably less than that of the Compressive Transformer.

At the word and sub-word levels, the Performer fared the worst at capturing semantic and contextual dependencies over long sequences. The Compressive Transformer used the highest amount of GPU RAM because of its dependence on traditional softmax attention, as opposed to the linear attention mechanism preferred by the three Performer variants.

At the character level, it was noted that

the activations computed by the convolutional compression algorithm took up slightly more RAM than those computed using autoencoder compression. The comparable training perplexity scores in Table 3 also showed the effectiveness of autoencoders in learning contextual representations from compressed data. The low perplexity scores compared to the word level and sub-word level models were due to the fact that there was a very small set of tokens (256 ASCII characters) that the model could predict. Additionally, the amount of GPU RAM used by the convolutional model was slightly higher than that used by the autoencoder model.

When the models were tested based on their prediction time (i.e., from input submission to output prediction), it was found that the Compressive Performer took much less time to predict the next tokens in a sequence at all three levels of tokenization. These results are demonstrated in the column "Prediction Time" in Table 1, Table 2, and Table 3. The models were also generalizable over new data and overfitted less than the base models, as demonstrated in the corresponding column "Test Perplexity".

Overall, the models proposed in this paper have yielded promising results in terms of space complexity and perplexity scores. Despite the use of random sampling, the perplexity scores of both Compressive Performers were observed to be consistently lower than the base models, while small variations were observed in the ranking of the base models themselves.

5 Conclusion

The goal of this work was to be able to train Transformer models on consumer-grade devices, without resorting to expensive cloud services and specialized hardware. It was determined that the Compressive Performer showed comparable space complexity to the Performer and maintained perplexity scores that were much lower than the base models. It was also determined that the autoencoder compression mechanism offered similar results to the convolutional compression mechanism. In addition to this, it was seen that the proposed model performed equally well on both training and testing data when compared to existing state-of-the-art models.

However, the model showed major limitations in the quality of text generated. Owing to a scarcity of suitable high-performance computing resources, the experiments in this paper were conducted with minimalistic models and a small dataset. While it is expected that the results can be replicated on a large scale, it would be necessary to attempt training the model from scratch using a benchmark dataset like PG-19 (Rae et al., 2019). Experiments may also be done with custom tokenizers and cleaning algorithms, as well as with other kernels and activation functions, to determine what results arise from them. Pre-training approaches like ELECTRA (Clark et al., 2020) could be implemented to help the model learn context adversarially.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*.

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

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Elec-

tra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

David Cox. 2016. Syntactically informed text compression with recurrent neural networks. *arXiv preprint arXiv:1608.02893*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. 2018. Deepzip: Lossless data compression using recurrent neural networks. *arXiv preprint arXiv:1811.08162*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Chip Huyen. 2021. Evaluation metrics for language modeling. https://thegradient.pub/understanding-evaluation-metrics-for-language-models/.

Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st edition. Prentice Hall PTR, USA.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning*, pages 5156–5165. PMLR.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.

Aran Komatsuzaki. 2019. One epoch is all you need. *arXiv preprint arXiv:1906.06669*.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

Xiang Li, Tao Qin, Jian Yang, and Tie-Yan Liu. 2016. Lightrnn: Memory and computation-efficient recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 4385–4393.

Valerii Likhosherstov, Krzysztof Choromanski, Jared Davis, Xingyou Song, and Adrian Weller. 2020. Sub-linear memory: How to make performers slim. *arXiv preprint arXiv:2012.11346*.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Matt Mahoney. 2006. Enwik-8 source version. https://cs.fit.edu/~mmahoney/compression/textdata.html.

Matthew V Mahoney. 2000. Fast text compression with neural networks. In *FLAIRS conference*, pages 230–234.

Toan Q Nguyen and Julian Salazar. 2019. Transformers without tears: Improving the normalization of self-attention. *arXiv preprint arXiv:1910.05895*.

Aliakbar Panahi, Seyran Saeedi, and Tom Arodz. 2019. word2ket: Space-efficient word embeddings inspired by quantum entanglement. *arXiv preprint arXiv:1911.04975*.

Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. 2019. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*.

Sovit Ranjan Rath. 2021. Using learning rate scheduler and early stopping with pytorch. https://debuggercafe.com/using-learning-rate-scheduler-and-early-stopping-with-pytorch/.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Apoorv Vyas, Angelos Katharopoulos, and François Fleuret. 2020. Fast transformers with clustered attention. *arXiv preprint arXiv:2007.04825*.

Phil Wang. 2020. Compressive transformer implementation pytorch. https://github.com/lucidrains/compressive-transformer-pytorch.

Phil Wang. 2021. Performer implementation pytorch. https://github.com/lucidrains/performer-pytorch.