PERFORMER BASED LANGUAGE MODEL

Dobiš Lukáš
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
xdobis01@stud.fit.vutbr.cz

Karabelly Jozef
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
xkarab03@stud.fit.vutbr.cz

Marek Vaško
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
xvasko16@stud.fit.vutbr.cz

6.5.2021

1 Introduction and related work

Language Modeling (LM) is one of the numerous essential parts of modern Natural Language Processing (NLP). Language modeling is the task of assigning a probability to sequences in a language. Apart from assigning a probability to each sequence of words, the language model assigns a probability for a given word or a sequence of words to follow a sequence of words [3]. Our specified task is to train such model, and generate best text possible with lowest perplexity (LM models quality measure). We are required to use novel techniques beyond basic vanilla RNN methods.

To narrow scope of related works to our solution, we will focus only on breakthrough papers based mainly around attention blocks. Transformers have provided state-of-the art results within this field for past few years. This architecture has been firstly proposed in [5], and was since basis of modern LM. Attention mechanism is crucial part of these models as the most important component for learning long range dependencies in text delivering better results than statistical methods, while avoiding causal training constraint of Recurrent Neural Networks (RNN). GPT [4] has been one of the first LM which demonstrated usage of this architecture. GPT was was at the time one of the best LM models to date. Achieving superior results thanks to having massive amount of parameters and largest training data set at the time. BERT [2] has later introduced usage of bidirectional attention while also introducing training processes different to GPT. BERT analyzes sequence from both direction to get both left and right context, which achieves state-of-the-art results for range of tasks. The limitations of these networks have lead to development of different approaches to attention mechanism itself, one of which is called Performer [1]. Performer improves to linear time and space complexity from quadratic complexity of classical attention.

2 Transformer architecture

Transformer architecture proposes two interlinked neural networks. The roles between two networks are split into encoder and decoder. The role of the encoder network is to create a embedding of the input sequence. This embedding among with beginning of target sequence is later used as input for decoder network.

Both of these networks are similar in nature, since both use multi-head attention. The main difference between the two is within usage of masked attention. Masking provides a way for attention mechanism to only focus on the words which appeared before a certain word, making it look only to the past. This mechanism is used for decoder network since it cannot know the relations to words which will be generated. The encoder on the other hand can look at the sentence as a whole, deeming masking not necessary.

Depending on the specific case, language models use one or another attention mechanism. Our specific implementation uses a decoder network similar to GPT model used in [4]. Except of the obvious architectural difference the decoder based language models differ in the way they are being trained. Our model is trained on the input sequence with a desired output offset by one symbol into the future. This way the model trains to predict next word given words before it. Masked attention or self-attention makes is in this case beneficiary since future words will be predicted and thus are not known. The overall architecture of used decoder layer can be seen on Figure 1.

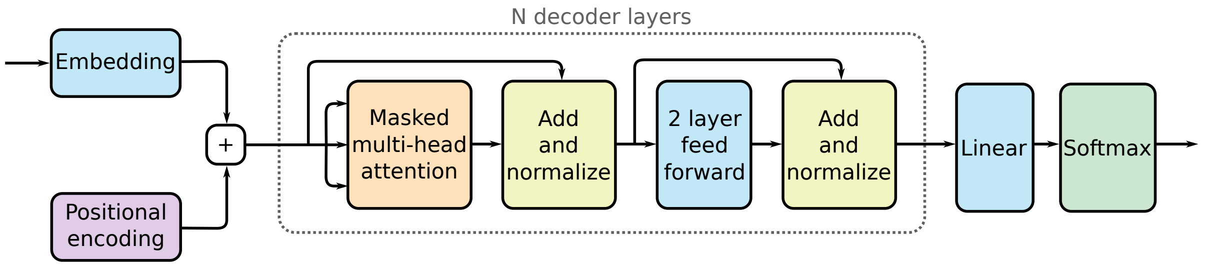

Figure 1: Language model architecture

3 Multi-head attention mechanism

A multi-head attention, is a extension of simple attention where different parallel instances of attention layer focus on different dimension of input. The input is in this case split into predefined number of instances, which equals to the number of heads. The dimension of single instance can be then determined as $d_{head} = \frac{d_{model}}{n_{head}}$, where d_{model} is number of dimensions provided by embedding layer and n_{head} is number of heads in multi-head attention mechanism. To make multi-head processing effective, one more dimension is added input. Output is later flattened.

