ANLP Assignment 3 Report

Sankalp Bahad Roll Number: 2021114015

1 Theory

1.1 What is the purpose of self-attention, and how does it facilitate capturing dependencies in sequences?

Self-attention is a crucial mechanism in the field of natural language processing and deep learning, particularly prevalent in models like the Transformer. Its primary purpose is to enable models to understand and capture dependencies within sequences, such as sentences or time series data. It accomplishes this by allowing the model to focus selectively on different parts of the input sequence when making predictions, which is particularly valuable for tasks like machine translation, text summarization, and sentiment analysis.

The mechanism works as follows:

- Calculating Relevance: Self-attention begins by calculating a relevance score between each element (e.g., word) in the sequence concerning all other elements. This relevance score, often referred to as "attention," quantifies how much each element should focus on other elements.
- Weighted Summation: The relevance scores determine the weight or importance of each element's contribution to the current element's representation. The weighted summation of these contributions forms the new representation for each element. Importantly, elements relevant to the current context receive higher weights, while irrelevant elements receive lower weights. This weighted summation captures the context and dependencies within the sequence.
- Parallel Processing: Notably, self-attention operates in parallel for all elements in the sequence, making it computationally efficient and suitable for tasks requiring extensive modeling of dependencies.

The benefits of self-attention are as follows:

- Long-Range Dependencies: Unlike traditional recurrent neural networks (RNNs) that struggle with capturing long-range dependencies, self-attention can effectively model relationships between elements that are far apart in the sequence. This is crucial for understanding the meaning of a word in the context of a more extended sentence.

- Non-Sequential Processing: Self-attention processes all elements simultaneously, making it highly parallelizable and suitable for modern hardware like GPUs and TPUs. This parallel processing significantly speeds up training and inference.
- Adaptability: The model can learn which elements are contextually relevant for each element, making it versatile and capable of handling a wide range of sequences and tasks without manual feature engineering.

1.2 Why do transformers use positional encodings in addition to word embeddings? Explain how positional encodings are incorporated into the transformer architecture.

Transformers use positional encodings in addition to word embeddings to address a fundamental limitation of their architecture: the absence of inherent information about the order of words in a sequence. Unlike recurrent neural networks (RNNs) and convolutional neural networks (CNNs), which inherently capture sequential or spatial information, the self-attention mechanism in transformers treats inputs as a set of unordered elements. To enable the model to differentiate between words based on their position, positional encodings are introduced.

1.2.1 Purpose of Positional Encodings

Positional encodings are designed to inject information about the position of words into the input embeddings. This is essential because in tasks like language modeling, translation, and text generation, the order of words is of paramount importance. Positional encodings help transformers understand the sequential nature of data, allowing them to capture the dependencies within a sequence.

1.2.2 Incorporating Positional Encodings into Transformers

Positional encodings are added to the word embeddings before feeding the input sequence into the transformer model. The specifics of their integration into the architecture are as follows:

- Positional Encoding Vectors: Positional encoding vectors are generated based on the position of each word in the input sequence. The two most common methods for generating positional encodings are the sine and cosine functions of different frequencies. These functions create a unique vector for each position, and these vectors are added to the word embeddings.
- Element-wise Addition: The positional encoding vectors are element-wise added to the word embeddings. The addition occurs before the input embeddings are passed to the self-attention mechanism. This means that

each word embedding now contains information about its position in the sequence.

- Positional Information Preservation: Positional encodings allow the model to discriminate between words based on their positions while retaining the essential semantic information from the word embeddings. This ensures that words with the same lexical meaning but different positions in the sequence are treated differently by the model.

1.2.3 Significance

The incorporation of positional encodings is essential for transformers' ability to handle sequential data effectively. It enables the model to consider the position of each word when computing self-attention, allowing it to capture dependencies that rely on the word order. Without positional encodings, transformers would be limited to understanding only the content of words and not their positions, rendering them less capable of tasks involving sequences, such as natural language understanding, generation, and translation.

2 Explanation

2.1 Model Explanation

The Transformer model architecture for machine translation broadly consists of encoder and a decoder component, each of which comprises multiple layers of identical sub-modules.

