ASSIGNMENT-2

Question -1

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

Self-attention is a mechanism used in neural networks, particularly in transformer models, to capture relationships between different elements in a sequence. Its main purpose is to help the model focus on the most relevant parts of the input when making predictions, regardless of the position of those elements in the sequence.

Self-attention is particularly useful in tasks like machine translation, where the meaning of a word in one language can depend heavily on the context provided by other words in the sentence. For example, when translating a sentence from French to English, the word "elle" could be translated as "she" or "it," depending on the surrounding words. Self-attention helps the model focus on the relevant parts of the sentence, allowing it to capture the correct meaning of "elle" based on the context.

In self-attention, every element in a sequence is compared with every other element to determine how much "attention" each one should pay to the others. This is done by calculating attention scores, which quantify the importance of one element relative to another. The model then generates a weighted representation of the sequence by taking a weighted sum of the elements based on these scores. By doing this, self-attention enables the model to capture long-range dependencies, meaning that it can understand relationships between words or tokens that are far apart in the sequence, unlike traditional RNNs that struggle with distant dependencies.

Because self-attention treats all tokens equally without regard for their position, it allows for greater flexibility in processing sequences and is highly parallelizable, leading to faster training compared to models that process sequences sequentially. This ability to capture global dependencies efficiently is one of the key reasons why self-attention has become the backbone of transformer models, leading to state-of-the-art performance in various tasks like machine translation, text generation, and more.

Question - 2

Why do transformers use positional encodings in addition to word embeddings?

Transformers use positional encodings in addition to word embeddings because, unlike traditional sequence models such as RNNs, transformers process all elements in a sequence simultaneously rather than sequentially. This parallel processing allows transformers to handle long sequences efficiently, but it also means they lose the inherent notion of word order. Word order is crucial for understanding meaning, as the position of words in a sentence affects how they relate to each other.

Positional encodings solve this problem by adding information about the relative or absolute position of words in a sequence to the word embeddings. These encodings are mathematical vectors that are combined with word embeddings before being fed into the model. By incorporating positional information, transformers are able to

distinguish between, for example, "The cat sat on the mat" and "The mat sat on the cat," despite processing the entire sequence in parallel.

Explain how positional encodings are incorporated into the transformer architecture.

Each word in the input sequence is first converted into a word embedding, which is a fixed-length vector representation capturing the meaning of the word. To include information about the word's position, a positional encoding is added to each word embedding.

Positional encodings are vectors of the same dimension as word embeddings, but they encode positional information. The encoding for each position in the sequence is unique and calculated using sine and cosine functions at varying frequencies. The positional encoding for position pos and dimension i is defined as:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Here, pos is the position in the sequence, i is the dimension index, and d is the embedding dimension.

After computing the positional encodings, they are added element-wise to the word embeddings. This results in a combined representation that incorporates both the semantic meaning of the word (from the embedding) and its position in the sequence (from the positional encoding). This combination is the input to the transformer model.

The combined embeddings (word embedding + positional encoding) are fed into the transformer's encoder and decoder layers. The self-attention mechanism in the transformer can now use these combined vectors to understand both the content of each word and its relative position in the sequence.

Transformers also incorporate residual connections, where the output of each layer is added back to its input. These residual connections not only help in addressing issues like vanishing gradients but also ensure that positional information is retained throughout the deeper layers of the network.

Briefly describe recent advances in various types of positional encodings used for transformers and how they differ from traditional sinusoidal positional encodings.

Recent advances in positional encodings for transformers have introduced various methods that go beyond traditional sinusoidal encodings to better capture positional information in different tasks and contexts.

1. **Learnable Positional Embeddings:** Unlike sinusoidal encodings, where the position of each token is determined using fixed sine and cosine functions, learnable positional embeddings treat positional encodings as parameters that the model can learn during training. This allows the model to adapt positional representations based on the specific task, providing more flexibility but potentially requiring more data to generalize effectively.
2. **Relative Positional Encodings:** Traditional sinusoidal encodings encode absolute positions, which may not always be optimal for capturing local dependencies. Relative positional encodings focus on the relative

distance between tokens, which is often more relevant for tasks like machine translation or text generation, where the relative ordering of words can carry more meaning than their absolute positions.