The input processing is firstly preceded by three parallel linear layers. Reason for these layer is to create three different input sequences described as $Q, K, V \in \mathbb{R}^{s_{len} \times d_{head}}$, where Q are considered to be queries, K keys and V values. The method first computes how well certain key reflects the query, creating a matrix with size of $s_{len} \times s_{len}$ in process. The matrix is then passed through scaling and masking layers. Results are then passed through a soft-max function thus creating weight matrix A. Values matrix multiplied by A matrix give results in the form of best matched values according to given queries and keys. Diagram of single head can be seen on Figure 2.

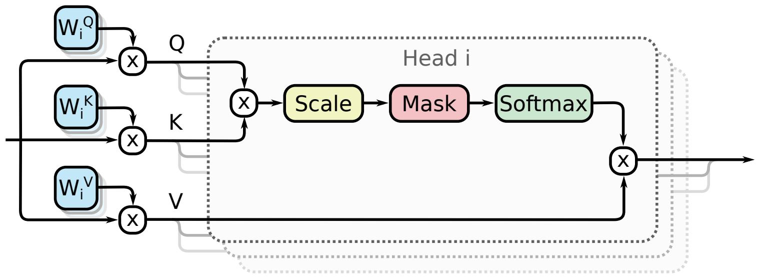

Figure 2: Multi-head attention

As can be seen the multi-head attention layer has by itself three trainable weight matrices $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{model} \times d_{head}}$. Our implementation combines these matrices for all heads into three matrices for entire multi-head attention mechanism so that $W^Q, W^K, W^V \in \mathbb{R}^{d_{model} \times d_{model}}$. The split for different heads is done with another added dimension and view of returned data. However in later equations only original way is used. Attention for single head is formally described in the Performer publication as follows:

$$Att(Q, K, V) = D^{-1}AV,\ A = exp(QK^T/\sqrt{d_{head}}),\ D = diag(A1_{s_{len}}), \tag{1}$$

where D represents sum of rows within a A matrix in a diagonal matrix form, and thus D^{-1} represents normalization for a softmax function. Our reference implementation uses a soft-max directly and does not compute n this way. Matrix $D^{-1}A \in \mathbb{R}^{s_{len} \times s_{len}}$, contains weights described in previous section.

As can be seen the complexity of this computation is highly depended on the s_{len}, where the time complexity to compute matrix AV can be described as $O(s_{len}^2 d_{head} + s_{len}^2 d_{head})$, thus having quadratic time but also space complexity.

4 Linear time complexity using a FAVOR+

The quadratic time complexity is where authors of [1] propose a effective and accurate linearization of attention mechanism. The main idea of this approach is to utilize fact that if soft-max function used to calculate weights could be approximated using matrix multiplication the attention equation could be rewritten as:

$$Att(Q, K, V) = \hat{D}^{-1}(Q'((K')^T V)),\ \hat{D} = diag(Q'((K')^T 1_{s_{len}})) \tag{2}$$

The time complexity is thus reduced since multiplication of matrices $Q'((K')^T V)$ has time complexity of $O(s_{len} d_{head}^2 + s_{len} d_{head}^2)$, thus making it linear, in relation to s_{len} but quadratic to d_{head}. This promises better performance among longer input sequences, which may not be possible due to computational limitation with classical attention mechanism.

This publication suggests an innovative method of creating projections Q' and K', so accuracy of attention mechanism stays. Previous methods have tried similar approaches but often failed to deliver needed accuracy in specific values. These inaccuracies lead to performance incomparable to classical attention mechanism. Estimation of attention is done via positive orthogonal random features and thus the name *Fast Attention Via positive Orthogonal Random features (FAVOR+)*.

