ANLP A2 Report

Question 1

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

Purpose of Self-Attention:
Self-attention is a mechanism used primarily in transformer architectures to enable models to weigh the importance of different tokens in a sequence relative to each other. The main purposes of self-attention are:

1. **Capturing Dependencies**: It allows the model to capture long-range dependencies in the data. Traditional models, like RNNs or LSTMs, often struggle with distant dependencies because the information may degrade as it travels through multiple layers or time steps. Self-attention, however, directly connects all tokens in a sequence, enabling the model to consider every token when processing a particular token.
2. **Dynamic Weighting**: Self-attention provides a way to dynamically compute the importance of each token in a sequence for a given token. This means that the model can focus on different parts of the input sequence based on the current context, which is particularly useful in tasks like translation, summarization, and sentiment analysis.
3. **Parallelization**: Unlike RNNs, where the input is processed sequentially, self-attention enables parallel computation since all tokens are processed simultaneously. This results in significantly faster training times on large datasets.

How Self-Attention Facilitates Capturing Dependencies:
The self-attention mechanism works as follows:

1. **Input Representation**: Each token in the input sequence is first represented as an embedding vector. This results in a matrix where each row corresponds to a token embedding.
2. **Query, Key, Value Vectors**: For each input embedding, the model computes three vectors:
   - **Query (Q)**: Represents the token for which the model seeks information.
   - **Key (K)**: Represents potential tokens that may provide relevant information.
   - **Value (V)**: Contains the information carried by the token.

Question 2

Why do transformers use positional encodings in addition to word embeddings? Explain how positional encodings are incorporated into the transformer architecture. Briefly describe recent advances in various types of positional encodings used for transformers and how they differ from traditional sinusoidal positional encodings.

Transformers are a type of neural network architecture that excels in processing sequences of data, particularly in natural language processing tasks. One key challenge they face is how to represent the order of tokens in a sequence since, unlike recurrent neural networks (RNNs), transformers process all tokens in parallel. To address this, they use positional encodings in addition to word embeddings.

Why Use Positional Encodings?

1. **Sequence Order Representation**:
    - Transformers do not inherently capture the sequential nature of data. In RNNs, the order of processing naturally encodes the position of each token. However, transformers treat each input token independently. Positional encodings ensure that the model has access to information about the position of each token in the sequence.
2. **Disambiguation of Meanings**:
    - The meaning of words can depend significantly on their positions within a sentence. For instance, in the phrases "The cat sat on the mat" and "The mat sat on the cat," the position alters the meaning. Positional encodings help the model to disambiguate such cases by maintaining order information.

Incorporation of Positional Encodings

Positional encodings are integrated into the transformer architecture through the following steps:

1. **Input Embedding**:
    - Each token in the input sequence is converted into a vector representation (word embedding). This captures semantic meanings but does not include positional information.
2. **Adding Positional Encodings**:
    - Fixed or learned positional encodings are generated for each position in the sequence. These encodings are then added to the corresponding token embeddings. The formula for traditional sinusoidal positional encodings, introduced in the original transformer paper (Vaswani et al., 2017), is given by:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Hyperparameter Tuning

Embedding Dimension	Number of Layers	Number of Heads	Dropout	Bleu Score	Avg Validation Loss
512	2	**8**	**0.1**	**0.018**	
512	2	16	0.2	0.04	
512	2	16	0.1	0.05	
300	2	6	0.1	0.13	
300	2	10	0.1	0.162	
300	4	6	0.1	0.168	
300	**6**	**6**	**0.1**	**0.279**	
300	6	10	0.1	0.126	

General Trend

Dropout

- The dropout rates tested were **0.1** and **0.2**.
- All configurations using a dropout rate of **0.1** generally produced better BLEU scores compared to **0.2**. For example:
  - **300, 2, 6, 0.1** achieved **0.168** while **300, 2, 16, 0.2** dropped to **0.04**.
- This suggests that a lower dropout rate helps retain more information during training, allowing the model to learn effectively. Higher dropout rates may lead to excessive regularization, causing the model to underfit.

Embedding Dimension:

- The embedding dimensions tested were 512 and 300.
- The configurations with an embedding dimension of **300** generally produced higher BLEU scores compared to those with **512**. For instance:
  - **300, 2, 6, 0.1**: BLEU Score = **0.168**
  - **300, 2, 10, 0.1**: BLEU Score = **0.162**

- In contrast, the best score with an embedding dimension of **512** was **0.05** with configuration **512, 2, 16, 0.1**. This suggests that a lower embedding dimension may have better representation capability in this context, potentially due to better generalization or less overfitting.

Number of Layers

- The number of layers varied between **2** and **6**.
- The model configuration with **6 layers** (**300, 6, 6, 0.1**) yielded the highest BLEU score (**0.279**). This indicates that increasing the model depth can lead to better performance, possibly due to its ability to capture more complex patterns in the data.
- Interestingly, configurations with **2 layers** generally resulted in lower scores, emphasizing the benefit of deeper models in this scenario.