3. **Rotary Positional Embeddings (RoPE):** RoPE extends the idea of relative positional encoding by using a rotation matrix to transform the token embeddings. This method preserves the relationship between tokens based on their relative distance, and it has been shown to work well in language models, particularly in improving long-range dependency capture.
4. **Logarithmic Positional Encodings:** These encodings scale positional information logarithmically, prioritizing nearby tokens while gradually diminishing the influence of distant tokens. This approach is useful for tasks that focus more on local context, such as sentence-level analysis, while still retaining some global positional awareness.
5. **Coordinate-Based Positional Encodings:** In tasks involving images, 2D coordinate-based positional encodings have been introduced to better capture the spatial relationships of pixels or patches, as opposed to traditional 1D encodings used in language tasks. These encodings map both row and column positions of image elements, providing a more natural representation for visual data.

These advanced methods improve the model's ability to capture complex positional relationships, offering more task-specific flexibility compared to the rigid, fixed nature of sinusoidal encodings.

Question - 3

Hyperparameter Tuning

The performance of the model is tested by varying the following hyperparameters:

1. Dimension of word embeddings: [256, 512]
2. No.of attention heads: [4, 8]
3. No.of Encoder layers: [1, 2]
4. No.of Decoder layers: [1, 2]
5. Dropout: [0.1, 0.2]

While keeping the following parameters constant:

1. Feedforward dimension: 2048
2. learning rate: 0.01
3. Batch size: 64
4. No.of epochs: 5

d_model	n_heads	n_en_layers	n_de_layers	dropout	bleu score
512	8	1	2	0.1	0.12546

d_model	n_heads	n_en_layers	n_de_layers	dropout	bleu score
512	4	1	2	0.1	0.12472
512	4	1	2	0.2	0.12304
512	8	1	2	0.2	0.11938
512	8	1	1	0.1	0.11756
256	4	1	2	0.1	0.11081
256	8	1	2	0.1	0.11046
512	4	1	1	0.1	0.11344
256	4	1	2	0.2	0.10810
256	8	1	2	0.2	0.10788
512	8	2	2	0.1	0.10911
256	4	2	2	0.1	0.10410
512	8	1	1	0.2	0.10468
512	4	2	2	0.2	0.10282
512	4	1	1	0.2	0.10233
256	8	1	1	0.2	0.09859
256	4	1	1	0.1	0.09848
256	8	2	2	0.1	0.09771
512	8	2	2	0.2	0.09888
256	8	1	1	0.1	0.09699
256	4	1	1	0.2	0.09602
256	4	2	2	0.2	0.08203
256	8	2	2	0.2	0.07752
256	4	2	1	0.1	0.05483
256	8	2	1	0.2	0.05227
512	8	2	1	0.2	0.03159
256	8	2	1	0.1	0.02444
512	4	2	1	0.2	0.02479
256	4	2	1	0.2	0.02073
512	8	2	1	0.1	0.02213
512	4	2	1	0.1	0.01865
512	4	2	2	0.1	0.01587

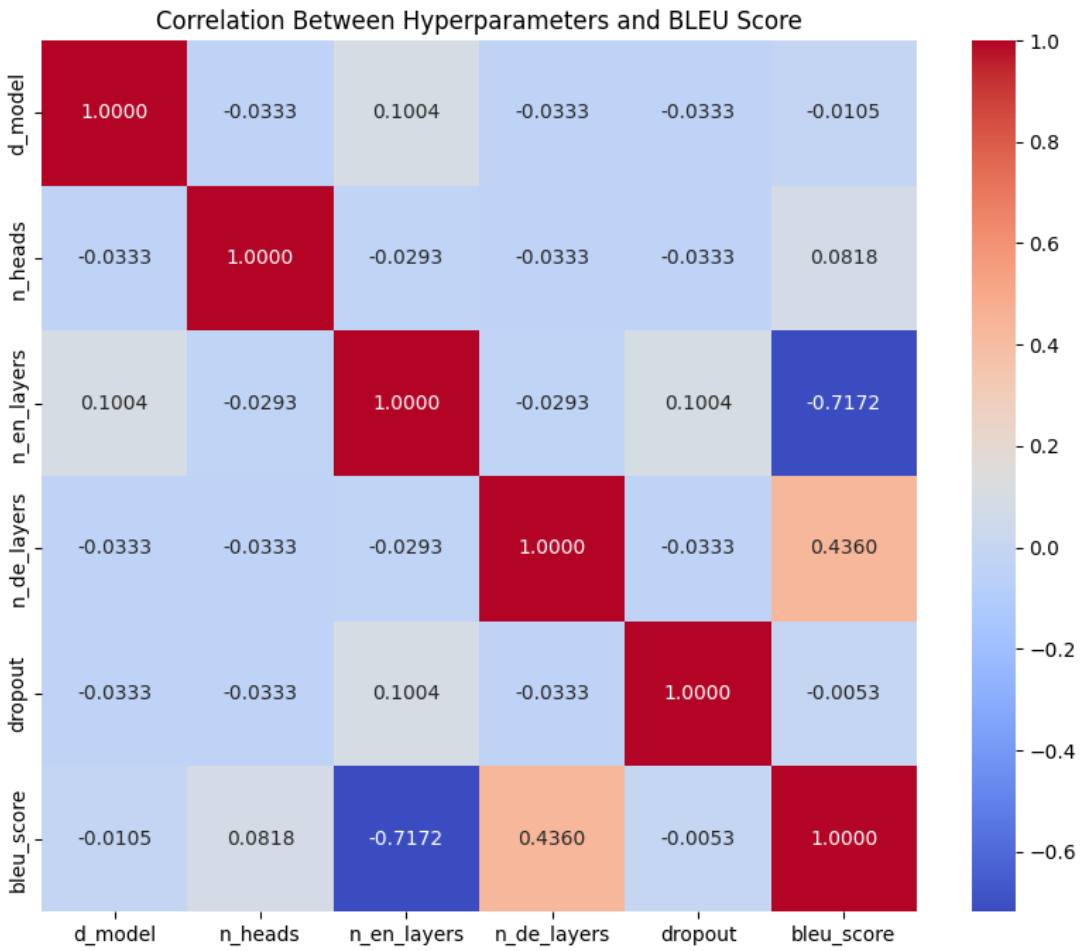
From the above experiments, the best hyperparameters are:

1. Dimension of word embeddings: 512

2. No.of attention heads: 8
3. No.of Encoder layers: 1
4. No.of Decoder layers: 2
5. Dropout: 0.1

The best-performing model and its predictions are saved. (predictions are present in testbleu.txt)

Analysis



- No.of encoding layers and decoding layers are showing a significant impact on the model's performance.
- There is a significant **negative correlation** (-0.7172) between the number of encoder layers and the BLEU score. This indicates that increasing the number of encoder layers might decrease the performance.

- There's a positive correlation (0.4360) between the number of decoder layers and the BLEU score. This suggests that increasing decoder layers tends to improve performance to some extent.

Explanation of Performance Differences:

- **Model Dimension (d_model)**: A larger d_model allows the model to learn richer representations, improving translation accuracy. Smaller models may struggle with complex translations, leading to lower BLEU scores.
- **Attention Heads (n_heads)**: More attention heads improve the model's ability to capture diverse relationships within the input sequence, though the difference between 4 and 8 heads is not drastic.
- **Encoder Layers (n_en_layers)**: While deeper models can theoretically capture more abstract features, the small number of layers (1 or 2) suggests the model reaches its limit quickly. Deeper architectures may suffer from overfitting or training instability without proper adjustments.
- **Dropout**: Higher dropout rates (0.2) may overly regularize the model, preventing it from effectively learning patterns in the data, which explains the drop in BLEU scores compared to lower dropout values (0.1).