The features are a result of projection of original values of Q and K into space where multiplication results in soft-max estimation. The general approach can be described as:

$$\phi(x) = \frac{h(x)}{\sqrt{m}}(f_1(\omega_1^T x), \dots, f_1(\omega_m^T x), \dots, f_n(\omega_1^T x), \dots, f_n(\omega_m^T x)) \tag{3}$$

Where projection $\phi(x)$ of vector x of given dimension d, is describes as multiple projection using function f of inner dot product of x with different random vectors ω. The number of vectors ω and number of functions f is by general definition arbitrary. The specific instance of such projection is proposed. Authors propose usage of vectors randomly drawn from normal distribution $\omega_1, \dots, \omega_m \sim \mathcal{N}(0, I_d)$, where I_d is identity matrix of dimension d. In addition vectors $\omega_1, \dots, \omega_m$ are orthogonalized using Gam-Schmidt method. Our implementation uses qr method available in *PyTorch* framework. The number of vectors is arbitrary an can be even larger than d, but organization need to be done on multiple slices of size d and later concatenated into single matrix, for effective projection computation.

Authors propose usage of a one function f which is simply described as $f(x) = exp(x)$. The $h(x)$ is proposed as:

$$h(x) = \frac{1}{\sqrt{2}} exp(-\frac{||x||^2}{2}) \tag{4}$$

Our implementation wraps whole projection into single method which on call will produce projected features for given batch of input sequences. It uses a soft-max kernel described in previous section with function and orthogonal features created according to authors recommendation. Additionally authors recommend so called *redrawing* of orthogonal random features which is done every 4000 iterations.

Multiplication of projected features can produce directly a bi-directional attention mechanism. Self-attention used for our language model needs a different approach. The method of masking used in classical multi-head attention cannot be used directly. This is mainly because this approach is not producing the A matrix an thus it cannot be masked. Authors provide a solution to this problem which introduces the use of prefix sums. Two method are described in implementation provided by authors and they describing the process of creating masked attention. First one is used to create so called casual numerator, or the $Q'(K')^T V$ matrix.

Algorithm 1: CasualNumerator

```
Result: Return numerator for self-attention
$result \leftarrow EmptyList$;
$S \leftarrow ZerosMatrix(d_{head}, d_{head})$;
for $i$ in $Range(0, s_{len})$ do
    $S \leftarrow sums + (K')[i]((V')[i])^T$;
    $result.append(S\ (Q')[i])$;
end
return $result$
```

The Algorithm 1 uses $X[i]$ to index i-th item in the sequence. The dimension of items within V and Q are considered to be equal. The other method creates normalizations for values created in Algorithm 1, hence the name casual denominator.

```
Algorithm 2: CasualDenominator
Result: Return denominator for self-attention
$result \leftarrow EmptyList$;
$S \leftarrow ZerosMatrixLike((K')[0])$;
for $i$ in $Range(0, s_{len})$ do
    $S \leftarrow sums + (K')[i]$;
    $result.append(InnerDotProduct((Q')[i], S))$;
end
return $result$
```

A reference implementation in *Python*, which is given by author and is described above differs in some cases from original *FAVOR+* defined in the paper. The differences are however formal and are mainly in the usage of C matrix instead of V which is equivalent to $C = [V\ 1_{s_{len}}]$. This makes it possible do discard Algorithm 2, because added 1 will make equivalent role to $InnerDotProduct$. Using this approach it is however necessary to later split results into numerator and denominator parts. As for original algorithm the resulting attention can be simply computed as:

$$Att(Q, K, V) = diag(CasualDenominator)^{-1} CasualNumerator \tag{5}$$

5 Implementation

Our implementation contains both method for providing attention the classical multi-head attention presented in Section 3 and linear attention mechanism described in Section 4. The implementation environment is *Python* wih a version *3.8+*. We have chosen *PyTorch* framework accompanied by *PyTorch Lightning* in order to provide efficient training and inference environment. The whole implementation can be seen in our *GitHub repository*[1].

The implementation is bases out of multiple sources which were used for reference. The transformer network architecture used for purposes of language modeling is loosely based out of implementation provided directly in *PyTorch* with module `TransformerDecoder`. The source code of which can be seen in *PyTorch GitHub repository*[2]. The network has been re-implemented from ground up to suit specific needs of language model. The original decoder network is suitable for sequence to sequence modeling and thus requires input of values. In order to make it so values are not needed and are taken from inputs of the decoder layer, the provided architecture had to be changed. Modification also include the possibility to switch out classical masked attention with performer attention, which was not possible with original setup.

