Multi-Resolution and Asymmetric Implementation of Attention in Transformers

by

Zaid Hassan Chaudhry

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Transformers are the state-of-the-art for machine translation and grammar error correction. One of the most important components of transformers are the attention layers, but they require significant computational power. We suggest a new way of looking at the "mixing" mechanisms of tokens by doing a multi-resolution implementation of attention, which maintains inference results while also improving training and inference speed, thus getting the best of both worlds. This approximation can be applied in symmtrical and asymmetrical manner within and across attention layers. We also suggest an interesting alternative for the softmax layer in attention. We also analyzed some other hyperparameters in detail. For example, our experiments indicate that we can have asymmetry among the attention layers w.r.t. number of heads, while still achieving similar results. In many cases, reducing the number of heads improves inference results. We also explored the role of weighting matrices for query, key, and value vectors; and show that in case of self-attention, absence of these matrices results in the collapse of the attention layers to an identity matrix.

Acknowledgements

In the completion of this project, I drew on the time and patience of many people. The depths incurred are more extensive and varied than can be detailed here, but certain salient contributions must singled out. Firstly, I would like to thank my thesis advisor Dr. Pascal Poupart for his valuable advice and guidance. It is all due to his unwavering support that I have been able to complete my thesis. There were so many roller coaster rides, but with his unconditional support, I was able to overcome them all.

I would like to thank my father whose support was instrumental in my achievements during every phase of my degree. I would also take the opportunity to thank my dear sisters Zainab and Maryam for the friendship and love. And, how can I forget Abdullah and Qasim, my younger brothers, for the entertainment they always provide. I would also like to thank my grandparents who have always been there for me. My grandfather has always been a source of guidance throughout my life, and my grandmother inculcated me with desire to be the best. Last, and not the least, I thank my dear mom, who will do everything to see me happy.

Furthermore, I would like to thank Asadul-Islam for his insightful comments during various stages of my research work in our frequent meetings with Scribendi, Inc. I would like to thank my dear friend Haris-Bin-Zahid for guiding me through various stages of the degree.

Dedication

To my Mom and Dad.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Grammar error correction and machine translation are two of the most important problems in the domain of natural language processing. Grammar error correction is a task where we take a sentence with incorrect grammar and generate a grammatically correct sentence. Whereas, in machine translation task, we are given a sentence in one language and we are supposed to translate it to another language. We will be applying our proposed techniques to these two tasks to see if our proposed ideas can carry over to different sequence generation problems.

Transformers are the state-of-the-art architectures for both grammar error correction and machine translation. Transformers are neural network architectures that use attention and feed forward layers in addition to some other auxiliary layers like positional encoding. The attention mechanism in transformer architectures is very good at modelling interactions between different words in a sentence. The better modelling of short term as well as long term interactions plays an important role in superior performance to previous techniques.

Training time is one of the bottlenecks in developing neural network models; Inference time is not a bottleneck in many applications. But, in some other applications inference times are also very critical. For example, in self-driving cars all the video data is processed in real-time and inference time is absolutely critical. We worked on grammar error correction applications as proposed by our industrial partner Scribendi, Inc., which is an editing and proof reading service. They have an elite editorial team which provides proof reading services. Scribendi is in the process of using the latest developments in deep learning to improve their services. For this purpose, they proposed different problems, ranging from improved automatic proof reading, to speeding up the inference process, since the clients

want to get the proof reading done in a matter of a few seconds. In this application, training times are much less critical, and inference time needs to be reduced to a minimum. In our work we focus on improving the inference speeds of transformers by focusing on the attention layer.

Scribendi Inc. funded the research work and provided its data for research also. The nature of the data was quite different from publically available datasets. When pre-trained models were fine tuned on this dataset, the results were worse than training the models from scratch on their proprietary data. Therefore, we decided to limit ourselves to developing techniques which do not necessarily need a pre-trained model. In order to show the generality of our techniques, we also added machine translation tasks, and our proposed ideas seems to carry over to machine translation tasks also. The way we formalize the problem, it is expected to carry over to other problems which involve training transformers from scratch.

To optimize transformers, it is possible to play with the number of encoder and decoder layers, optimize the attention layer, etc. For example, Kasai et al. [20] suggests the use of a deeper encoder and a shallower decoder to obtain transformers with reduced complexity. This approach favors inference due to a lighter decoder - which needs to run sequentially during the inference phase. Computational complexity of attention layer depends quadratically on sequence length and linearly on embedding dimension. Many papers make an attempt to reduce complexity of attention in order to improve transformer performance. For example, Katharopoulos et al. [21] swaps the order of softmax and dot product such that computational complexity depends linearly on sequence length and quadratically on embedding dimension. This reduces complexity for sentences which are longer than the embedded dimension.

We also optimize computational complexity of transformers by reducing complexity of attention layers. Our approach is orthogonal to previous approaches in improving attention and can be applied in conjunction with other techniques which optimize attention layers. We introduce new hyperparameters which can be used to reduce the complexity of attention. Two approaches to improving the complexity of attention layers are proposed. The first approach is the use of multi-resolution attention, where reduced dimensional implementation improves the speed while maintaining similar accuracy. Another approach is to improve the complexity of attention layers by replacing the expensive operation of softmax with other options. The multi-resolution idea introduced in this thesis, when used in a "symmetric manner", linearly reduces the number of computations of attention layer.

Results for the proposed hyperparameters are explored in this thesis. For a wide range of configurations, the results show that the new hyperparameters lead to interesting choices.

Different nonlinearities for attention layer as an alternate to softmax, are also proposed. RELU turns out to be a possible choice to replace softmax for certain applications. Finally, a broad exploration of architectures is done using random search and the top architectures proposed by the neural architecture search are reported. There are also other explorations and suggestions made in this thesis. Namely, the asymmetric use of heads is explored. The importance of linear layers in attention is also explored. Experimental results confirm the importance of linear layers, especially in self-attention.

To give very good results for the above tasks one needs to train the transformers for millions of sentences, which required days of training time. Any significant improvement in performance can save hours and even days in training time. Our work improves the speed of transformers for both inference and training by a factor of upto 17%.

Chapter 2

Background information

2.1 Recurrent Neural Networks

Recurrent neural networks (also known as RNNs) revolutionized neural architectures. This architecture is designed to process sequential data. This architecture is able to create an embedding of a sequence of tokens that has no restriction on length. In Figure 2.1, we can see that a sentence "This is a car" is being input into an RNN. Each word is being fed at different time steps. Once a word is input into the RNN block, the RNN block takes the current input and the state from the previous time step to compute the current state. The initial state of the RNN can be initialized to a zero vector.

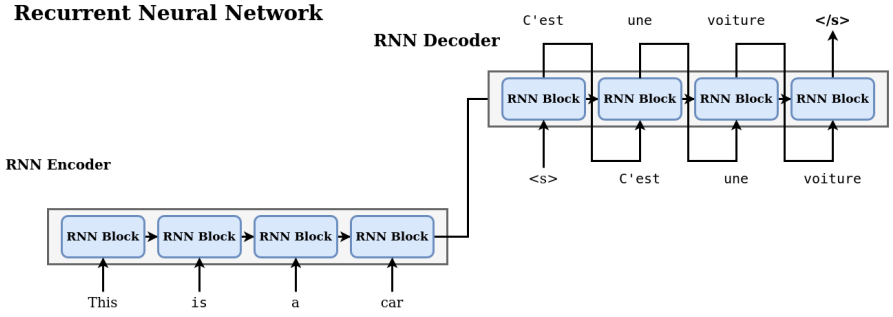


Figure 2.1: Recurrent Neural Network

At each time step, the previous state of the RNN and the input for the time step are transformed. The transformed input and transformed previous state are added and the

result is transformed to give the current state of the RNN. Transformations used here can be linear or non-linear.

Once the sequence is passed through the RNN, we can add more RNN layers on top of the first RNN layer. These layers will take as input the states from the previous layer. Then they will process the input as described in the previous paragraph. An RNN component processing a sentence as described up to this point is called an "RNN encoder" as labelled in Figure 2.1.

In order to solve a sequence classification task (such as emotion classification), we can apply a classifier to the current state from the last time step. To solve a sequence tagging task (such as part-of-speech tagging), we can have a classifier repeated at each time step. However, if we wish to solve a conditional sequence generation task (such as machine translation), we need a decoder.

For a decoder, the method of operating is the same as the encoder. The difference is that in this case the input for each time step is the output token from the previous time step. Also the hidden state of the decoder is not initialized by a zero vector but rather by the hidden state from the last time step of the encoder. During training, we can either take the output from the previous time step generated by the model or use the token from the ground truth output sequence. Using the previous token from the ground truth is called *teacher forcing*.

One addition which can be made to the decoder is to include a third input into the RNN block. This input is a weighted combination of the hidden states from each of the encoder time steps. The weights for the weighted combination can be found by computing the similarity of the decoder hidden state with the encoder time steps. Once the similarities have been computed, they are used to weigh the encoder hidden states. In the case of English-French translation this process allows the encoder to focus on the right input words to produce the next output word. For example, when translating the sentence "This is a car" to "C'est une voiture", the RNN is able to focus on the word 'car' to produce the translation 'voiture' instead of having to rely only on the final sentence representation produced by the encoder.

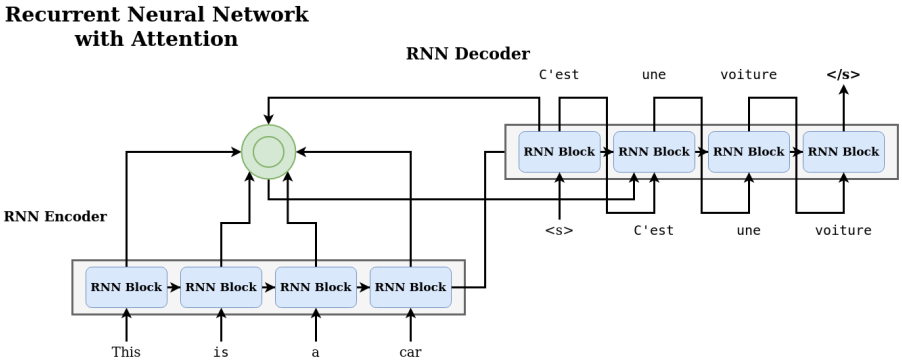


Figure 2.2: Recurrent Neural Network with Attention

This process known as attention is represented by a green circle in Figure 2.2. The green circle has arrows coming in from each of the encoder time steps and the first decoder time step. These represent the flow of states from these blocks to the attention module. The operation of attention is computed using these states and the output is fed into the next decoder RNN block as the third input. This input is transformed by a linear layer and added to the transformed input and transformed previous hidden state.

The attention mechanism eases the problems caused by sequential processing of data. In sequential processing, information from tokens is transformed multiple times across different time steps. By the time, information from initial words of the sentence reaches the future time steps, the information has been mixed with multiple intermediary words. As we will see in the next section, this problem can be completely alleviated by using an attention mechanism in place of the RNN layer.

2.2 Transformers

In Figure 2.3, we can see the transformer encoder inside a gray box on the left. The transformer decoder is shown in a gray box on the right. The transformer decoder and transformer encoder are just like the encoder and decoder in RNNs in terms of their functionality. Inside the transformer encoder, we can see self-attention in a blue box as the first computational block. Self-attention mixes the embeddings along the sequence length dimension. We will discuss self-attention in the next section. There is a skip connection going besides the self-attention. The output from the self-attention, and the skip connection over the self-attention, are combined using an addition represented by a green circle. This

is followed by a normalization layer as described in Ba et al. [3]. Layer normalization is followed by a linear layer, which in turn is followed by a non-linear activation and finally by yet another linear layer. The linear layers and activation are shown in red. The linear layers mix the embedding along the dimension of the embedding.