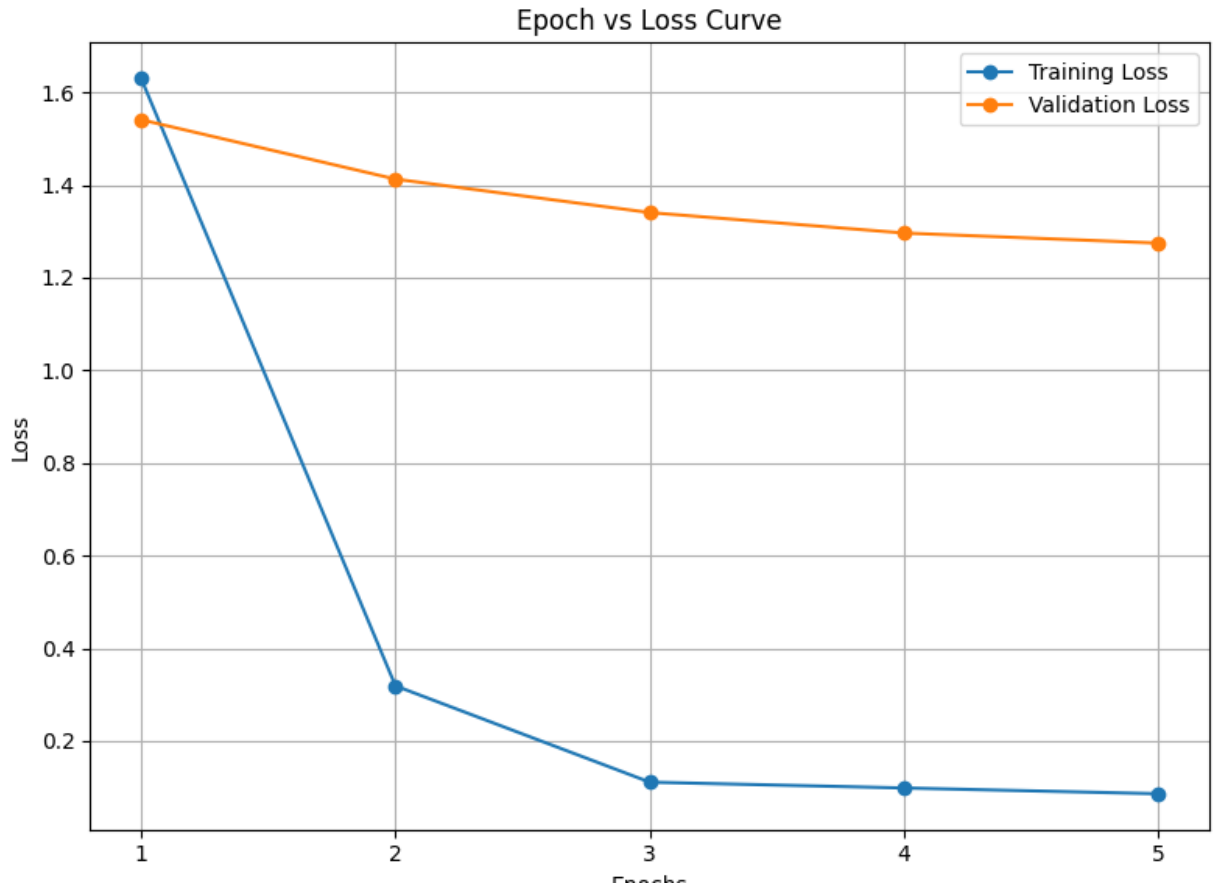
Training

Now, the model is trained on the above set of best hyperparameters

Training:

```
train size: 30000
val size: 887
test size: 1305
training started
Epoch 1/5 - Step 100 - Train Loss: 1.4749
Epoch 1/5 - Step 200 - Train Loss: 1.8399
Epoch 1/5 - Step 300 - Train Loss: 1.3185
Epoch 1/5 - Step 400 - Train Loss: 1.3494
Epoch 1/5 - Train Loss: 1.6299 - Val Loss: 1.5408
Epoch 2/5 - Step 100 - Train Loss: 0.2188
Epoch 2/5 - Step 200 - Train Loss: 0.1801
Epoch 2/5 - Step 300 - Train Loss: 0.1255
Epoch 2/5 - Step 400 - Train Loss: 0.2943
Epoch 2/5 - Train Loss: 0.3188 - Val Loss: 1.4126
Epoch 3/5 - Step 100 - Train Loss: 0.1227
Epoch 3/5 - Step 200 - Train Loss: 0.0926
Epoch 3/5 - Step 300 - Train Loss: 0.1097
Epoch 3/5 - Step 400 - Train Loss: 0.1154
Epoch 3/5 - Train Loss: 0.1109 - Val Loss: 1.3404
Epoch 4/5 - Step 100 - Train Loss: 0.1034
Epoch 4/5 - Step 200 - Train Loss: 0.1097
Epoch 4/5 - Step 300 - Train Loss: 0.0988
Epoch 4/5 - Step 400 - Train Loss: 0.0934
Epoch 4/5 - Train Loss: 0.0984 - Val Loss: 1.2962
Epoch 5/5 - Step 100 - Train Loss: 0.0894
Epoch 5/5 - Step 200 - Train Loss: 0.0878
Epoch 5/5 - Step 300 - Train Loss: 0.0878
Epoch 5/5 - Step 400 - Train Loss: 0.0793
Epoch 5/5 - Train Loss: 0.0859 - Val Loss: 1.2745
Test Loss: 1.2312
```

Loss Curve:



Testing:

```
spacy loaded
tokenization done
vocab created
preprocessing done
test size: 1305
Average BLEU score: 0.11292271575281447
(env) pavani@bhashini-ilmt:/home/pavani/sbcavva$ nano train.py
```