The authors of performer provided implementation of their attention mechanism in one of two *Python* frameworks which include *TensorFlow* and *JAX*, neither of which are directly compatible with *PyTorch*. Implementation in both frameworks can be seen in *Google Research GitHub repository*[3]. The whole mechanism was re-implemented with base of implementation being reference provided by *Google Research*. We have re-implemented only parts of the original which were needed for self-attention, the rest of the model was created in a way to be interchangeable with classical multi-head attention. Methods used internally within the modules are one-to-one equivalent of *TensorFlow* method provided in original but with according *PyTorch* method calls and initializations.

PyTorch by itself does not provide any way to create positional encoding. Some specific instance use `Embedding` layer in order to provide this functionality, however in our case this approach resulted in high measures of perplexity and low convergence rate of the model. After this we resulted in using positional encoding provided in *Sequence-to-Sequence PyTorch tutorial*[4]. This implementation is equivalent to proposal found in original *Attention Is All You Need* paper [5].

The main language model resides within `models` directory and is part of the *transformer.py*. The main class for creating instance of language model is `LMModel`. The class defines configuration of optimizer but also training and and validation steps which is a possibility with usage of *PyTorch Lightning*. The model itself can be initiated using given list of hyper-parameters:

[1]`https://github.com/karabellyj/KNN2021-LM`
[2]`https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/transformer.py`
[3]`https://github.com/google-research/google-research/tree/master/performer`
[4]`https://pytorch.org/tutorials/beginner/transformer_tutorial.html`

- `vocab_size` - Size of vocabulary used, this selection determines dimension of last linear layer, and initialization of word embedding layer at the front of the network
- `d_model` - Number of dimensions used to embed symbols
- `n_layers` - Number of sequentially arranged decoder networks
- `n_heads` - Number of heads within multi-head mechanism
- `d_ff` - Dimension of center network within 2 layer feed forward network used inside of decoder layer
- `embd_pdrop` - Dropout rate used for embedding layer *(currently not implemented)*
- `attn_pdrop` - Dropout rate used after attention layer
- `resid_pdrop` - Dropout rate used within residual connections inside of decoder layer
- `pad_id` - Identifier of symbol which is used for padding purposes
- `attention` - Choice of attention layer used within decoder network

In order to train the models a module providing data is needed to be implemented. The implementation of this module is within `data.py`. The data module is responsible for processing and supplying the processed data set for purposes of training and validation. The module is coded to provide *wikitext-103* data-set. It will firstly check the existence of tokenizer files within `data` directory. If they are not present the script will *train* Byte-wise byte-pair encoding tokenizer from loaded data-set. The tokenizer is then used to tokenize data-set. Used super class provides standard interface used with trainer to provide *PyTorch* data loader for training, validation and testing data-sets.

Model can be trained through *Jupyter* notebook provided a file `train.ipynb`. This notebook is designed to be compatible *Google Colab* enviroment, thus making training more effective. First sections of this notebook are used to provide setup of environment within *Colab*, if local training is to be used instead it is needed to skip firs sections and set directories for checkpoints and other variable accordingly. Later section provide setup of of training environment such as data modules and callbacks. The notebook can also create a instance of *TensorBoard* in order to monitor training process. The last section is dedicated towards simple inference.

Inference can be done through notebook or with dedicated *Python* script `inference.py`. It can take input from standard input and will print out input plus predicted sequence of set size of next tokens. Tokens are predicted from probability distribution given by output of the network which is later raised to the power of 10. The output is printed in two forms which include split tokens and merged sentence. The script provides simple command line interface and can be run with:

```
./inference.py [-h] -c CHECK -y HYPER -t TOKEN -m MERGE
```

- `-h` Print out a help page
- `-c CHECK` Path to checkpoint of the network saved in the file with extension *.ckpt*
- `-y HYPER` Path to *.yaml* file containing hyper-parameters of the model
- `-t TOKEN` Path to vocabulary of the tokenizer with extension *.json*
- `-m MERGE` Path to merges file used by tokenizer with extension *.txt*

6 Experiments

We have trained these network on machine with *AMD Ryzen 5 2600* CPU, *Nvidia GTX 1060* GPU with 6GB of VRAM and 16GB of RAM. In training we used Adam optimizer, with learning rate 0.0001, batch size 64 and input sequences of length 64. All trained models are available in our public *Google Drive folder*[5].