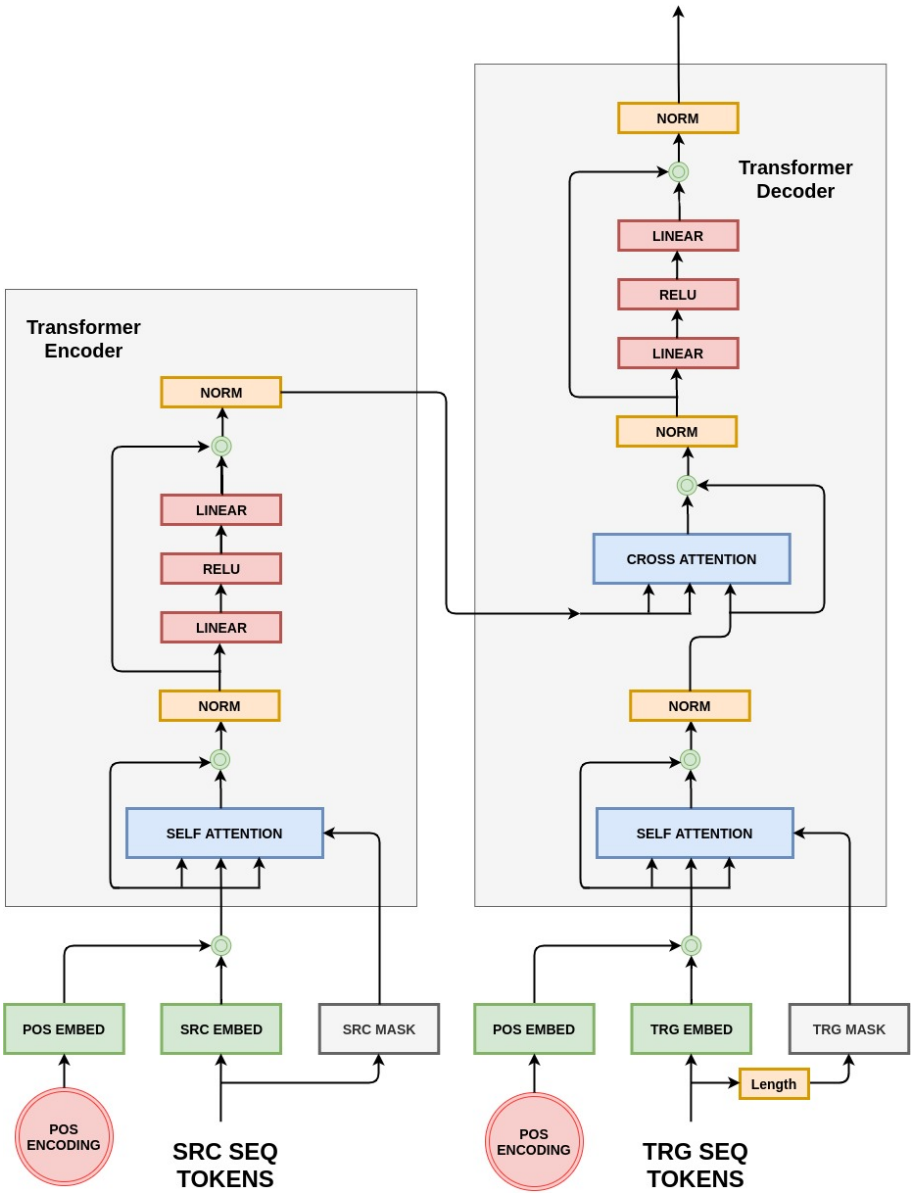


Figure 2.3: Transformer diagram

In the transformer decoder, the first block applied to the input is again the self-attention block. This block is however slightly different compared to the self-attention block in the encoder because it is a causal self-attention block. In causal self-attention block each hidden state is computed from a weighted combination of hidden states of the decoder in the previous layer. However, there is one constraint. The weighted combination is taken over only the positions from the past up to the current position. That is, values from the future are not used to compute the hidden state for the current state.

Causal self-attention allows training to work just like inference. At inference time, future tokens are not available, which is why the network should be trained using causal self-attention. To train causal self-attention in a parallel fashion, a mask is generated. This mask zeros the weights corresponding to the future hidden states computed by a dot product between all embeddings. The weighted combination is then computed as usual.

Another technique for running the transformer in a parallel manner is to always do teacher forcing. This means that we do not have to wait for the entire stack of layers to run and produce an output token before running the entire stack of layers all over again for the next time step. Instead with the masking combined with teacher forcing, we can run the training for all the time steps simultaneously.

In the transformer decoder, there is an additional block of cross-attention. This is shown (in Figure 2.3) as the blue block in the center of the decoder with two arrows coming from the encoder. In this block, similarity between embeddings from the encoder and the embeddings from the decoder are computed. The matrix of similarities is used to compute a weighted sum of the encoder hidden states for each of the embedding position in the decoder. This layer is followed by linear layers sandwiching an activation. These linear layers transform the weighted combination of the encoder hidden states to a new domain (more appropriate for the decoder).

Transformers have attention as the mixing mechanism along the sequence length dimension. This is different from RNNs [18] which use a recurrent connection to process words along the sequence length dimension. Transformers not only have attention for paying attention from the decoder to the encoder hidden states but also self-attention within the encoder and the decoder. This is why transformers can be called self-attention networks.

Traditional Transformers use the dot product attention. The dot product attention makes the mixing of the encoder states dependent on the data. This makes it more expressive compared to methods which mix the tokens based on constant matrices. There has been work on mixing tokens using constant matrices [24]. It essentially adds the embeddings based on preset weights. Meanwhile dot product attention takes a weighted combination of the embeddings based on the similarity of the embeddings to one another.

Transformers have two major components: the encoder and the decoder (see Figure 2.3). Between these components there are three attention layers. The first attention layer is the self-attention in the encoder, the second attention layer is the causal self-attention in the decoder and the last attention layer is the cross-attention from the decoder to the encoder hidden states. The decoder contains two attention layers and runs autoregressively and hence is the main bottleneck in the speed of transformers. Therefore, our work (where we tackle attention) impacts the performance of the decoder more than it impacts that of the encoder.

Attention is the central component of transformers and is the focus of our work. As already described, transformers rely on attention to have tokens impact one another. One thing which sets apart transformers from RNNs is that all the tokens are neighbours in transformers while they are separated by the intervening words in RNNs. This makes understanding the meaning of the sentence easier for transformer since the impact of words is not diminished due to passing through multiple time steps. For transformers, therefore, it is easier to understand context than RNNs. For example, in the sentence, "He went to the bank to pay his bills", the affect of "pay" and "bills" will allow transformer to understand that bank means a "money bank" as opposed to a "river bank" or "relying upon someone". Meanwhile for RNNs, understanding the context will be harder given how words "decay" across time steps. In other words, transformers create contextualized word embeddings that can be used to predict the output. Each token depends on all other tokens as we go deeper through a transformer. This allows for better understanding of each token in the context of the sentence.

Besides attention layers transformers use fully connected layers, positional embeddings and word embeddings. A fully connected layer is a standard neural network layer consisting of a weight matrix and an optional bias. Positional embedding is added to the word embedding to enable sensitivity to position since there is no inherent mechanism in the transformer which accounts for position.

The transformer has been designed to be relatively (compared to RNNs) more efficient for training. It does not have to train autoregressively (even though for inference it runs autoregressively). This is achieved by introducing a masking method in the decoder self-attention. This masking method is an interesting innovation of the transformer. Words in the future are only allowed to attend to past words in the decoder self-attention by using this masking method. Furthermore teacher forcing is always applied to teach the transformer to predict the next word. Once the transformer is trained it runs autoregressively and is able to naturally see only past words. Since the transformer has been trained appropriately this is not a problem.

2.3 Attention

This section details self-attention a key contribution of the transformer architecture. Attention is a mechanism whereby contextualized embeddings are created in transformers. Initially, embedding vectors are simply a representation of each token independently. However, as we move through attention we create a weighted combination of all embedding vectors in the sentence. This incorporates information from all the other tokens in the sentence into the current token. To understand why this is important, take the sentences "I have read the paper" and "I will read the paper". In the first sentence and the second sentence there are two different phrases (spelled the same coincidentally) "have read" and "will read". The difference between the two phrases can only be understood if information from "will" and "have" tokens is incorporated into the token "read". By incorporating this information, a transformer trained for text-to-speech synthesis will pronounce the word "read" differently in the two phrases.

Attention is a key component of Transformers. To understand how it works we take the self-attention layer in the transformer as an example. It takes the embeddings for the tokens as input. It then transforms the tokens into three different domains. The first two domains are used to compute similarity between the embeddings. This is done by taking a matrix product between embeddings from the first domain with embeddings from the second domain. Once the similarity is computed, a weighted combination over the embeddings in the third domain are taken using the weights previously computed. The equation for attention is given below:

$$V' = W_P \left(softmax \left(\frac{QW_q(KW_k)^T}{\sqrt{d_k}} \right) (VW_v) \right) = W_P A_w (VW_v) \tag{2.1}$$

where V' is the output of the attention layer. W_q, W_k, W_v and W_P are the weights of linear layers applied to the queries matrix Q, keys matrix K, values matrix V and attention output P. The choice of the query, key and value matrices depends on which of the three attention layers is under question. Specifically for self-attention in encoder, all three matrices contain the same embeddings that correspond to the source tokens or the output from the previous encoder layer. For causal self-attention in the decoder, all three matrices contain the same embeddings that correspond to the target tokens or the output from the previous decoder layer. For cross-attention, the query matrix corresponds to embeddings from previous decoder layer, while the key and value matrices both contain the outputs from the encoder. d_k is the embedding dimension for the query, key and value matrices.

Moreover,

$$A_w = softmax\left(\frac{(QW_q)(KW_k)^T}{\sqrt{d_k}}\right) \tag{2.2}$$

Note that in this equation, the embeddings in V, K and Q are all arranged in rows. This means that the application of W_v is on each of the embeddings independently. Meanwhile, application of W_a takes a weighted combination of the transformed embeddings in V.

2.3.1 Components of Attention Layers

The complete block diagram of the transformer is shown in Figure 2.3 with the three attention layers shown in blue color. In Figure 2.4, we zoom into the attention block to show the internal components of the attention layer. The attention layer can be broadly divided into five stages: The pre-attention linear layers, the dot product, the softmax layer, then a weighted sum, and finally a projection matrix.

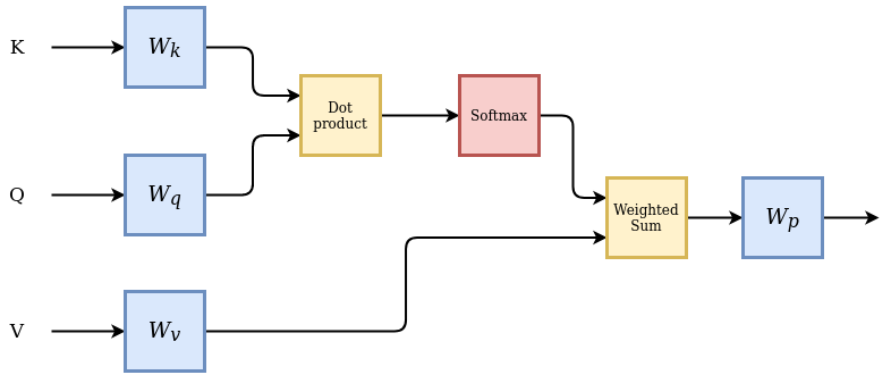


Figure 2.4: Attention diagram

The pre-attention linear layers transform the inputs Q, K and V to another same dimensional space. In the second stage, dot product between transformed Q and K is computed. This dot product computes the correlations between the embeddings of the various tokens. In the third stage we compute softmax of the dot product. In the fourth stage, we compute the weighted combination of the value embeddings based on the correlations found in the first dot product; this has been represented by the yellow box labelled

"Weighted Sum". Finally, the the post-attention linear layer (shown in blue on the right) is applied to the result of the weighted sum of values.

2.3.2 Computational Complexity of Attention

The computational complexity of attention layer is determined in this section. In order to define the computational complexity, the following variables have been defined.

$$\text{Source Length} = n_s$$
$$\text{Target Length} = n_t$$
$$\text{Embedding dimension} = d$$

Computational complexity of a matrix multiplication operation is known to be $O(mnp)$, where the two matrices are of the order $m \times n$ and $n \times p$, respectively. Precisely speaking, there are $m \times n \times p$ multiplication operations and $m \times (n-1) \times p$ addition operations, for n large enough we can say there are roughly $2 \times m \times n \times p$ operations.

Computational complexity of attention is dominated by the linear layers in attention. Each linear layer is equivalent to a matrix multiplication and a bias addition. All of these matrices are $d \times d$, and inputs are either $n_t \times d$ (in case of Q and P) or $n_s \times d$ (in case of K and V). Hence, we get the following computational complexities:

$$O(QW_q) = n_t \times d \times d \tag{2.3}$$
$$O(KW_k) = n_s \times d \times d \tag{2.4}$$
$$O(VW_v) = n_s \times d \times d \tag{2.5}$$
$$O(PW_p) = n_t \times d \times d \tag{2.6}$$

where the variables W_q, W_k, W_v and W_P are the weights of the linear layers applied to Q, K , V and P. Q, K and V are the query, key and value matrices respectively. P is the attention output before application of W_P.

Similarly, for dot product in second stage of attention, the complexity is:

$$O(QK^T) = n_t \times d \times n_s \tag{2.7}$$

where, Q is equivalent to QW_q and is simply the transformed query matrix. Similarly, K is equivalent to KW_k. When computing correlations, we will use K and Q to represent the transformed matrices instead of the original matrices throughout the thesis.

The softmax operation is applied to QK^T and for e digit accuracy the computational complexity for this operation is [1]:

$$O(softmax(QK^T)) = n_t \times n_s \times \log^2 e \tag{2.8}$$

The order of the softmax matrix is $n_t \times n_s$, and we multiply it with the projected V matrix which has the order $n_s \times d$, so for the weighted sum operation, the complexity is:

$$O(A_w V) = n_t \times n_s \times d \tag{2.9}$$

where A_w is as defined in Equation 2.2.

The FLOPs for the baseline implementation of attention for typical values is given below. Here we have assumed source and target lengths of 33 and 30, respectively. An embedded dimension of 512 is assumed.