- Encoder and Decoder: The encoder processes the source language sequence, while the decoder generates the target language sequence
- Stack of 2 Identical Layers: The encoder and decoder each consist of a stack of 2 layers. These layers are designed to capture hierarchical representations of the input sequences. The use of multiple layers allows the model to capture both low-level and high-level features.
- Attention Heads and Hidden States: Within each layer, self-attention is applied in parallel across multiple attention heads. The attention heads help the model to focus on different parts of the input sequences simultaneously. The number of heads can be changed in order to enable better attention mechanism. Hidden states are the intermediate representations within each layer, and they are computed using self-attention and feed-forward neural networks.
- Self-Attention Mechanism: Self-attention is a crucial component of the architecture that allows the model to weigh the importance of different words or tokens in the input sequence when making predictions. Self-attention operates on a set of queries, keys, and values. These are linear transformations of the input sequence. The mechanism computes weighted sums of values based on the compatibility between queries and keys. The multi-head self-attention mechanism allows the model to learn different attention patterns.
- Feed-Forward Neural Networks: Feed Forward Neural networks are applied to the self-attention output. They consist of fully connected layers and activation functions. They enable the model to perform complex, non-linear transformations of the data.
- Positional Encodings: Positional encodings are added to the input embeddings to provide information about the position of words in the sequence. These are typically learned during training to help the model understand the sequential order.

2.2 Training Procedure

The training procedure for the Transformer model involves the following steps:

2.2.1 Data Preparation

We preprocess the data, train, test and validation data. This involves splitting the sentences into word tokenized lists and then converting them into tensors after proper padding and adding ¡EOS¿ and ¡SOS¿ tags. We do this for both source and target sentences and create a parallel data loader.

2.2.2 Model Initialization

We initialize the transformer model with random weights that are later modified during the training process, we also define a set of parameters like model and feedforward dimensions which can be tuned later for better performance of the model.

2.2.3 Loss Function

We define a suitable loss function to measure the dissimilarity between the predicted sequences and the actual target sequences. This loss guides the model during training.

2.2.4 Optimization:

We choose an optimization algorithm, set hyperparameters like the learning rate and weight decay. These parameters control how the model's weights are updated.

2.2.5 Forward Pass

During each training iteration, we feed a batch of source sequences into the encoder, and target sequences into the decoder of the model. We first generate word embeddings, where words are converted into continuous vector representations. To ensure the model understands the position of words in a sequence, we add positional encodings. Then for every word in the source sentence, we enable our model to pay varying degrees of attention to all other words in the sentence, no matter how far apart they are. This way, we are able to capture dependencies between words effectively. We use multiple attention heads that work in parallel, each focusing on different aspects of these dependencies. They then combine their findings, allowing us to capture a wide array of relationships between words, making our model a powerful tool for tasks like machine translation.

2.2.6 Loss Calculation

We calculate the loss by comparing the model's predictions with the actual target sequences. Essentially, the loss quantifies the dissimilarity between what the model predicted and what it should have predicted. If the model's predictions align perfectly with the target sequences, the loss will be minimal. However, if there are discrepancies, the loss will be higher, indicating areas where the model needs improvement. Minimizing this loss is the primary objective during training, as it drives the model to make more accurate translations. So, the loss serves as a critical feedback signal, guiding the model towards better translation quality with each training iteration.

2.2.7 Backpropagation and Gradient Descent

After calculating the loss, the next step is to compute the gradients of the model's parameters with respect to that loss. This process is known as backpropagation. It essentially determines how each parameter should be adjusted to minimize the error. The gradients indicate the direction and magnitude of the changes needed in the model. After obtaining these gradients, we move on to updating the model's weights using the chosen optimization algorithm. The optimizer takes the gradients into account and updates the model's parameters accordingly. By iteratively adjusting the weights based on the gradients, our model gradually learns to make better predictions and improve its performance over time.

2.3 Hyperparameter Tuning

The hyperparameters that were tuned were batch size, dropout, learning rate and number of heads for attention. Here is how these hyperparameters affect the performance of model:

2.3.1 Batch Size

Batch size affects the quality of weight updates and the training speed of the Transformer model. Larger batch sizes can provide smoother gradients and more accurate weight updates, resulting in faster convergence. Smaller batch sizes can lead to noisier gradients, but they may generalize better and require less memory. The optimal batch size can vary depending on the dataset and the available hardware, so it often requires experimentation.

2.3.2 Dropout Rate

Dropout is a regularization technique used to prevent overfitting in Transformer models. The dropout rate controls the probability of dropping out neurons during training. A moderate dropout rate, such as 0.1, is commonly used to regularize the model without significantly impacting performance. Higher dropout rates can lead to stronger regularization but may hinder the model's ability to learn complex patterns. It's essential to strike a balance.

2.3.3 Number of Attention Heads

The number of attention heads in the model's multi-head self-attention mechanism affects its ability to attend to different parts of the input sequences. More attention heads can capture more complex relationships in the data but increase computational complexity. A typical choice is 8 attention heads, as it balances performance and efficiency for various tasks. If the task involves capturing highly complex dependencies, increasing the number of heads may be beneficial, but this requires more computational resources.

2.3.4 Learning Rate

The learning rate determines the step size during gradient descent and has a significant impact on the training process. A learning rate that is too high can cause the model to overshoot minima and diverge, while a learning rate that is too low can result in slow convergence. Learning rate schedules, such as learning rate warm-up and decay, can help the model converge more effectively. The ideal learning rate depends on the specific task, dataset, and model architecture, and it often requires hyperparameter tuning to find the best value.