As the baseline transformer model we trained GPT model with hyper-parameters in Table 1. Our best achieved validation perplexity had value 54.74 after 8 epochs of training, and can be seen on figure 3. Overall real time of training was 26 hours and 15 minutes. This is the largest model we could trained with our hardware resources.

The following two examples show behaviour of this model when predicting next 64 tokens. Tokens used for this experiment are basic *BPE*, meaning that there may be separators in between every token, thus some works are split with spaces.

[5] `https://drive.google.com/drive/folders/1TL5ELIC9gEiN3qaTOFd40FlC3oY3k2qj?usp=sharing`

```
INPUT:  In 1691 Moscow established

OUTPUT: In 16 91 Moscow established the first Polish city of the
        Polish Republic , the first Polish city in the Polish
        city . The Polish city was the first city in the Polish
        city to be built in the Polish city . The city was the
        capital of the Polish city of Warsaw , the capital of
        the Polish city of Warsaw , and the city of Warsaw ,

INPUT:  The Drunk parody

OUTPUT: The Drunk parody The film was released on the episode of
        the United States. The film was written by Tom's first
        film was written by a writer. The film was directed by
        critics and directed by the film, and directed by the
        film directed by John, and directed by her film, and
        directed by David S. James
```

Hyper-parameters	vocab_size	d_model	n_layers	n_heads	d_ff
Values	30 000	768	6	8	1024

Table 1: Architecture parameters for our largest trained GPT model.

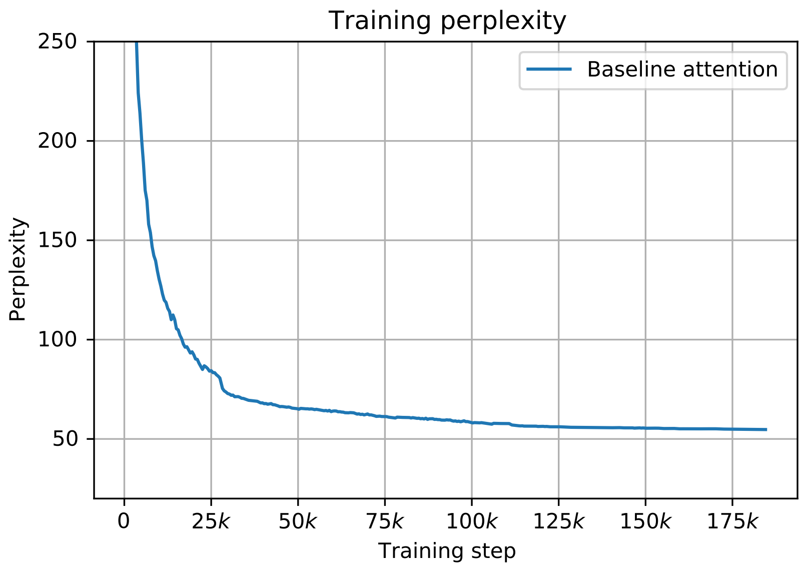

Figure 3: Training perplexity progress of baseline solution

6.1 Performer model experiments

Next we have chosen in order to compare results of classical attention and the FAVOR+ attention to train two net-works with equal hyper-parameters, only difference being in used attention mechanism. The network hyper-parameters are shown in the Table 2. The smaller network size is mainly due to performance requirements of performer network on smaller input sequence sizes. In this smaller setting we have achieved only 129.4 perplexity for GPT variant and 129.2 perplexity for Performer variant, after 5 epochs of training, and can be seen on figure 4. Overall real time of training for GPT variant was 4 hours and 59 minutes, and for Performer it was 21 hours and 11 minutes. Next are examples of input sequences of:

Performer

```
INPUT:  In 1691 Moscow established

OUTPUT: In 1691 Moscow established a in =ed was cont f gener 1950
        gener 1950 a in gener 1950ches gener F a in gener 1950 a
        gener 1950ed was gener 1950 cont gener 1950 f gener 1950
        a in gener 1950chchyl gener F a in gener 1950 F
```

Transformer (Equivalent to performer parameters)

```
INPUT:  In 1691 Moscow established

OUTPUT: In 1691 Moscow established ecn = a =ivingently life =ed
        Cionionionionionionionionion C was contol However
        Howeverionionionionionionion C was contol Howeverol
        Howeverol However distributed contol However f
        Republicecnired return f Republicecn =ch Republic
```

Hyper-parameters	vocab_size	d_model	n_layers	n_heads	d_ff
Values	30 000	128	2	4	512

Table 2: Shared architecture parameters for trained models.

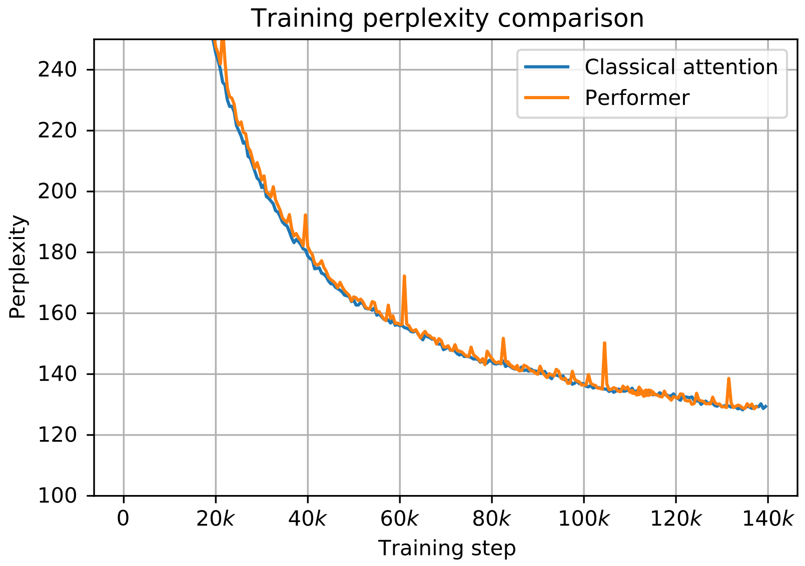

Figure 4: Training perplexity progress with multi-head attention and performer

7 Conclusion

Our work focused on creating language model using two different approaches. We have successfully implemented both attention and performer based language models and showed their train-ability. The attention model has shown upon experimentation promising results and has is able to generate coherent English sentences. Our experimentation using *FAVOR+* algorithm has shown some of its limitations. Approach promises linear time and space complexity of attention approximation however it does not provide these advantages when smaller sequences are used. Thus our experimentation only allowed small models. These results are coherent with theoretical evaluation where it is determined that time and space complexity with relation to sequence length is linear but it is quadratic with relation to embedding size. This is exactly opposite of classical attention. Our model used significantly larger embedding size an thus it could not benefit from advantages provided by *FAVOR+*, which in return worsened its performance.

The experimentation shows that wen equivalent performer and attention model size was used the results were similar. This can be seen in training progress graph but also with the similar output. The output has not been as good as with larger model size but if enough computational resources we would expect similar results. This can be said as training progress of both models is more less equivalent.

Future experimentation may take advantage of larger memory size or by experimenting with different sizes of embedding. The linear time and space complexity can thus provide its advantages and allow training and evaluation on larger samples of data.

References

[1] Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A. et al. Rethinking Attention with Performers. *CoRR*. 2020, abs/2009.14794. Available at: `https://arxiv.org/abs/2009.14794`.

[2] Devlin, J., Chang, M., Lee, K. and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*. 2018, abs/1810.04805. Available at: `http://arxiv.org/abs/1810.04805`.

[3] Goldberg, Y. *Neural Network Methods in Natural Language Processing*. 2017.

[4] Radford, A., Narasimhan, K., Salimans, T. and Sutskever, I. Improving language understanding by generative pre-training. 2018.

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention Is All You Need. *CoRR*. 2017, abs/1706.03762. Available at: `http://arxiv.org/abs/1706.03762`.