1. $n_t = 30$, $n_s = 33$, $d = 512$
2. $FLOPS(W_q Q) = FLOPS(W_P P) = 30 \times 2 \times 512 \times 512 = 15,728,640 = A$
3. $FLOPS(W_k K) = FLOPS(W_v V) = 33 \times 2 \times 512 \times 512 = 17,301,504 = B$
4. $FLOPS(QK^T) = 30 \times (2 \times 512) \times 33 = 1,013,760 = C$
5. $FLOPS(A_w V) = 30 \times (33 \times 2) \times 512 = 1,013,760 = D$
6. Total $= 2A + 2B + C + D = 68,087,808$

2.4 Computational Efficiency for Transformers - A Literature Survey

For complex sequence generation tasks, the training data is usually enormous, say millions of sentences. To train transformers for such large amounts of data, we need several layers of encoders and decoders. The training times are very significant, of the order of days on very powerful GPUs. Hence, there has been an interest in making them faster and therefore easier to train and deploy. Attention uses a large number of operations in sequences with longer length. Specifically, the number of operations scale quadratically with sequence length.

2.4.1 Generating Long Sequences with Sparse Transformers

Child et al. [8] improves the efficiency of transformers for problems where sequence length n is large. It does so by making attention sparse. Instead of computing similarity between all pairs of embeddings, it computes similarity over a given subset of embeddings for each position. The restriction is that there should be connectivity between all positions over several steps of attention. Another restriction is to have a maximum length for the shortest path connecting any pair of position. The subsets of embeddings are chosen such that this paper improves complexity from $O(n^2)$ to $O(n\sqrt{n})$.

2.4.2 Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context

Recurrence is used in Dai et al. [12] to handle long sequences. Instead of applying attention over all neighbouring segments simultaneously, this paper establishes a recurrence relationship. It uses the hidden states from the previous segment and the hidden states from the current segments to compute the hidden states for the next layer. This approach can be extended to using hidden states from the previous T segments.

By applying the mechanism above, this paper increases long-range dependencies while also solving the context fragmentation problem found in previous approaches where text is split without regards to any sentence or semantic boundary. By establishing a recurrence relationship, this paper allows information from the first part of a sentence to flow to the next part of the sentence, hence solving the context fragmentation problem.

2.4.3 Reformer: The Efficient Transformer

Kitaev et al. [22] reduces the complexity of attention by computing a sparser version of attention. Here, this sparsity is achieved by computing the value of QK^T by using only the closest 32 or 64 keys for each query. In other words, only a subset of the attention weights with the largest values are calculated, since they contribute most to the final values.

The closest vectors are found using a locality-sensitive hashing scheme. This is a hashing scheme where nearby vectors get the same hash with high probability and distant ones do not. A random matrix is multiplied with each embedding and the position of the maximum value inside the vector is taken as the hash. Once hash buckets are computed, attention is computed within the buckets and not across them. By doing this, the paper reduces the complexity from $O(n^2)$ to $O(n\log(n))$.

2.4.4 Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention

Katharopoulos et al. [21] changes the complexity of attention from $O(n^2d)$ to $O(nd^2)$ where n is sequence length and d is embedding dimension. This is achieved by changing the order of the two dot products in attention. They also speed up inference by showing that doing so leads to transformers being equivalent to RNNs.

2.4.5 Big Bird: Transformers for Longer Sequences

Zaheer et al. [36] uses "memory" and sparse attention to reduce complexity. For sparse attention it has two ways of choosing the keys with whom similarity is computed for a given query. The first is to take the neighbours of the current token from a window extending in both directions. The second is to take r random keys from the input sequence.

For "memory", additional tokens are added to the sequence. These tokens are able to access the entire sequence. These "global tokens" can be thought of as the "memory" which holds a compressed representation of the entire sequence by aggregating the input embedding g number of times. This paper reduces the complexity from $O(n^2)$ to $O(n)$.

2.5 State of the Art for Grammar Error Correction

2.5.1 Better Evaluation for Grammatical Error Correction

MaxMatch is an evaluation procedure which measures how well an algorithm did in producing a grammatically correct sentence from a grammatically incorrect sentence. It focuses on finding edits in the sentence. Edits are the differences between the original sentence and the correction. If "Our transformer *process the* tokens efficiently" is the original incorrect sentence and "Our transformer *processes* tokens efficiently" is the reference correction, then the edit will be ((), *process the* $\rightarrow$ *processes*). Thus, an edit can be a replacement of a word. It can also involve adding a new word in the sentence or removing a word from the sentence. That is, a series of edits are transformations which result in the corrected sentence.

The reference corrected sentence can be compared with the original sentence to identify the ideal model edits. Then the system generated sentence can be compared with the

original sentence to identify the model edits. These are then compared with each other as described in the next paragraph.

Precision, Recall and F1 score are then computed for how many edits are identified correctly. Precision tells us how many of the model edits were present in the perfect model edits and Recall tells us how many of the perfect model edits were discovered by the model.

2.5.2 Neural Grammatical Error Correction Systems with Unsupervised Pre-training on Synthetic Data

Roman et al. [17] uses the transformer architecture described in Section 2.2. This paper improves results by creating a large synthetic data set. Grammatically correct sentences are taken and errors are added to them. These sentences containing errors are then used as input, and the original grammatically correct sentences are used as the ground truth during training. To train the system, 200 million pairs are generated.

We now describe the steps taken to generate the error-full sentences. The number of errors in a sentence are sampled from a Gaussian distribution. Then for each error to be introduced a word is chosen and an error is introduced in it. On the selected words, one of three operations may be performed: replacement, insertion or deletion. The first operation is replacing a word by another word. Replacements could have been chosen randomly from a dictionary. However, this paper does a good job of replacing the current word with a word which is commonly confused with the current selected word. Commonly confused words are available in the confusion lists available inside spell checkers. This paper uses these confusion lists to choose replacements. The other two operations involve adding a random word after the currently selected word or deleting the current word.

2.5.3 An Empirical Study of Incorporating Pseudo Data into Grammatical Error Correction

Kiyono et al. [23] is similar to Roman et al. [17]. It uses a full transformer just like Roman et al. [17]. This paper also augments data by introducing errors in grammatically correct sentences. They produce synthetic data using two methods. The first method puts noise directly into the sentences and the second method for generating sentences involves training a system to produce sentences with errors in them.

The first approach of directly putting noise into sentences involves choosing one of four options for each word in the sentence: masking, deletion, insertion or keeping the original.

The mask operation is novel to this paper and is not seen in Roman et al. [17]. The second approach is called back translation. In this approach a system is trained in the reverse direction. That is, grammatically correct sentences are used as input and grammatically incorrect sentences are used as output. Once the system is trained, any corpus with good grammar can be input into the trained model and corresponding incorrect sentences can be generated. According to the experiments conducted in the paper the first approach of directly injecting noise into the correct sentences works better.

This paper also explores the choice of the corpus from which incorrect sentences are generated and finds that a corpus with good grammar is a better choice than a corpus with weaker grammar. It also explores the choices of either doing training on the synthetic and human annotated data in the same stage, or, training on synthetic data in the first stage and then fine-tuning on the human annotated data in the second stage. It finds that the latter choice is better, probably since the teaching signal from the synthetic data becomes dominant when the two corpora are used in the same stage. Moreover, the paper shows that increasing the number of synthetic sentence pairs improves the results.

Compared to Roman et al., Kiyono et al. [23] used the algorithm described above which is different from the one used to generate synthetic data in Roman. Moreover, Kiyono et al. made a more thorough evaluation of training and dataset choices. His paper also makes corrections in sentences only if they are classified by a specially trained classifier as having a grammatical error which is not the case for Roman. Kiyono et al. also beats the state-of-the-art result which had been established by Roman et al.

2.5.4 Parallel Iterative Edit Models for Local Sequence Transduction

Awasthi et al. [2] predicts edits instead of tokens. Edits can include replace, append, copy and custom transformations. Examples of custom transformations include changing words from “-s” to “-ing” and from “-ed” to “-s”. Predicting edits makes the output vocabulary smaller. This means that we do not need a large amount of data to train the system.

Edits are computed using a transformer encoder followed by a classifier. This makes this model a sequence labelling model unlike the two models above which are sequence generation models. By doing sequence labelling using a transformer encoder followed by classifier, the system is allowed to run non-autoregressively. This makes it 5 to 15 times faster compared to models based on the full transformer.

Running non-autoregressively does not allow dependency between output tokens to be modelled. To improve the output of this model, the sentence is passed through the model

more than once. This allows the model to do things like making compound changes such as inserting two words after a given word (one edit can insert only one word at a time).

To reduce the amount of specialized data for training a good system, a pre-trained language model is used in place of transformer encoder. For this paper, the pre-trained language model described in Devlin et al. [14] is used.

2.5.5 GECToR – Grammatical Error Correction: Tag, Not Rewrite

GECToR [28] is closely related to Awasthi et al. [2]. Just like Awasthi et al., GECToR does not attempt to rewrite the correct answer, instead it does edits into the sentence iteratively to generate a correct sentence.

The differences between GECToR and Awasthi et al. [2] are in the classifier and the vocabulary of edits. The classifier for GECToR is a simple linear layer. Meanwhile for Awasthi et al., the classifier has different equations for computing the probability for each of the edits. Unlike Awasthi et al., the custom transformations do not attempt to do things like changing "-ed" to "-s". Instead, they change verb form. For example, "drinking" can be converted to "drank". These differences in vocabulary and classifier allowed GECToR to beat the state-of-the-art models to become the leading model.

Recently, Roche et al. [32] beat the state-of-the-art set by GECToR. It used a large synthetic data and a very large pretrained language model to achieve state-of-the-art performance.

2.6 State of the Art for Machine Translation

2.6.1 BLEU: A Method for Automatic Evaluation of Machine Translation

Papineni et al. [30] introduces the BLEU score. This score is used to evaluate machine translation systems. For finding the value of the measure, the number of "sets of n consecutive words" (n-grams) found in both reference and system translations are divided by the total number of n-grams in the candidate translations. This corresponds to precision for a given value of n and is denoted by p_n. This process is repeated for different values of n (up to 4). A logarithm function is applied to the precision obtained for each value of n.

A weighted average of the p_n's is taken using the weights w_n. The result is then exponentiated and multiplied by a brevity penalty. BP (brevity penalty) is a term which increases exponentially as the length of the sentence becomes smaller than the reference translation, while it is clipped at 1 if it is longer than the reference translation. The equation is given below:

$$\text{BLEU} = BP. \exp \sum_{n=1}^{N} (w_n \log p_n)$$

where, p_n = Precision for n-grams of size n, w_n = Weight corresponding to the value of n and BP = Brevity penalty

The brevity penalty discourages sentences shorter than the candidate sentence by penalizing shorter sentences using an exponential relationship. BLEU has been analyzed to show a high correlation with human judgement. Being an automated measure, it saves a lot of time compared to human judgement which can take days and weeks.

Evaluation of Grammar Error Correction cannot be done using the measures commonly using BLEU. We found that measures for machine translation like BLEU give very good results on grammar error correction if the source sentence is used as the system output and then compared with the reference output. However, MaxMatch introduced in Dahlmeier et al. [11] does not face this problem.

2.6.2 Massive Exploration of Neural Machine Translation Architectures

Le at al. [6] made an exploration of RNN architectures for neural machine translation. In doing so, the model achieved state-of-the-art performance. It uses GRUs [9] and LSTMs [18] which are both variations of RNNs. This paper acknowledges the amount of time it takes to train RNN variants. It attempts to reduce the amount of time spent by other researchers on training these models by identifying intuitions for choosing hyperparameters.

This paper gives a series of findings for training RNN variants. It finds that LSTMs give better results compared to GRUs. It also finds that using smaller embeddings also performs surprisingly well. They also demonstrated that deeper models need residual connections to train. Moreover, the results in the paper show that bidirectional encoders (which process the sentences in both directions) perform better than unidirectional encoders.

2.6.3 Scaling Neural Machine Translation

Even with transformer architecture introduced in Vaswani et al. [34], neural machine translation models can take days to train. The problem can be resolved by applying more resources to the task. That is, by parallelizing the code across multiple GPUs and machines. Parallel training can speed up training but it introduces a number of challenges. These challenges include having to deal with different mini-batches taking different amounts of time when run in parallel on different GPUs, having to use larger batch sizes which can affect generalization performance and so on.

This paper [29] overcomes these challenges and succeeds in parallelizing the training across 128 GPUs while achieving state of the art performance. This paper allows for faster training which allows us to train for more epochs in lesser time. Training for more epochs gives better results than [34]. Not only does this paper improve speed and hence performance by training across multiple machines, it also speeds up the training on a single GPU. It does so by implementing reduced floating point precision and increasing the batch size on a single GPU.

2.6.4 Lessons on Parameter Sharing Across Layers in Transformers

This paper [33] explores how efficient use of parameters can improve performance. Generally, models with larger number of parameters lead to better results. However, there is a restriction on the number of parameters that can be used due to the size of the GPU memory. Since there is often a limit on the GPU memory and hence the size of the model, the question of how to use a given number of parameters efficiently arises. A vanilla model can be outdone by models such as Universal Transformers (Dehghani et al. [13]) which use all the parameters in a single layer which is then repeated for a given number of times.

This paper does not use all the parameters in the same layer since using all the parameters in the same layer increases the size of individual layers and hence the time taken to train. Instead, it places unique parameters in M layers. These M layers are then repeated in L times to give a total of M x L = N layers. This paper tries different ways of repeating the M layers. In one method (CYCLE), the layers are repeated in the original order L number of times. In another method (SEQUENTIAL), L consecutive layers share the same parameters and the M sets of these L consecutive layers are repeated. The paper finds that CYCLE and similar methods work better than SEQUENTIAL.