The best quadruplet is batch size of 64, learning rate of 0.00001, number of heads as 16 and dropout rate of 0.1.

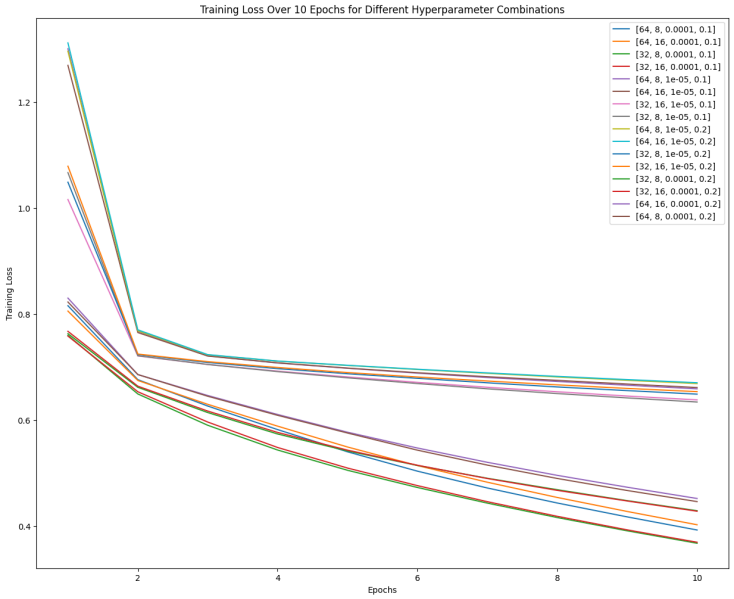

Figure 1: Train Loss

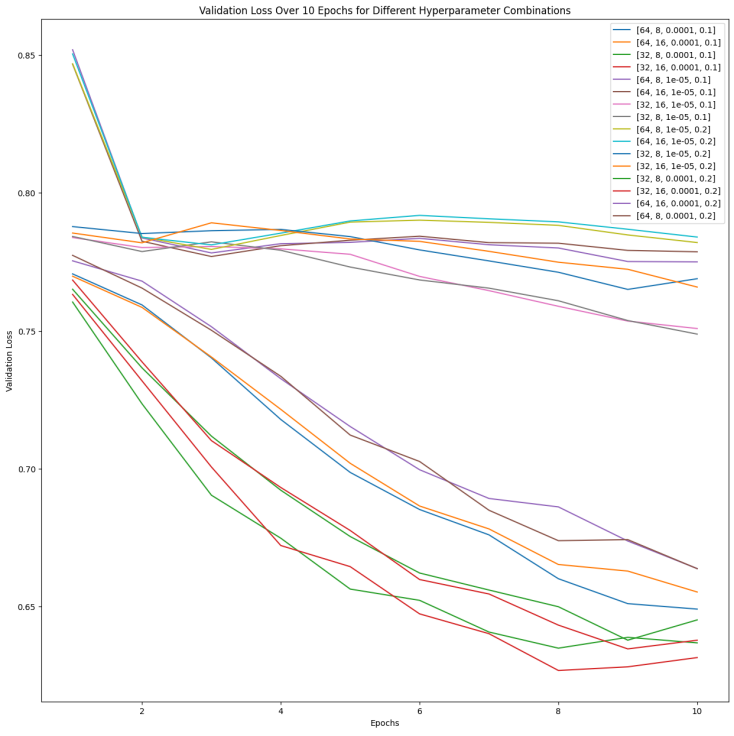

Figure 2: Validation Loss

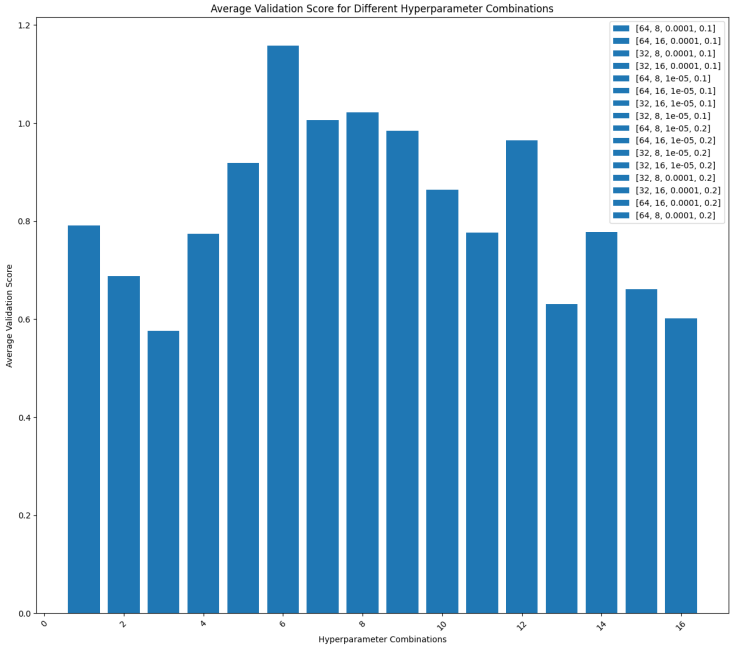

Figure 3: BLEU Score

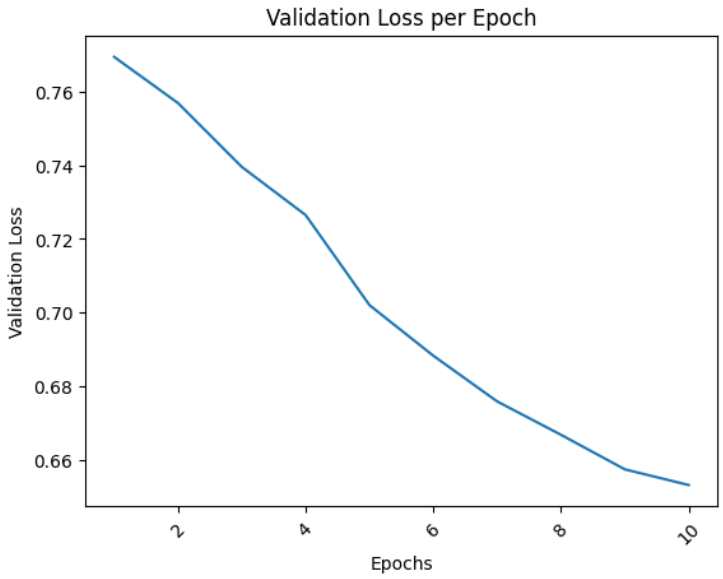

Figure 4: Train loss per epoch

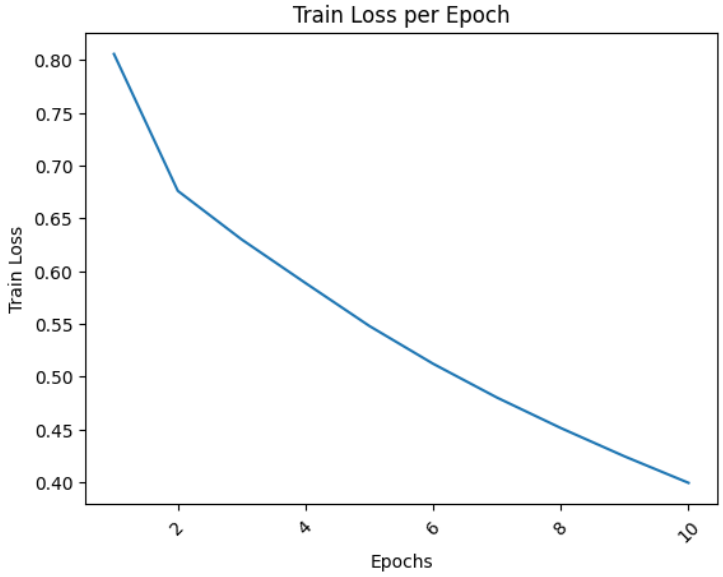

Figure 5: Validation loss per epoch

3 Analysis

This Transformer model designed for machine translation, has shown promisindecent performance. To evaluate its quality, we used the BLEU metric, a widely used measure for assessing the translation quality. The BLEU metric quantifies the similarity between machine-generated translations and reference translations based on n-gram precision and brevity penalty. In our analysis, we considered both the training and test datasets to provide a comprehensive assessment of the model's performance.

For the training dataset, we calculated BLEU scores for all the sentences. The bleu scores observed were decent, in the range of 0 to even 50 for some sentences. The model does perform well and predicts full words, but sometimes repeats the words. The main reason might be the shortage of data or short sentence lengths.

We can observe that during training, the loss is decreasing with every epoch but the validation bleu scores are not too good, the main reason being not using any pretrained model but training on less data from scratch.

This model is trained from scratch and there isnt a lot of data to train on due to which it is expected for this model to perform relatively poorly. Hence, from the bleu scores of the output sentences, we can see that the scores are not too good, even the validation scores during epochs is not too high.

It also gives decent results on test set, with slightly lower range than the train set results, due to the fact that this is unseen data. The model still performs well on unseen data and does not produce garbage words on most occassions

which means the model has learned well.

Below are the graphs for Train and Validation losses while training epochs. We have also attached files that have predictions, true sentences, and bleu scores for test and train data.