Number of Attention Heads

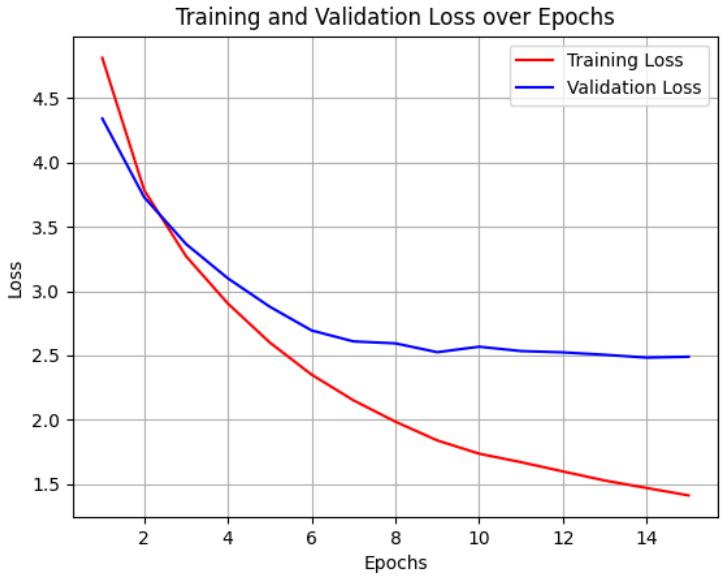
- The attention heads tested were **6**, **8**, and **16**.
- The configuration **300, 6, 6, 0.1** with **6 heads** outperformed all others with more heads, suggesting that more attention heads do not always equate to better performance. For example:
  - The configuration **512, 2, 16, 0.1** with **16 heads** produced a lower score (**0.05**).
- This may imply that using a more focused approach (fewer heads) could enhance the learning process, allowing the model to specialize in certain aspects of the input data.

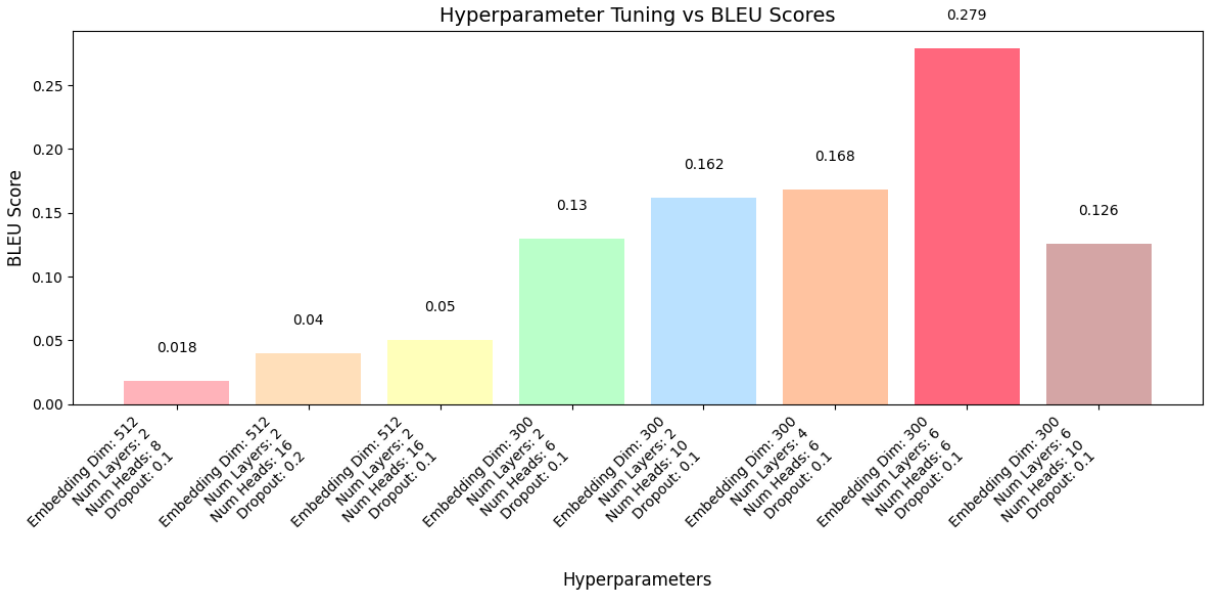
The Bleu Score improves on increasing the number of heads from 3 to 6, but beyond that, on increasing to 10 the performance degraded slightly.

Sentence-wise Bleu scores have been reported in **testbleu.txt**

The source, target sentences and their predictions have been included in **predictions.txt**

Loss Graph for Best Hyperparameters


Training and Validation Loss over Epochs
Training Loss
Validation Loss
Loss
Epochs
4.5
4.0
3.5
3.0
2.5
2.0
1.5
2
4
6
8
10
12
14


Hyperparameter Tuning vs BLEU Scores
BLEU Score
Hyperparameters
0.018
0.04
0.05
0.13
0.162
0.168
0.279
0.126
Embedding Dim: 512 Num Layers: 2 Num Heads: 8 Dropout: 0.1
Embedding Dim: 512 Num Layers: 2 Num Heads: 16 Dropout: 0.2
Embedding Dim: 512 Num Layers: 2 Num Heads: 16 Dropout: 0.1
Embedding Dim: 300 Num Layers: 2 Num Heads: 6 Dropout: 0.1
Embedding Dim: 300 Num Layers: 2 Num Heads: 10 Dropout: 0.1
Embedding Dim: 300 Num Layers: 4 Num Heads: 6 Dropout: 0.1
Embedding Dim: 300 Num Layers: 6 Num Heads: 6 Dropout: 0.1
Embedding Dim: 300 Num Layers: 6 Num Heads: 10 Dropout: 0.1