Using the techniques suggested in the paper [33], Kiyono et al. succeeded in achieving state-of-the-art results. This paper currently has the best performance across all papers on the WMT2014 English-German dataset.

Chapter 3

Contributions

3.1 Main Ideas

Attention is the key component of transformers. In this capacity, it warrants a special exploration into its various facets. We have identified the main features, strengths and function of the various components of the transformer. In this work, we report a detailed discussion into a neural architecture search, which is the highlight of this thesis. This thesis is interspersed with the introduction of a new architectural component of the attention layer called reduced order dimension, a study into the need for linear layers within the attention layer and various new choices in hyperparameters. The thesis finishes with the use of neural architecture search to identify architectures with the best results.

One architectural choice is the use of different dimensions across different attention layers, or within an attention layer. When similar configurations are not used in the attention layers, we call such implementations "asymmetric". The word asymmetry will not only be used w.r.t. dimensions, but also will be used w.r.t different configurations, e.g. difference in number of heads, etc. For asymmetric implementations we study asymmetry among the different attention layers as well as within an attention layer.

Another interesting idea is the exploration of a different non-linearity instead of softmax. We have also explored the necessity of linear layers in the attention mechanism for different attention layers. Most of the study has been done to reduce the computational complexity of the attention layers.

Having introduced all the terminology, we can now discuss examples of asymmetry. For example, we can run cross attention without linear layers and self attention with linear

layers, this is asymmetry with respect to the three attention layers. Another example of asymmetry is running different attention layers with different number of heads, this is again asymmetry with respect to the three attention layers). In asymmetry within an attention layer, we run part of the linear layers at different resolutions.

To summarize, the three primary ideas/results which we will discuss in this chapter are:

- Reduced Order Attention:
  - The idea is to use existing linear layers processing Q, K, V to project them into a lower dimensional space and another projection, finally projects them back in the original order. Note that the layers are already there running at higher dimensions (embedded dim) and project Q, K, and V to a space suitable for doing the attention. We use them to achieve both functions simultaneously.
- A Different Choice of Non-linearity
  - We explore simpler functions to replace the softmax function. These functions will have lower complexity than softmax.
- Removal of Linear Layers
  - For cross attention we can remove the linear layers which project Q, K, and V to a different space.
  - We will show that we cannot remove these linear layers in self-attention, otherwise, a self-attention layer reduces to an identity transformation.

3.2 Reduced Order Attention

In the previous section, we alluded to a possibility of running attention at a different "resolution" or dimension other than the embedded dimension. In this section we will elaborate on this work. Before introducing the idea, we take a quick look at the basic structure of attention layer. Also, we establish a baseline complexity for attention layer. This baseline will then be compared with complexity of the corresponding layer running at a reduced dimension.

3.2.1 Basic Structure of an Attention Layer

In Fig. 3.1, the diagram of attention is shown. The dot product of K and Q is taken after projecting them to a different space via W_k and W_q respectively. Then, softmax is taken over the result of the dot product. The output of softmax acts as a weight matrix which acts on V and produces V' which is a linear combination of vectors of V. Finally, a projection W_p is applied to the weighted sum found in the previous step.

Having explained the basic structure of attention layers, we will explain the main contribution of our work and the trick we use to reduce the computational complexity.

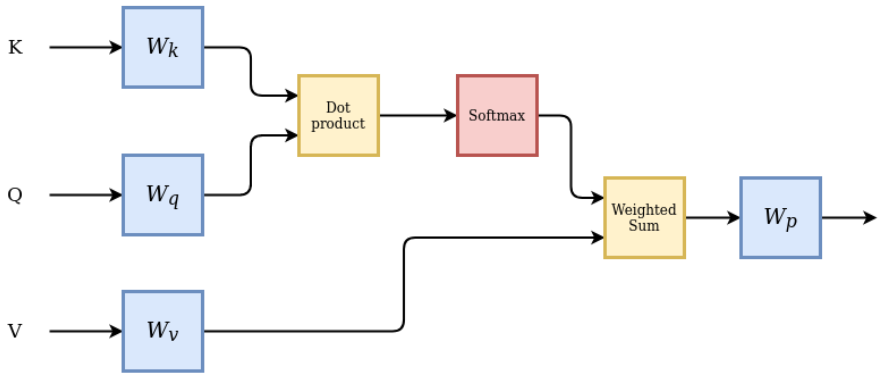


Figure 3.1: Attention diagram

3.2.2 The Trick

In this section, we introduce additional linear layers in series with the projection matrices W_q, W_k, W_v, and W_p (see Fig. 3.2). Since, our focus is reduction in complexity, this step sounds quite counter intuitive. Later we will see, how we can get rid of these matrices.

The matrix which comes immediately after W_q is $\overline{W_q}$ different input and output dimension. Similarly, there are matrices right after W_k and W_v. These matrices are used to change the dimension of K and V respectively. Nothing prevents us now from choosing a lower dimension for the output of these matrices as compared to the input. However, the reduction factor for $\overline{W_q}$ and $\overline{W_k}$ needs to be the same.

The reduced dimension of the above two matrices lead to a reduced dimension for dot-product between projections of K and Q, and low dimensional weighted summation of

projections of V. The result of the weighted summation is then scaled back up via another scaling matrix $\overline{W_p}$. This matrix acts as a bridge to come back to the original dimension of the transformer.

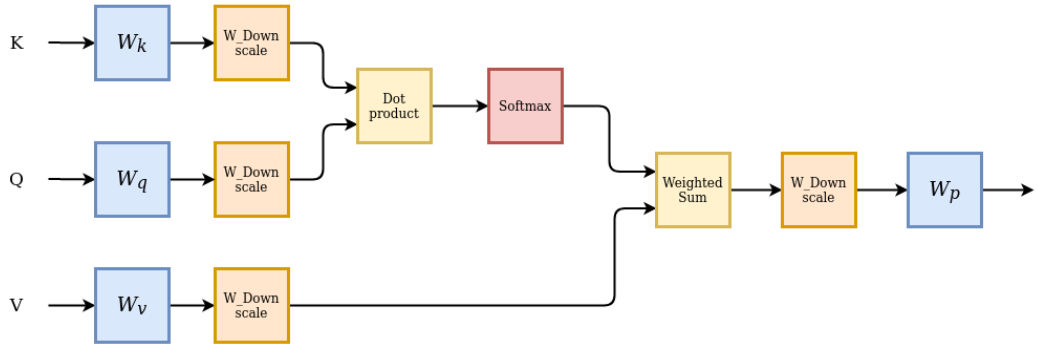


Figure 3.2: Introduction of additional scaling matrices

In Fig. 3.3, we have boxed together the counter part linear layers of W_q, W_k, W_v, and W_p. It is quite straightforward to see that the matrices within a box can be combined together into a single matrix. Obviously, the computational complexity will be reduced if we use this equivalent matrix directly.

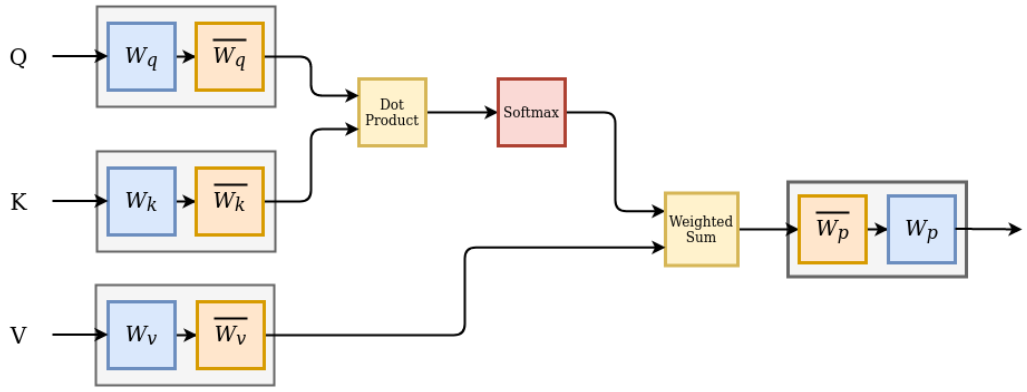


Figure 3.3: Combining the scaling matrices with their counterparts

In Figure 3.4, we can see that once we combine the linear layers for downscaling with the linear layers W_K, W_Q and W_V, we get the same diagram as the original. This gives lower

computational complexity than before. In Section 3.10, we will compare the computational complexities of traditional attention layer and the one with proposed modifications.

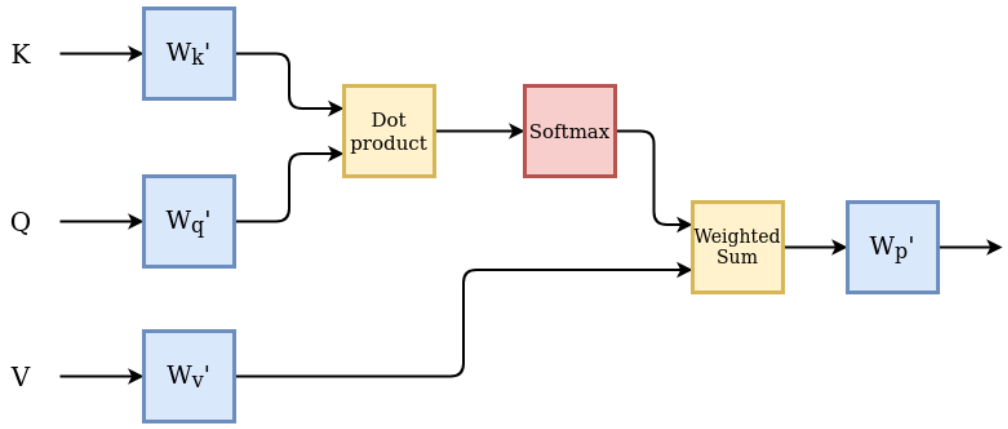


Figure 3.4: Combining matrices - Final configuration is similar to base structure

3.2.3 Asymmetric Compression within an Attention Layer

The two operations in the attention layers which place a constraint on the reduction in dimensions are dot product and weighted sum. The dot product operation takes the output of W_k' and W_q', thus they need to have the same output dimension.

In Figure 3.5, what I call the upper leg is shown in green and what I call the lower leg is shown in blue. This terminology will be used throughout the rest of this work. Since the dot product requires Q and K to have the same dimension, the upper leg has to operate on the same dimension. It is, however, independent of the lower leg.

Since W_P operates on the result of the weighted sum of values, the input dimension of W_P is depends on the output dimension of W_V. There are, therefore, two separate legs in attention. These two legs can have different dimensions. When different dimensions are used for the two legs we call it asymmetric compression within an attention layer. Results for this asymmetric compression are given in Section 3.6.

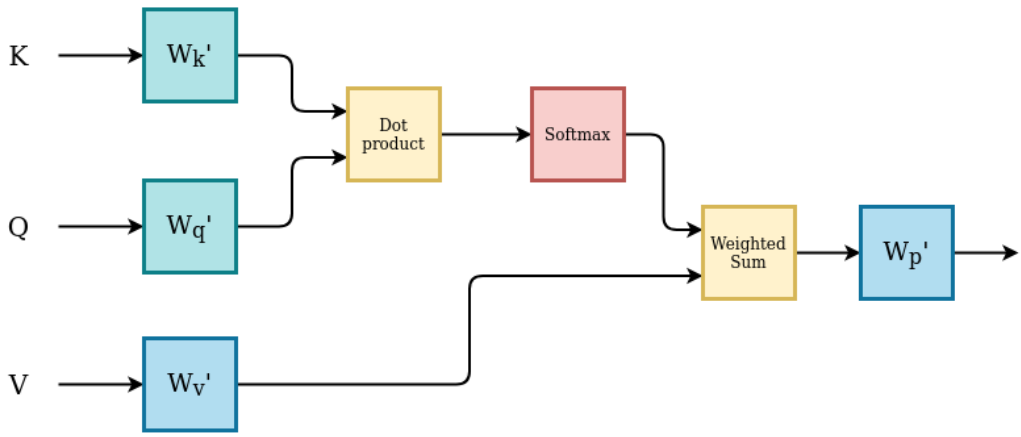


Figure 3.5: Reduced order attention

3.2.4 Asymmetry within or across layers

The results where I show compression by 2 (in Section 3.2.6), are for symmetric compression within an attention layer. For asymmetric compression within an attention layer, we can expect results to be somewhere between symmetric reduced order attention and no dimension reduction.

It is also evident that we can run some attention layers with compression or without compression. This is what I call asymmetry across attention layers. The results are in Section 3.6, apply asymmetry across linear layers.

