Reed Shay Final Report

**Introduction**

For my project I decided to tackle a relatively new development in Natural Language Processing problems (NLP). NLP problems are is an important field of study that focuses on the interaction between human language and computers. Language translation, one of the key applications of NLP, has become increasingly essential in today's globalized world.

In my research, I learned about the evolution of these types of problems. From in class discussions about RNN’s to my research findings of Statistical machine translation, the field of NLP is massive. In recent years, several new approaches called Neural Machine Translations (NMT) have gained popularity, employing deep learning. One specific model, which I focus on, is the Transformer.

The Transformer was introduced in the paper [“Attention is All You Need”](https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf) by Vaswani et al. in 2017 and has been the framework for my whole project. The purpose of this report is to explore the use of the Transformer architecture for language translation. We will provide a detailed description of the Transformer architecture and its key components, as well as the hyperparameter tuning process. Furthermore, we will discuss our approach to building a Transformer-based NMT model for English-Spanish language translation, including the dataset used and the preprocessing steps taken. Finally, we will present our results and provide an analysis of the model's performance.

**Diagram

Description automatically generated**

**Transformer Structure**

(I will briefly and broadly go over each component of the transformer)

The transformer architecture consists of two main components: the encoder and the decoder. The encoder takes an input sequence and transforms it into a set of hidden state representations, while the decoder takes those representations and generates an output sequence. Each component, encoder/decoder, is a pair of identical layers. For example, in the paper they use num\_layers = 6, so the encoder is run 6 times before being processed in the decoder. It is important to note that the actual learning and connection between the two is at the second Multi-Head Attention layer in the decoder. This layer takes in the output from the encoder as input and connects it to the target sequence in the Add & Norm layer (by residual connection).

**Positional Encoding**

Positional encoding is used to introduce positional information into the model, as traditional RNNs do not naturally encode this information. It is a crucial component of the transformer architecture, as it provides positional information into the model, unlike traditional models that process sequences sequentially, transformers process inputs in parallel.

* + Element wise addition of either :
    - (Even indices)
    - (Odd indices)
    - Where pos is the position of the word, i is the dimension of encoding and d is the dimension of the positional encoding.

One of the most important aspects of the transformer architecture is the use of multi-head attention. Multi-head attention allows the model to focus on different parts of the input sequence simultaneously, enhancing its capacity to capture complex relationships and dependencies. This is a significant departure from traditional attention mechanisms, such as Recurrent Neural Networks (RNNs), which only allow the model to focus on one part of the sequence at a time. Despite the increased complexity, these attention heads are run in parallel, ensuring that the model remains efficient and not too slow.

**Multi-Head Attention**

1. Input is projected into three different spaces, V K & Q.
   * Using separate learned weighted matrices for each head
   * i.e. V =

2) Scaled dot-product attention

* + For each attention head (running in parallel), the attention is calculated as :
  + This mechanism computes the importance of each input value with respect to the queries –> determining how much each input value contributes to the final representation of the sequence.
  + The scaling factor, sqrt(d\_k), helps to prevent gradients from becoming too small in deeper networks.

3) Concatenation and Transformation

* After computing the attention for each head, the results are concatenated, and a linear transformation is applied to produce the final output. This step allows the model to combine the learned features from each head, enabling it to capture a diverse range of information from the input sequence.

4) Add & Normalize layer

* Lastly a residual connection is added to the output from the multi-head attention, followed by a layer normalization step. This helps to stabilize the learning process and ensures that the model can effectively learn from the combined information of all the attention heads.

Also, we have to note that the Decoder is using what is referred to as a Masked Multi-Head Attention layer initially. This is fundamentally different from our normal Multi-Head Attention as the sequence is not processed holistically but in an ‘upper-triangular’ like manner, enforcing that the model only knows previous and current positions (if we gave it the whole sequence, well then, the model would have the answer… and it would not learn). The main difference is the application of a masking matrix, which is an upper-triangular matrix.

**Feed-forward Layers**

In addition to multi-head attention, the transformer architecture also includes feed-forward layers and positional encoding. Feed-forward layers are used to introduce non-linearity into the model and allow it to learn complex relationships between the input and output sequences.

The FFN’s are simply two linear transformations, with a ReLU applied in-between layers.

1. First, we apply a linear transformation with weight matrix and bias.
   * Mathematically where x is the input.

2) Apply a ReLU activation function, elementwise to

3) Apply a second linear transformation

* + Mathematically

4) Add & Normalize layer

* Again, we have an add & normalize layer.
* It is worth noting and making sure you know where this ‘residual connection’ is coming from. We can see these as side arrows, branching into the Add & Norm layers.

With all these components, you can begin to create your transformer.   
Problem initially :

* R has no packages yet
* Update : [April 28th a package was released,](https://cran.r-project.org/web/packages/transformer/index.html) too late as my project has already finished.
  + Will make sure to continue the project using this

**Hyperparameter Decision Making**

Ideally, I would employ either Grid Search or Random Search to test the variety of parameter values. Unfortunately, transformers are terribly expensive and on a single laptop, this was made difficult. For example, my initial SIMPLE model took 3+ hours to run. This was with what I thought, in comparison to the paper, a tiny amount of estimated parameters. With this being said, it was difficult to really explore Hyperparameter values and most were simply copied from the paper.

* Learning Rate: We used the original paper's learning rate schedule, which involves linearly increasing the learning rate for a warmup period, followed by a decrease proportional to the inverse square root of the training step.
* Batch Size: We set the batch size based on the available computational resources (e.g., GPU memory) and the specific dataset being used. A common choice is 64 or 128, but this can be adjusted depending on the requirements of the problem.
* Number of Layers: We used the same number of layers as the original paper (6 layers for both encoder and decoder). This depth allows the model to learn complex relationships in the data while maintaining a reasonable training time.
* Embedding Size: The size of the word embeddings used in the model.
* Number of attention heads
* Feed-forward Network Size (DFF)
* Dropout Rate

As we can see, there are many, many different customizable parameters. If further process is to be made in this project, this would definitely need to be addressed.

Overall, the majority of my time was actually spent digesting what the transformer model is. It took way longer than expected to get through the paper but I feel like I have a sound understand now.

**Simple Transformer Architecture**

**Data**

The data used for training and testing the model consists of parallel English and Spanish sentences collected from various sources, such as news articles, websites, and books. The dataset is the Tatoeba Translation Dataset, with the ultimate goal of taking English sentences and translating them to Spanish.

**Model Architecture**

Encoder : consists of multiple layers of self-attention followed by layer normalization and dropout. Similar to the papers, but with no added FFN.

Decoder : also consists of multiple layers and has a similar structure to the encoder. The input to the decoder is the tokenized and padded Spanish sentences. Then passed through an embedding layer and layer normalization.

* The first multi-head attention layer is applied to the input.
* Dropout & layer normalization applied.
* Then, the second multi-head attention layer is applied between the decoder and encoder outputs. This allows the decoder to derive relevant parts of the encoder output when generating the translation.

Output Layer : after passing the decoder, the output is passed through a dense layer with a SoftMax activation.

Results from the training : Table

Description automatically generated

The training itself took over 3 hours, leading to not much exploration of the parameter space. Lets look at the results above.

1. **Training Loss and Accuracy**: The training loss decreases consistently over the 10 epochs, and the training accuracy increases. This indicates that the model is learning and improving its performance on the training data.
2. **Validation Loss and Accuracy**: The validation loss decreases initially and then starts to increase after epoch 5. The validation accuracy, on the other hand, increases over the 10 epochs. The increase in validation loss while accuracy continues to improve may indicate that the model is overfitting the training data. This is further supported by the gap between the training and validation accuracies.

Odds are, there is a potential overfitting issue, some things to consider :

1. Increasing dropout rate.
2. Implement early stopping.
3. Using more regularization.
4. Reducing model complexity (already simply…)

**In Summary & Next Steps**

Overall, I really enjoyed this project. I felt like I really understood the framework and the functions behind the transformer architecture. It is a little ridiculous how popular transformers are becoming, especially with how popular Chat-GPT models are.

Moving forward, I would really like to fix my prediction function. When running in python and I actually go to translate, I am running into an error. I need to debug and get this prediction to work, as the model has already been trained.

Also, the transformer() package in R has just recently been added. Initially, I started the project in R but after about 3-4 hours of writing source code, I realized it would take way too long for this project. So, finally seeing output of the model would be a really big step.

I have all the code attached in a .ipynb file, with the prediction function still flawed.

Overall, I learned a lot about transformers in this project and saw first hand just how powerful they an be.