3.2.5 Complexity of Components of Attention Layers for Reduced Dimension

The equation for attention is given in the Equation 3.7. The computational complexity is defined in terms the following variables.

$$\text{Target Length} = n_t$$
$$\text{Source Length} = n_s$$
$$\text{Embedding dimension} = d$$
$$\text{Reduced embedding dimension} = d'$$

Now we discuss the computational complexity for the three pre-attention linear layers shown in Figure 3.5. Each embedding has a dimension of d and the application of the linear layer requires d' set of weights. The number of embeddings in Q, K, V and P are the same as before:

$$O(QW_q) = n_t \times d \times d' \tag{3.1}$$
$$O(KW_k) = n_s \times d \times d' \tag{3.2}$$
$$O(VW_v) = n_s \times d \times d' \tag{3.3}$$

The output linear layer has the opposite input and output dimensions but complexity remains the same for each embedding:

$$O(PW_p) = n_t \times d \times d' \tag{3.4}$$

To compute the first dot product we should note that Q contains n_t vectors and K contains n_s vectors. We compute dot products for each pair of vectors, constructed with one vector each from Q and K. The dot products take $2d'$ operations since d' multiplications and d' additions are required. We get the following computational complexity:

$$O(QK^T) = n_t \times n_s \times d' \tag{3.5}$$

This equation requires n_t computations of dot products of vectors of size n_s for each dimension of V i.e. d. The computational complexity is hence:

$$O(A_wV) = n_t \times n_s \times d' \tag{3.6}$$

The FLOPs for this baseline implementation of attention for typical values is given below:

1. $n_t = 30$, $n_s = 33$, $d = 512$
2. $FLOPS(W_qQ) = FLOPS(W_PP) = 30 \times 2 \times 256 \times 512 = 7,864,320 = \text{A}$
3. $FLOPS(W_kK) = FLOPS(W_vV) = 33 \times 2 \times 512 \times 256 = 8,650,752 = \text{B}$
4. $FLOPS(QK^T) = 30 \times (2 \times 256) \times 33 = 506,880 = \text{C}$
5. $FLOPS(A_wV) = 30 \times (33 \times 2) \times 256 = 506,880 = \text{D}$
6. Total = 2A + 2B + C + D = 34,043,904 (2x faster)

Reduced Attention	RF enc	RF dec	RF crs	Validation BLEU	Test BLEU
Baseline [512]	1	1	1	37.48	36.81
Baseline [256]	-	-	-	36.29	36.45
Baseline [128]	-	-	-	33.31	31.48
Enc Comp [512]	2	1	1	36.81	35.98
Enc Comp [256]	2	1	1	36.31	36.40
Dec Comp [512]	**1**	**2**	**1**	**37.31**	**37.69**
Dec Comp [256]	1	2	1	36.22	36.22
Cross Comp [512]	1	1	2	37.38	36.45
Cross Comp [256]	1	1	2	36.43	34.89
All Comp [512]	2	2	2	36.15	36.77
All Comp [256]	2	2	2	36.39	36.18

Table 3.1: Results for symmetric reduced order attention. * The square brackets show the embedding dimension. Enc stands for encoder attention, Dec stands for decoder attention and Comp stands for compressed.

3.2.6 Results

The results for symmetric reduced order attention are computed by training and testing on the English-German Multi30K dataset [16] and given in Table 3.1. The sentences are tokenized before being used for training. German is used as source while English is used as target. The cross entropy loss is used to judge performance of the architecture during training. The Adam optimizer is used to minimize the loss. For validation and testing, the model is evaluated using the BLEU score.

3.2.7 Effect of Reducing Order of Attention

The results of transformers generally improve by increasing the embedded dimension as well as increasing the layers. Usually, increasing the layers is more useful than increasing the embedded dimension [4]. Higher embedding order improves accuracy of transformers with a computational complexity trade-off. Reducing order of attention linearly reduces computational complexity of attention layers. In inference, decoder runs sequentially, and has two attention layers. Thus improvement is magnified in the decoder during inference. Our trick enables running the transformer at higher embedded dimension (attention runs

in reduced order) with less computational requirements.

3.3 A Different Choice of Non-linearity

Now, we successively approximate softmax by activation functions that are similar to the exponential activation function. Figure 3.6 shows a plot of the activation functions that we will try.

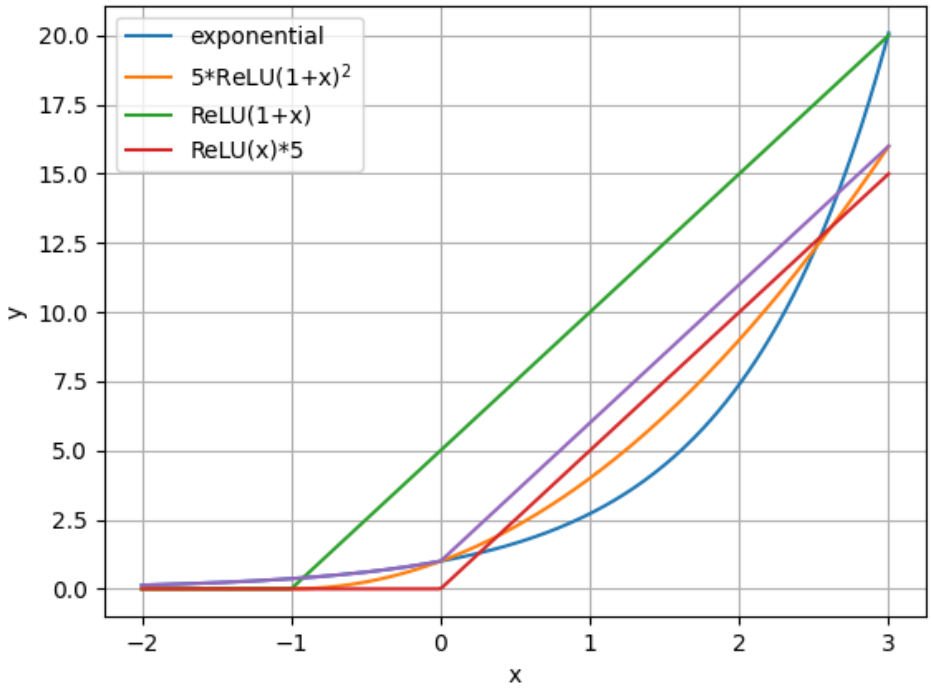


Figure 3.6: Plots of approximations of softmax

Note that softmax has not been plotted since it is a complex multi-dimensional function which is not amenable to a 2-D plot. Meanwhile, Exponential is a simpler function than softmax and its value depends only on a single variable. Hence, it has been shown in the plot given above.

Approximation	Validation BLEU	Test BLEU	Time complexity
Softmax	**37.48**	**37.72**	$O(n_t n_s \log^2 e)$
Exponential	36.58	36.31	$O(n_t n_s \log^2 e)$
ReLU(1+x)^2	36.11	35.46	$O(n_t n_s)$
ReLU(1+x)	36.40	33.27	$O(n_t n_s)$
RELU(x)	36.46	36.39	$O(n_t n_s)$

Table 3.2: Results for approximations of softmax. n_t is target length, n_s is sequence length and e is the number of digits in the exponential.

3.3.1 Results of Approximations of Softmax Applied in Cross Attention

The results in this section are computed by applying the transformer for machine translation on the Multi30K dataset. Adam optimizer is used to optimize the model with respect to cross entropy loss. Each batch consists of 800 tokens. The learning rate is 0.00015. A dropout of 0.1 is used. Table 3.2 shows the results for approximations of softmax.

The second entry ReLU(1+x)^2 is the closest to softmax and, it is tried first. Its results are reasonably good. Next, we remove the power in the expression of the second activation function. This leaves us with a shifted ReLU function. Unfortunately, results for this activation function are not good. Now we try a simple ReLU without any shift. We can see that a simple ReLU performs reasonably well. This is interesting since ReLU is less computationally expensive being $O(n_t n_s)$ compared to softmax which is $O(n_t n_s \log^2 e)$, where e is the number of digits in the computational complexity.

In the complexity of softmax, the $\log^2 e$ expression corresponds to computing the exponential. The complexity of the exponential is described in Ahrendt et al. [1].

In practice, the time gain is not very significant due to the relatively low number of operations that involve the computation of the softmax. The average time was roughly 15 seconds per epoch for all the activation functions.

Encoder Heads	Decoder heads	Cross Heads	Validation BLEU	Test BLEU	Time (sec)
8	8	8	37.70	37.42	11
4	8	8	37.75	36.86	12
2	8	8	37.83	37.49	11
1	8	8	37.60	37.42	11
8	**4**	**8**	**37.77**	**37.84**	**11**
8	2	8	37.72	37.17	11
8	1	8	37.79	37.49	11
8	8	4	37.60	36.98	12
8	8	2	37.28	37.42	11
8	8	1	37.59	37.51	11

Table 3.3: Results for asymmetric use of heads

3.4 Asymmetric Use of Heads in Different Attention Layers

In this work, we suggest the use of a different number of heads for each of the attention layers. This is named the asymmetric use of heads in different attention layers. This is different from previous work where the number of heads were kept the same for each of the attention layers.

3.4.1 Interesting Results w.r.t Heads

In Table 3.3, the results for asymmetric use of heads are given. We can see a variety of results in the table. We can see that some of the combinations give better performance than the baseline, which has 8 heads in each of the attention layers. This baseline can be seen in the first row of the table. The best result achieved with the asymmetric use of heads is shown in bold text.

3.5 Significance of Linear Layers in Self-Attention

We also studied the significance of linear layers in the attention layers. In this section, we show that the cross attention can function reasonably well without the internal linear

layers. However, if we remove linear layers in self attention of encoder or decoder, the attention layer collapses down to a simple identity matrix. Thus, for self-attention to work properly, it needs to have a projection mechanism in place for the inputs.

In particular, for the self-attention to work, at least one of Q or K needs to be projected to a different space via linear transformation.

3.5.1 Mathematical Equation of Attention?

$$V' = W_P \left(softmax \left(\frac{(QW_q^T)(KW_k^T)}{\sqrt{d_k}} \right) (VW_v^T) \right) = W_P A_w (VW_v^T) \tag{3.7}$$

The equation above is the equation of attention. The matrices W_q, W_k, W_v and W_p show the application of linear layers to Q, K, V and the output of softmax. In the next section we analyze the above equation for the case of self attention.

3.5.2 The Collapse of Softmax to an Identity Matrix

As we can see in the diagram of transformers (Figure 2.3), in self-attention, query and key are both the same. This means that:

$$Q = K \tag{3.8}$$

This is different from cross attention where the query comes from the decoder while key comes from the encoder hidden states.

The correlation between the same vectors is expected to be higher than the correlation between vectors that are not the same. This leads us to ask whether the result of the dot product QK^T is a diagonally dominant matrix when $Q = K$. If the square symmetric matrix QK^T is diagonally dominant, we are left with an interesting proposition. The application of softmax to this matrix would result in the diagonal dominance of the matrix being exaggerated. That is to say that the application of softmax will bring the matrix close to being an identity matrix.

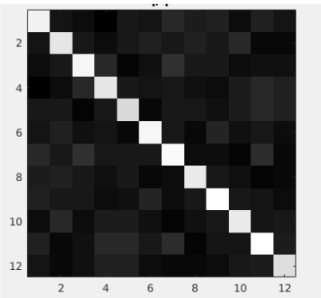
$$softmax \left(\frac{QK^T}{\sqrt{d_k}} \right) \approx I \tag{3.9}$$

The dot product of equal magnitude vectors will always be smaller if the vectors are not aligned. This means that under the assumption that the vectors have the same magnitude, the dot product of a vector with itself will be greater than its dot product with a distinct vector.

For $K = Q$, each value on the diagonal of QK^T corresponds to the dot product of a vector with itself. The values on the diagonal will thus be higher than the values off the diagonal which correspond to the dot product of a vector with a different vector. This means that we will have a diagonally dominant matrix. The application of softmax will make the diagonally dominant matrix approach to being an identity matrix.

What Do the Numerical Results of $QK^T/\sqrt{d_k}$ Say?

The transformer is trained on the Multi30K dataset for the English to German task. Once the transformer is trained, the matrix QK^T is computed (where Q is the queries matrix and K is the values matrix). The numerical values are plotted in Figure 3.7. The result shows that the matrix is indeed a diagonally dominant matrix, with the vector dot-products on the diagonal being much greater than the values off the diagonal.



46	0	-2	-5	0	-1	4	1	2	-2	2	0
0	44	1	-2	0	2	1	2	1	4	-3	-3
-2	1	47	4	-4	-1	6	1	1	-2	-1	-1
-5	-2	4	43	0	0	0	-1	-2	1	4	2
0	0	-4	0	41	-3	0	0	-1	1	4	2
-1	2	-1	0	-3	48	0	-3	3	-1	1	-2
4	1	6	0	0	0	48	-2	-2	-4	4	-3
1	2	1	-1	0	-3	-2	44	1	-1	-4	-3
2	1	1	-2	-1	3	-2	1	49	0	0	-2
-2	4	-2	1	1	-1	-4	-1	0	45	0	1
2	-3	-1	4	4	1	4	-4	0	0	49	1
0	-3	-1	2	2	-2	-3	-3	-2	1	1	42

Figure 3.7: Value of encoder QK^T

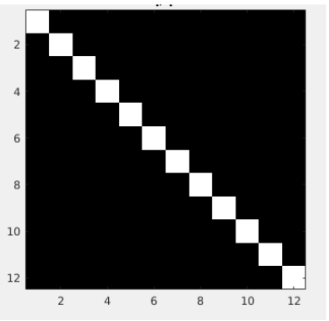
The next question that arises upon verifying the diagonal dominance of QK^T, is whether the values of the diagonal will increase relative to the values of the diagonal after softmax is applied to the matrix QK^T. This is discussed below.

Result of Normalization Performed by $softmax(QK^T/\sqrt{d_k})$

The softmax operation scales the values of each row of the matrix QK^T by an exponential. The values are then normalized to sum up to 1. In doing so, larger values which give

much higher values upon exponentiation grow much more than smaller values. After normalization, we expect the values on the diagonal to be close to 1.

Indeed our expectation is correct at least for this application, and we can see that we get a matrix that is so close to the identity that the differences are not visible with a few significant digits as shown in Figure 3.8. Therefore, we can replace $softmax(QK^T)$ by an identity matrix. This leads us to the realization that removing the linear layers applied to Q and K, would make the computation of QK^T a fruitless activity. In order for the operation to compute something of value, we cannot do without the linear layers.



1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1

Figure 3.8: Softmax applied to result of encoder QK^T

This confirms that in order for self-attention to be a practically useful operation the linear layers are definitely required since when they are applied Q and K no longer have the same vectors. When K and Q are different, QK^T is not diagonally dominant matrix. This makes self-attention a useful operation.

For the experimental results shown below, we use six layers of encoder and decoder. Figure 3.9 shows the value of QK^T for the first layer to the sixth layer in the encoder. When we go deeper into the layers, the correlation matrices change with each layer. However, the correlation matrix after softmax still stays close to and identity matrix. We get very similar results in decoder layers also.

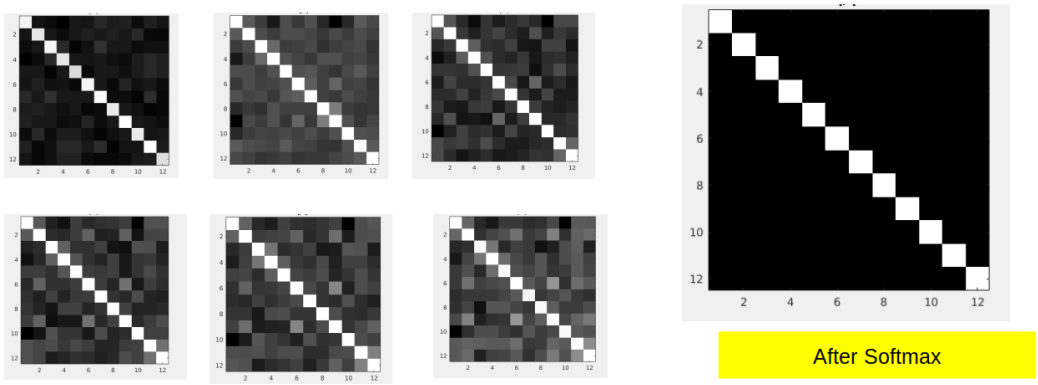


Figure 3.9: QK^T for each layer in the encoder. The layer number increases from left to right and top to bottom. The result on the right shows the results of the sixth layer after softmax

3.5.3 Selected Results

In Table 3.4, we can see that when we remove the linear layers from the encoder attention, we see a drop in performance. Similarly, when we remove linear layers from the decoder attention, we see a drop in performance. Meanwhile, in the first result in the table, we can see that when we remove linear layers from cross attention, the performance does not drop. To explain this we do an analysis in the following section.

In Table 3.4, the True and False choices for linear layers show whether linear layers were used in each of the three attention layers.

3.6 Neural Architecture Search in Transformers

3.6.1 Neural Architecture Search

There are many hyperparameters in Transformers. Hyperparameters in transformers include number of layers, embedded dimension, number of heads and feedforward dimension. In this work, I am proposing a few more hyperparameters. The two hyperparameters are lower dimensionality ratio of attention layer and type of non-linearity in attention.

Encoder Attention Linear Layers	Decoder Attention Linear Layers	Cross Attention Linear Layers	Validation BLEU	Test BLEU
False	True	True	36.07	35.99
True	False	True	36.14	35.38
True	**True**	**False**	**37.70**	**37.42**
True	False	False	34.83	35.10
False	True	False	35.43	34.93
False	False	True	34.71	31.32
False	False	False	33.24	33.10

Table 3.4: Results for removing linear layers

The next challenge is to find the "optimal" hyperparameters. The problem of searching for the best hyperparameters is not new. In NAS (Neural Architecture Search), we search for the best network configuration which achieves best results.

3.6.2 Neural Architecture Search Methods

There are different methods for finding out architectures. Some methods are given below:

- Reinforcement Learning
  - This is a very compute intensive method which requires a large number of data points in order for it to be effective. Each data point is generated after fully training the network. A controller is used to pick the next choice of the architecture. Zoph et al. used this method in NLP [37].
- Random Search
  - This method has been explored in Yoshua et al. [5]. There has also been more recent work.
- Train a Large Network and Prune the model to get a smaller model
  - The process involves training a large network, from which smaller networks can be derived.

3.6.3 Random Search Diagram

Figure 3.10 gives an idea of how random search works. We search randomly on the multi-dimensional grid of hyperparameters. Once we find the best configuration, we can optionally do a random search around the best option. And to refine things further, we can then do a grid search around the best option.

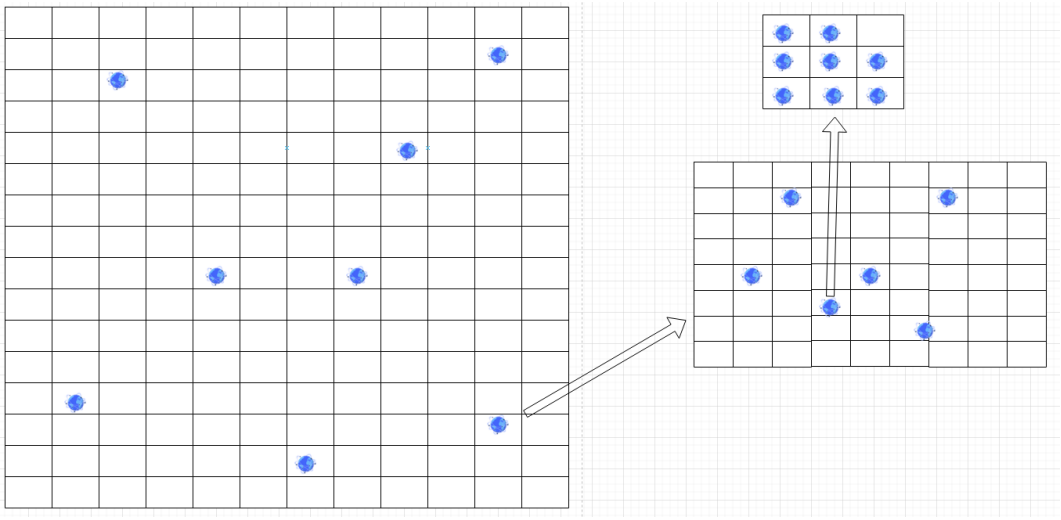


Figure 3.10: Random search diagram

3.6.4 Choice of Parameters

We decided to iterate over 15 parameters. There are 5 parameters in each of the attentions and there are 3 attention layers.

The five parameters of attention layers are:

- Number of heads
- Upper Leg Compression (K, Q)
- Lower Leg Compression (V, P)
- Upper Leg Linear Layers
- Lower Leg Linear Layers

Another optional choice of parameters is removing the linear layers in attention. However, there is a constraint while removing linear layers. To implement compression in a leg, we need to pass them through linear layers. For example, to implement compression in upper leg, we need to have linear layers which scale down K and Q. Similarly for compression in lower layer we need to have V and output pass through linear layers. For V we will need compression, and for final projection we will need scale up.

3.6.5 Results of Random Search

In Figure 3.11, the results of the random search have been plotted. The results are again computed for the English-German Multi30K dataset with the settings described in the previous sections. Since it is a random search, the configurations are random. To make the results of the search clearer we will sort the results by BLEU score in the next section.

Figure 3.11: Plot of random search

3.6.6 Results of Sorted Test Scores

The results from the previous section are plotted after being sorted based on the BLEU score. The plot is shown in Figure 3.12.[1]

It can be seen that most configurations lead to fairly good results. The results are mostly higher than 34 BLEU score and there are around four which are higher than 37 BLEU score.

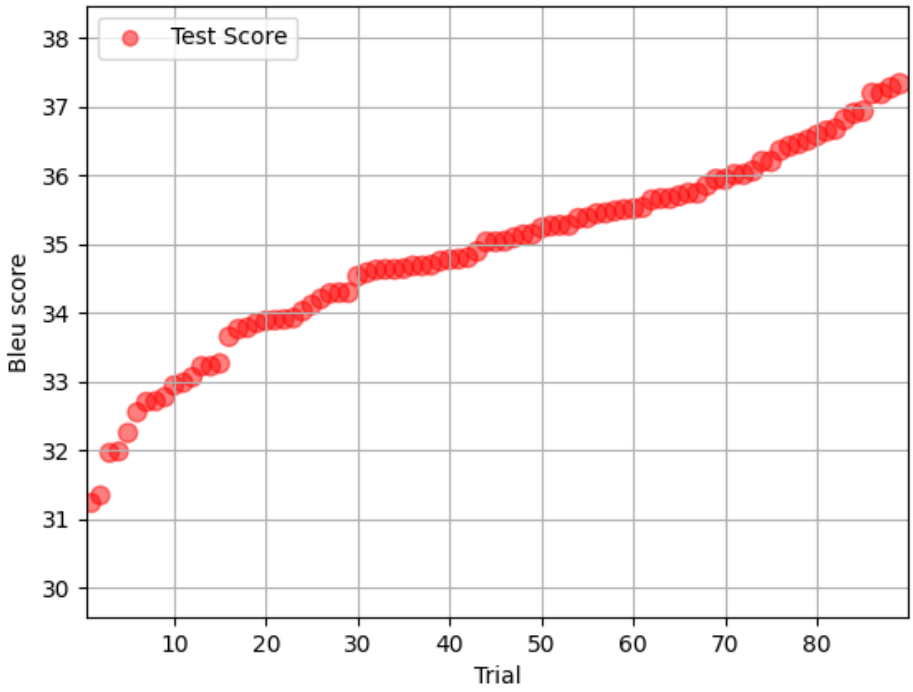


Figure 3.12: Random search results sorted by BLEU score for Multi30K dataset

3.6.7 Results of Validation and Test scores

To avoid "training" on the test set we sort by the validation score. A plot of validation and test scores sorted by increasing validation score, is shown in Figure 3.13.

[1]The best *options* are given in Section 3.6.8.

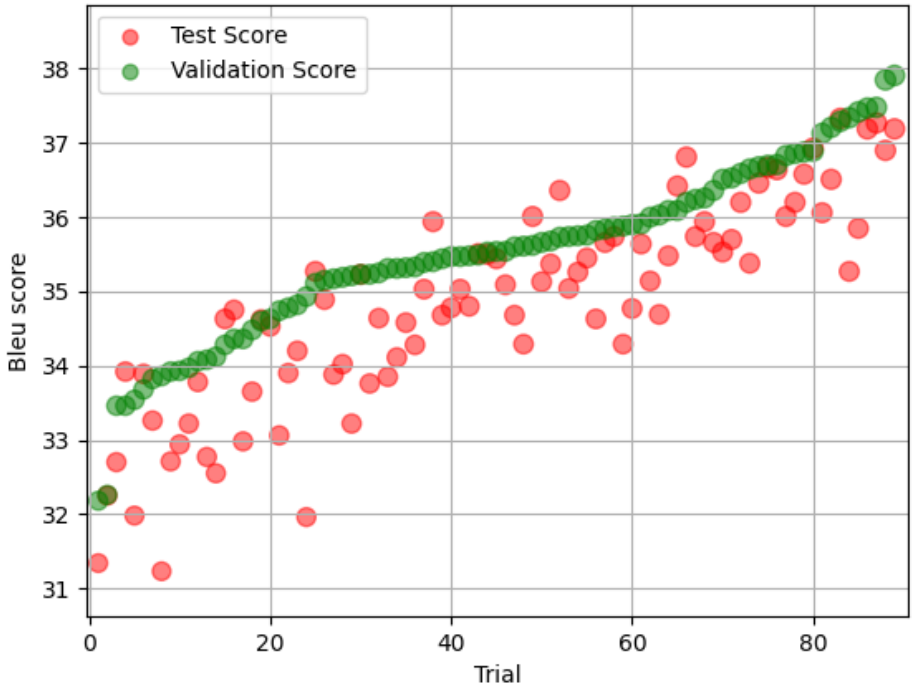


Figure 3.13: Plot of test and validation scores sorted by validation score for Multi30K dataset

3.6.8 Top 10 Results from Random Search

The top 10 options obtained by applying random search on Multi30K dataset are shown in Table 3.5. The values in the table are sorted by validation score. The non-linearity as mentioned in the table description can either be softmax (denoted by M) or RELU (denoted by R). $QKVP$ each denote the four linear layers in attention applied to query, key, value and output respectively. If the value is Y, it means that the linear layer is present. If it is N, it means that the linear layer is not present. The values of $QKVP$ can either be Y or N. "Heads" are the number of heads used in each of the attention layers. CompQK and CompKV denote the reduction in dimension for the upper leg and lower leg respectively. The variables described here have been expressed in short form in the table.

The state-of-the-art method for this dataset is [27]. Their method gives 39.7 BLEU

Rank	Encoder Attention	Decoder Attention	Cross Attention	Valid BLEU	Test BLEU
1	R-YYYY-2-11	R-YYNN-4-11	S-NNYY-2-14	37.91	37.19
2	R-YYYY-2-41	R-YYNN-2-11	S-YYYY-1-12	37.85	36.90
3	S-YYYY-8-12	S-YYYY-4-14	S-YYYY-1-11	37.48	37.27
4	S-YYYY-4-42	R-YYYY-2-11	S-YYNN-1-11	37.47	37.19
5	S-YYYY-2-41	S-YYYY-1-11	R-NNYY-1-12	37.42	35.85
6	S-YYYY-8-22	R-NNYY-2-14	S-YYYY-2-14	37.34	35.27
7	S-YYYY-2-21	R-YYYY-2-24	R-YYNN-2-11	37.29	37.33
8	R-YYYY-1-21	R-YYNN-8-11	S-YYNN-2-11	37.21	36.51
9	R-YYYY-4-14	R-YYYY-8-21	S-NNYY-4-14	37.13	36.06
10	R-YYYY-2-41	S-YYNN-4-21	S-YYYY-4-21	36.89	36.93

Table 3.5: Top 10 results from random search for Multi30K dataset. The short form is nonlinearity-QKVP-heads- compQK and compVP.

score which is slightly higher than the results achieved by our transformer implementation. The reason is that they use images in addition to the source text, giving their method an unfair advantage over our method which uses only the source text. We could have probably used the complete training/test harness of images and text, but our focus was to show that our new method of reduced order attention and other approximations, gives results comparable with the baseline we start off with. To simplify our training process, we decided to use only text processing, and our baseline for the Multi30K dataset with text processing only was 37.80 validation BLEU score. We achieved similar scores with reduced order attention and other proposed ideas.

3.6.9 Top 10 Result Conclusions

We analyzed the top 10 winners for features. The first point is that we can get a good idea of what configuration should work well. Interestingly, the winner with random search has two RELUs in Attention Layer. Based on the different results in Table 3.5, 2 heads is a good option and 8 heads does not seem to be a good option. Finally, randomization can pick options, one would never try intuitively.

3.7 Results Obtained on IWSLT

3.7.1 Plot of Results Obtained from Random Search

IWSLT, International Workshop on Spoken Language Translation, provided an English to German dataset in its 2014 edition. This dataset is 5 times larger than the Multi30K dataset. It includes complex sentences and is a more challenging problem. The results obtained by random search on IWSLT dataset are plotted in Figure 3.14.

The results show a close correlation between test and validation scores. The results are pretty good with a number of configurations scoring over 30 BLEU points. Note that the reason for running the experiment on IWSLT is that it is much larger than the Multi30K dataset previous experiments were run on.

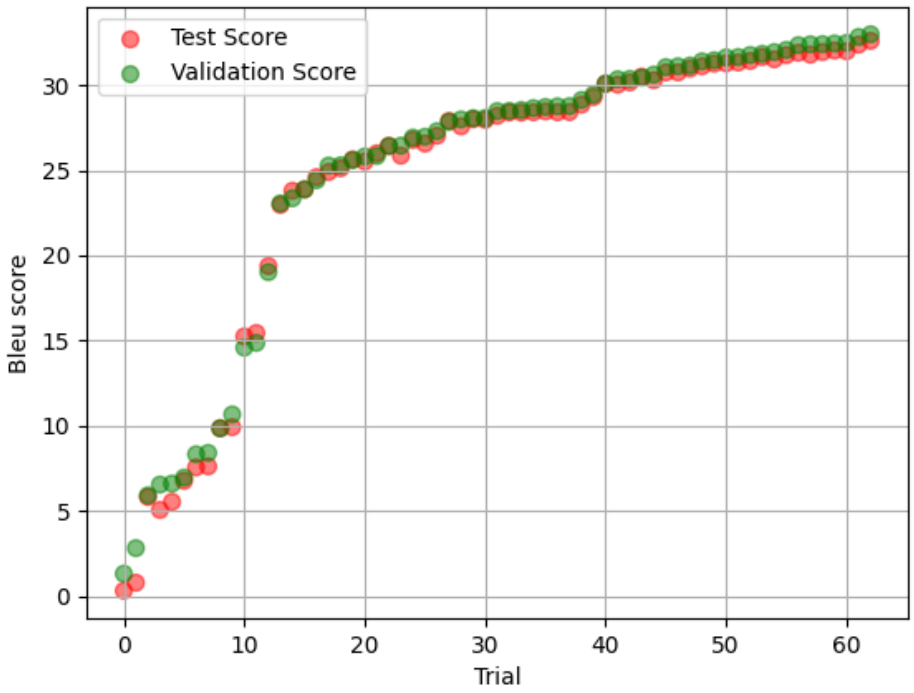


Figure 3.14: Plot of test and validation scores sorted by validation score on IWSLT

Rank	Encoder Attention	Decoder Attention	Cross Attention	Valid BLEU	Test BLEU
1	S-4-11	S-2-11	S-8-11	32.99	32.60
2	S-4-14	S-8-11	S-8-11	32.83	32.38
3	S-4-11	S-2-11	S-2-42	32.48	31.99
4	S-8-22	S-2-22	S-2-11	32.45	32.03
5	S-4-21	S-8-21	S-1-11	32.41	31.94
6	S-1-41	S-2-11	S-8-11	32.40	31.79
7	S-4-21	S-1-14	S-8-21	32.35	31.90
8	S-4-12	S-8-11	R-8-12	32.07	31.76
9	S-1-41	S-2-24	S-2-11	31.94	31.52
10	S-4-11	R-8-11	S-1-42	31.84	31.65

Table 3.6: Top 10 results from random search on IWSLT. In all the configurations linear layers are required.

3.7.2 Top 10 Results from Random Search for IWSLT 2014 Dataset

The top 10 results obtained by applying random search on IWSLT 2014 dataset are shown in Table 3.6. Again there are configurations where RELU has been used and good results have been obtained. Similarly, the dimension has been reduced up to a factor of 4 with good effect. The short form used in this table is the same as discussed in Section 3.6.8. Note that in the short form shown in the table, none of the configurations contains an "N" denoting absence of a linear layer. This is consistent with the theory which shows that linear layers are required in self-attention for self-attention to have an affect.

The best performing method [35] for this dataset gives 38.61 BLEU score. To prove the efficacy of our technique, we need a baseline. Which means that we need a model, a dataset, and the corresponding hyperparameters needed to produce the results. Then we use our approximations to indicate the applicability of our method. The best results we could reproduce were using the Fairseq reference code and we achieved validation BLEU score of 33.23, and set it as our baseline. Our proposed work again achieved results very similar to the baseline.

3.8 Results for Grammar Error Correction

For GEC we used the dataset by Roman et al. [17] for training purposes and tested on CoNLL 2014 dataset. The baseline, without our approximation gives an M2 score of 59.62.

Our results are summarized in Table 3.7. We get a speed gain of upto 17% with negligible loss in accurary. All the top 10 configurations run with softmax since the results with RELU are not as good for this dataset. It appears that with larger models RELU does not have the same performance as Softmax. Linear layers are used in all configurations to enable compression in attention.

The curve for the speed and accuracy trade-off is given in Figure 3.8. Note that, the cross represents the configuration running without any reduced order attention. This configuration clearly performs the worst among all the configurations in speed-accuracy trade-off plot.

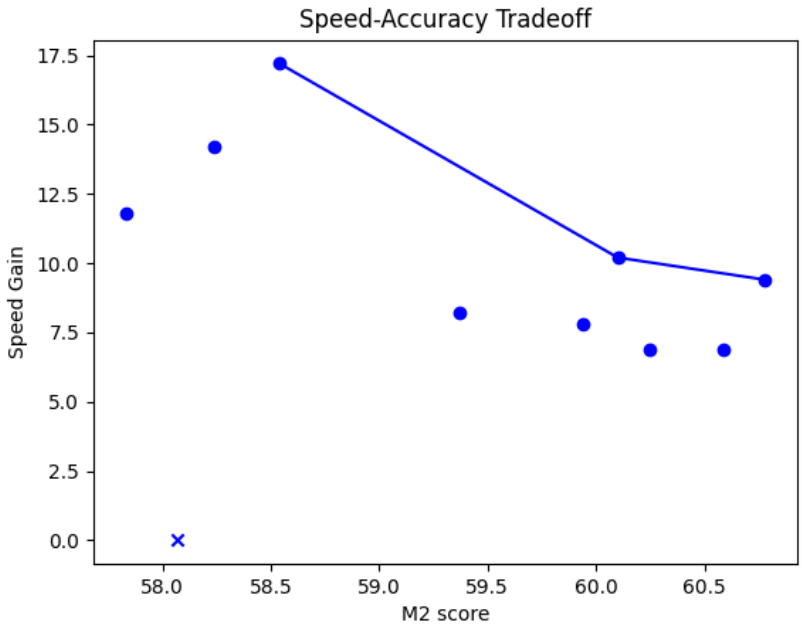


Figure 3.15: Effectiveness-Efficiency Trade-off Plot

3.9 Motivation for Neural Architecture Search

The same set of architectural configurations and hyperparameters cannot be expected to work across all datasets and tasks; that's the reason why we have to do the neural architecture search. Neural architecture search should be used to identify the best configuration

No.	Encoder Attention	Decoder Attention	Cross Attention	Valid M2	Test M2	Time	Reduction(%)
1	S-4-11	S-2-11	S-8-11	41.86	58.07	4:05	—
2	S-4-14	S-8-11	S-8-11	43.47	60.78	3:42	9.4
3	S-4-11	S-8-41	S-2-11	44.25	60.59	3:48	6.9
4	S-8-22	S-2-22	S-2-11	42.20	58.24	3:30	14.2
5	S-4-21	S-8-21	S-1-11	43.12	60.10	3:40	10.2
6	S-1-41	S-2-11	S-8-11	42.05	59.94	3:46	7.8
7	S-4-21	S-1-14	S-8-21	41.95	57.83	3:36	11.8
8	S-4-12	S-8-11	S-2-11	43.33	59.37	3:45	8.2
9	S-1-41	S-2-24	S-2-11	42.12	58.54	3:23	17.2
10	S-4-11	S-8-11	S-1-41	42.17	60.25	3:48	6.9

Table 3.7: Top 10 results from random search on Grammar Error Correction. Time is the time per epoch in hours and minutes (Hours:Minutes).

for a given task and dataset. Following this hypothesis, random search has been repeated on the various datasets and tasks instead of performing it on one dataset only and then simply reporting the results for the best configuration on the other tasks.

In performing the random search separately on each dataset and task, the hypothesis that the same architectural configuration does not perform the best on all datasets has been confirmed. For the first entry in Table 3.6, the configuration is the same as the entry in Table 3.7. However, the configuration is not the best performing configuration in the Table 3.7. It is in fact the third entry in Table 3.7 which has the highest validation score on grammar error correction. This shows that repeating independent neural architecture search for different datasets is required.

Due to the limitation of computational resources, exhaustive neural architectural search is not feasible today. However, subset of search space in each "dimension" can be identified to significantly reduce the overall search space. Thanks to the Vector Institute resources, computational resources helped us to utilize hundreds of hours of GPU compute power to verify our findings.

3.10 Computational Complexity of Asymmetric Reduced Order Cross Attention

This section shows the computational complexity for the asymmetric reduced order attention and compares it against original attention. Expressions written after the colon represent the computational complexity for the equation. The target and sequence length are denoted as follows:

$$\text{Target Length} = n_t, \text{ Source Length} = n_s$$

The dimensions of the input embeddings are as follows:

$$\text{embedding dimension} = d$$

The variables used here are as described in the Sections 2.3 and 2.3.2. Q, K and V are embedding matrices, while W_q, W_k, W_v and W_p are linear layers. It can be seen that the linear layers W_q and W_k have the same input dimensions. As we can see in Figure 3.16, the output dimension for both the linear layers is d'.

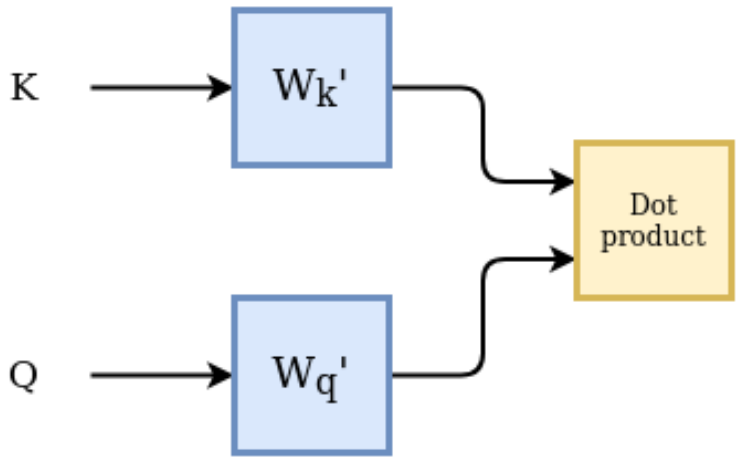


Figure 3.16: Attention Dimensions

The output dimension is the same because we need to compute the dot product of the vectors in Q with the vectors in K. These two linear layers operating at the same dimension constitute the upper leg of the attention layer. The complexity of the linear layers is then as follows:

$$O(W_q Q) = n_t \times d \times d'$$
$$O(W_k K) = n_s \times d \times d'$$

Taking the dot product of the matrix Q and the matrix K^T, gives us the following complexity:

$$O(QK^T) = n_t \times n_s \times d'$$

where $Q = QW_q$ and $K = KW_k$.

In the equation above, the output of the dot product is denoted by D. The operation applied to D is the softmax operation and the computational complexity for this operation is [1]:

$$O(softmax(QK^T)) = n_t \times n_s \times \log^2(e)$$

where e is the number of digits in the exponential of each element.

The weighted combination of V is computed using a second dot product in the attention layer after passing it through W_v. As described previously in Section 3.2.3, W_v does not have to be the same output dimension as W_k. We denote the output dimension of W_v as d''. The complexity of applying W_v is given below:

$$O(W_v V) = n_s \times d \times d''$$

The weighted sum block is the second yellow block in Figure 3.17 going left to right. The complexity is the following:

$$O(softmax(QK^T) \times V) = n_t \times n_s \times d''$$

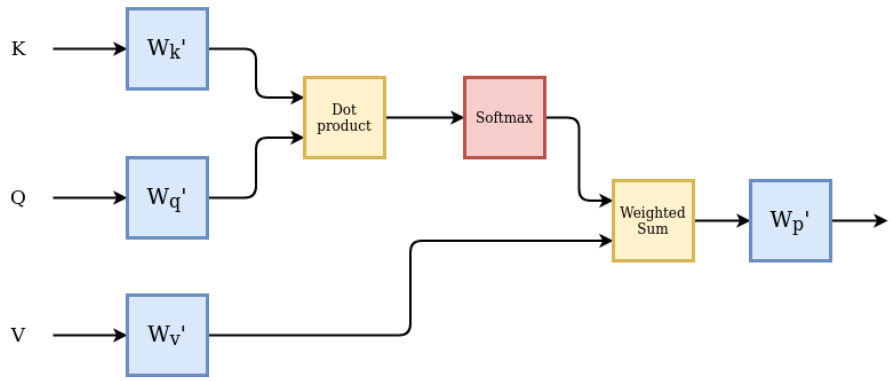


Figure 3.17: Dimension of result of Linear Layers W_v and W_p

The output of the weighted combination is passed through W_p and has the following complexity:

$$O(W_p \times (softmax(QK^T) \times V)) = n_t \times d'' \times d$$

where $V = VW_v$.

The total number of operations in the attention layer is given below:

$$\text{Total operations} = n_t dd' + n_s dd' + n_t dd'' + n_s dd'' + n_t n_s d' + n_t n_s d'' + n_t n_s d' + n_t n_s \log^2(e)$$

If $n_s = n_t$ and $d'' = d'$,

$$\text{Total operations} = 4n_t dd' + 2n_t^2 d' + n_t^2 \log^2(e)$$

3.10.1 Number of Computations for Fastest Configuration for Grammar Error Correction

Now, we compute the number of operations used in the fastest configuration in the Table 3.7.:

$$
\begin{aligned}
n_s &= 33 \\
n_t &= 30 \\
d &= 512 \\
e &= 10
\end{aligned}
$$

The number of FLOPs has been computed using the computational complexity described in the previous section.

The FLOPs for the Encoder Attention

1. $n_t = 30, n_s = 33, d = 512, d' = 256, d'' = 256$
2. $FLOPS(W_qQ) = 30 \times 2 \times 512 \times 256 = 7,864,320 = A$
3. $FLOPS(W_kK) = 33 \times 2 \times 512 \times 256 = 8,650,752 = B$
4. $FLOPS(W_vV) = 33 \times 2 \times 512 \times 256 = 7,864,320 = C$
5. $FLOPS(W_PP) = 33 \times 2 \times 256 \times 512 = 8,650,752 = D$
6. $FLOPS(QK^T) = 30 \times (2 \times 256) \times 33 = 506,880 = E$
7. $FLOPS(A_wV) = 30 \times (33 \times 2) \times 256 = 506,880 = F$
8. Total $= A + B + C + D + E + F = 34,043,904$

The FLOPs for the Decoder Attention

1. $n_t = 30, n_s = 33, d = 512, d' = 256, d'' = 128$
2. $FLOPS(W_qQ) = 30 \times 2 \times 512 \times 256 = 7,864,320 = A$
3. $FLOPS(W_kK) = 33 \times 2 \times 512 \times 256 = 8,650,752 = B$

4. $FLOPS(W_vV) = 33 \times 2 \times 512 \times 128 = 4,325,376 = C$
5. $FLOPS(W_PP) = 30 \times 2 \times 128 \times 512 = 3,932,160 = D$
6. $FLOPS(QK^T) = 30 \times (2 \times 256) \times 33 = 506,880 = E$
7. $FLOPS(A_wV) = 30 \times (33 \times 2) \times 128 = 253,440 = F$
8. Total $= A + B + C + D + E + F = 25,532,928$

The FLOPs for the Cross Attention

1. $n_t = 30, n_s = 33, E = 512, d' = 512, d'' = 512$
2. $FLOPS(W_qQ) = 30 \times 2 \times 512 \times 512 = 15,728,640 = A$
3. $FLOPS(W_kK) = 33 \times 2 \times 512 \times 512 = 17,301,504 = B$
4. $FLOPS(W_vV) = 33 \times 2 \times 512 \times 512 = 17,301,504 = C$
5. $FLOPS(W_PP) = 30 \times 2 \times 128 \times 512 = 15,728,640 = D$
6. $FLOPS(QK^T) = 30 \times (2 \times 512) \times 33 = 1,013,760 = E$
7. $FLOPS(A_wV) = 30 \times (33 \times 2) \times 512 = 1,013,760 = F$
8. Total $= A + B + C + D + E + F = 68,087,808$

The total number of operations across the three attentions is $127,664,640$. Without asymmetric compression, the total number of operations is $204,263,424$. The number of operations show a gain of 37.5%. This is higher than the 17% gain shown in Table 3.7. The reason for the time gain being lesser than shown numerically, is that the number of operations does not correspond directly to time improvement. Also the components besides attention contribute to the time taken, hence, the time gain is not directly proportional to the time improvement given by the computation shown in this section.

Chapter 4

Conclusion and Future work

4.1 Conclusion

Transformer architectures are the state of the art algorithms for sequence generation tasks. They are a significant improvement over their RNN predecessors in the sense that all the token of the target are predicted in parallel during the training phase. However, there is still a need to improve the training as well as inference speeds. Attention, being one of the core components of the transformers, has received lots of "attention" in recent literature in this regard.

Our work focuses on reducing the complexity of attention and takes a very different approach. The primary responsibility of attention layer is to "mix" the inputs by using a input dependent weighting matrix. We worked on computing an approximation of this weighting matrix by working in a reduced dimensional space. This was achieved by reducing the dimension of query, key, and value vectors. It is worth noting that even in the reduced dimensional space, the order of weighting matrix is still the same as in the conventional implementation. Traditionally, all the attention layers run in the embedded dimension space. In fact, all of the transformer runs in this space. By using our approximation, different attention layers can run at different resolutions. Going a step further, we can have a different resolutions for weighting matrix and a different resolution for the dot product, making multi-resolution implementation of attention layer possible.

This technique can be applied to any of the encoder, decoder, or cross attention layers. This technique can be applied in symmetrical or asymmetrical manner. Asymmetrical number of heads in different attention layers was explored. We also approximated softmax

with a faster alternative, e.g., ReLU. Since, we deep dived into the attention mechanism, we did an analysis of projection matrices of Q, K, and V for self attention and showed that without these projection matrices, self-attention layers collapse to an identity matrix.

4.2 Limitations

We discussed architectural modifications in transformers to achieve reduction in computational complexity with negligible compromise in accuracy. Since most of the proposed modifications are architectural modifications, these are not directly applicable for fine-tuning of pre-trained transformers. One will need to retrain the backbones in order to use our techniques.

Another limitation of our proposed technique is increase in number of computations during training. The extensive neural architecture search leads to significant increase in computations during training. However, as indicated in Chapter 1, our primary focus was to reduce the inference speed, and increase in number of computations during training is not as serious a bottleneck for many applications.

4.3 Future work

We expect our proposed architecture for computing the weighting matrix at a reduced dimension in the attention module to be widely applicable. We have tried our approximation in the context of NLP - more specifically for GEC and NMT. It will also be worth exploring how our work can be extended to language models which involve attention mechanism. It can not only be extended to other NLP tasks, but can also be extended to other domains like vision, since vision transformers (Dehghani et al. 2021 [15]) also use a similar attention mechanism.

We can also piggyback our approximation over other novel approaches for complexity reduction of attention layer, e.g., Katharopoulos et al. [21]. In works like these, our method will have an even greater impact on the complexity of attention. It would be interesting to explore how our work can combine with others which introduce new hyperparameters such as Jungo et al. [20], where they propose a deeper encoder and a shallow decoder, which gives better inference performance. Our work can even improve this further.

References

[1] Timm Ahrendt. Fast computations of the exponential function. *Lecture Notes in Computer Science*, Volume 1563, 2002.

[2] Abhijeet Awasthi, Sunita Sarawagi, Rasna Goyal, Sabyasachi Ghosh, and Vihari Piratla. Parallel iterative edit models for local sequence transduction. *Empirical Methods in Natural Language Processing*, 2019.

[3] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *Neural Information Processing Systems*, 2016.

[4] Ankur Bapna, Mia Chen, Orhan Firat, Yuan Cao, and Yonghui Wu. Training deeper neural machine translation models with transparent attention. *Association for Computational Linguistics*, 2018.

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 2012.

[6] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. *Association for Computational Linguistics*, 2017.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, and et al. Language models are few-shot learners. *arXiv:2005.14165*, 2020.

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

[9] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv:1409.1259*, 2014.

[10] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *International Conference on Learning Representations*, 2016.

[11] Daniel Dahlmeier and Hwee Tou Ng. Better evaluation for grammatical error correction. *NAACL*, 2012.

[12] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *Association for Computational Linguistics*, 2019.

[13] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *International Conference on Learning Representations*, 2019.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Empirical Methods in Natural Language Processing*, 2018.

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations*, 2021.

[16] Desmond Elliott, Stella Frank, Khalil Sima'an, and Lucia Specia. Multi30k: Multilingual english-german image descriptions. *arXiv:1605.0045*, 2016.

[17] Roman Grundkiewicz, Marcin Junczys-Dowmunt, and Kenneth Heafield. Neural grammatical error correction systems with unsupervised pre-training on synthetic data. *Association for Computational Linguistics*, 2019.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[19] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *International Conference on Learned Representations*, 2022.

[20] Jungo Kasai, Nikolaos Pappas, Hao Peng, James Cross, and Noah Smith. Deep encoder, shallow decoder: Reevaluating non-autoregressive machine translation. *International Conference on Learning Representations*, 2021.

[21] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. *International Conference on Machine Learning*, 2020.

[22] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *International Conference on Learning Representations*, 2019.

[23] Shun Kiyono, Jun Suzuki, Masato Mita, Tomoya Mizumoto, and Kentaro Inui. An empirical study of incorporating pseudo data into grammatical error correction. *Empirical Methods in Natural Language Processing*, 2019.

[24] James Lee-Thorp, Joshua Ainslie, Ilya Eckstein, and Santiago Ontanon. Fnet: Mixing tokens with fourier transforms. *arXiv:2105.03824*, 2021.

[25] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Association for Computational Linguistics*, 2020.

[26] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint*, 2019.

[27] Huan Lin, Fandong Meng, Jinsong Su, Yongjing Yin, Zhengyuan Yang, Yubin Ge, Jie Zhou, and Jiebo Luo. Dynamic context-guided capsule network for multimodal machine translation. *arXiv:2009.02016*, 2020.

[28] Kostiantyn Omelianchuk, Vitaliy Atrasevych, Artem Chernodub, and Oleksandr Skurzhanskyi. Gector – grammatical error correction: Tag, not rewrite. *15th Workshop on Innovative Use of NLP for Building Educational Applications*, 2019.

[29] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. *WMT*, 2018.

[30] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. *Association for Computational Linguistics*, 2002.

[31] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[32] Sascha Rothe, Jonathan Mallinson, Sebastian Krause Eric Malmi, and Aliaksei Severyn. A simple recipe for multilingual grammatical error correction. *arXiv:2106.03830*, 2021.

[33] Sho Takase and Shun Kiyono. Lessons on parameter sharing across layers in transformers. *arXiv:2104.06022*, 2021.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Neural Information Processing Systems*, 2017.

[35] Haoran Xu, Benjamin Van Durme, and Kenton Murray. Bert, mbert, or bibert? a study on contextualized embeddings for neural machine translation. *Empirical Methods in Natural Language Processing*, 2021.

[36] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. *Neural Information Processing Systems*, 2020.

[37] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *Conference on Computer Vision and Pattern Recognition*, 2018.