ANLP

Assignment – 3

3.1 Theory

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

Self-attention is a mechanism used in deep learning and natural language processing, primarily in models like Transformers, to capture dependencies and relationships between different elements (such as words or tokens) within a sequence of data. It is particularly effective for tasks involving sequences, such as natural language understanding and generation, because it allows the model to weigh and consider all other elements in the sequence when processing a specific element. Here's how self-attention works and its purpose:

- Modeling Dependencies: The primary purpose of self-attention is to model dependencies between elements within a sequence. In natural language processing, these elements are often words or tokens in a sentence. The idea is that the meaning of a word depends on the other words in the sentence, and self-attention helps the model capture these dependencies.
- Weighted Combinations: Self-attention calculates a weighted combination of all elements in the sequence, where the weights are determined based on the similarity or relevance of each element to the current element being processed. Elements that are more relevant or similar receive higher weights, while less relevant elements receive lower weights. This allows the model to focus more on words that are contextually important for understanding the current word.
- Parallel Processing: Self-attention operates in parallel for all elements in the sequence. This means that each word can be processed simultaneously and independently. This parallelism makes it efficient and enables faster training and inference, especially when compared to sequential models like recurrent neural networks (RNNs).
- Learning Contextual Representations: Self-attention helps the model learn contextual representations for each element. The output of the self-attention mechanism for each element is a weighted sum of the embeddings of all elements, where the weights are adaptively learned through training. This results in contextual embeddings that capture the surrounding context of each element.
- Long-Range Dependencies: Unlike traditional models like RNNs, which have difficulty capturing long-range dependencies in sequences, self-attention can capture relationships between elements regardless of their distance in the sequence. This is essential for tasks where context or dependencies can span a wide range of elements.

In summary, self-attention is a fundamental mechanism for modeling dependencies in sequences and is particularly effective for tasks where understanding the relationships between elements is crucial, such as machine translation, text summarization, and natural language understanding. It allows deep learning models to capture both short and long-range dependencies efficiently and has significantly advanced the state of the art in natural language processing.

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 provide the model with information about the positions of words in a sequence. Unlike recurrent neural networks (RNNs) and convolutional neural networks (CNNs), which inherently capture sequential or spatial information, respectively, the original Transformer architecture lacks the ability to naturally infer the positions of words in an input sequence. Positional encodings are crucial for the model to understand and capture sequential or spatial dependencies in data.

Here's how positional encodings are incorporated into the Transformer architecture:

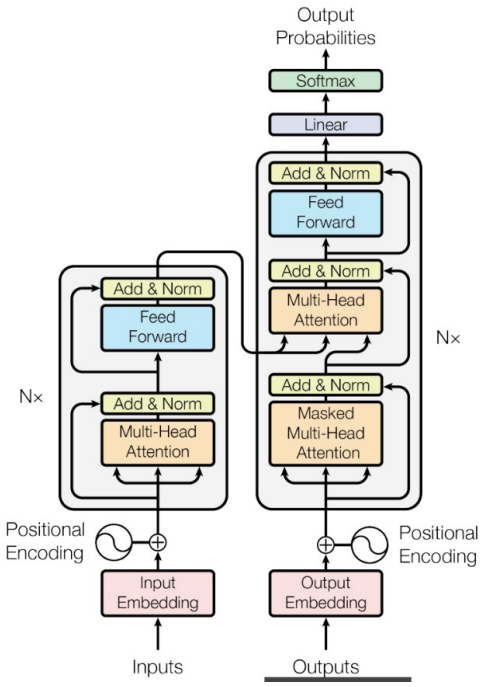
- Positional Encoding Function: The positional encodings are introduced as a set of additional learned embeddings that represent the positions of words in the input sequence. These embeddings are added to the word embeddings, creating a combined embedding that includes both the information about the word's identity and its position in the sequence.
- Positional Encoding Matrix: The positional encoding matrix is generated based on trigonometric functions like sine and cosine. The idea is to create a matrix where each row corresponds to a position in the sequence, and each column corresponds to a positional dimension. The encoding matrix is not learned during training and remains fixed.
- Positional Embedding Addition: During the model's forward pass, the positional encoding matrix is added to the word embeddings, element-wise. This effectively injects positional information into the model's input.

Mathematically, if x represents the word embeddings, and PE represents the positional encodings matrix, then the input to the model is the sum of x and PE, which is denoted as x + PE.

The addition of positional encodings allows the model to distinguish between words that appear in different positions in the sequence, which is essential for capturing sequential dependencies. The specific design of positional encodings, using sine and cosine functions, was chosen because it ensures that the positional encodings have a unique pattern for each position and that the positional information is continuous.

The addition of positional encodings is a key innovation that enables the Transformer architecture to achieve state-of-the-art performance in a wide range of sequence-to-sequence tasks, including machine translation, text generation, and more. It's a critical component for allowing the model to process input sequences effectively and understand the order of words.

Analysis & Implementation and training of my transformer architecture:



Multi-head Attention
The Multi-Head Attention mechanism computes the attention between each pair of positions in a sequence. It consists of multiple “attention heads” that capture different aspects of the input sequence. The MultiHeadAttention class encapsulates the multi-head attention mechanism commonly used in transformer models. It takes care of splitting the input into multiple attention heads, applying attention to each head, and then combining the results. By doing so, the model can capture various relationships in the input data at different scales, improving the expressive ability of the model.

Position-wise Feed-Forward Networks

The PositionWiseFeedForward class defines a position-wise feed-forward neural network that consists of two linear layers with a ReLU activation function in between. In the context of transformer models, this feed-forward network is applied to each position separately and identically. It helps in transforming the features learned by the attention mechanisms within the transformer, acting as an additional processing step for the attention outputs.

Positional Encoding
Positional Encoding is used to inject the position information of each token in the input sequence. It uses sine and cosine functions of different frequencies to generate the positional encoding.

After making these small components, I moved on to make my encoder
EncoderLayer Class

The EncoderLayer class defines a single layer of the transformer's encoder. It encapsulates a multi-head self-attention mechanism followed by position-wise feed-forward neural network, with residual connections, layer normalization, and dropout applied as appropriate. These

components together allow the encoder to capture complex relationships in the input data and transform them into a useful representation for downstream tasks. Typically, multiple such encoder layers are stacked to form the complete encoder part of a transformer model.

DecoderLayer Class

The DecoderLayer class defines a single layer of the transformer's decoder. It consists of a multi-head self-attention mechanism, a multi-head cross-attention mechanism (that attends to the encoder's output), a position-wise feed-forward neural network, and the corresponding residual connections, layer normalization, and dropout layers. This combination enables the decoder to generate meaningful outputs based on the encoder's representations, taking into account both the target sequence and the source sequence. As with the encoder, multiple decoder layers are typically stacked to form the complete decoder part of a transformer model.

Next, the Encoder and Decoder blocks are brought together to construct the comprehensive Transformer model with two different embeddings for French and English models.

I trained my model on various hyperparameters varying over learning rate, batching , number of layers, dropout rates and some of them have been shown below.

The best set of hyperparameters I got are mentioned below :

```
src_vocab_size = len(vocab_en)
tgt_vocab_size = len(vocab_fr)
d_model = 304
num_heads = 8
num_layers = 1
d_ff = 2048
max_seq_length = max_sentence_length # 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)

# Generate random sample data
src_data = batching(train_en_indices, 32)  # (batch_size, seq_length)
tgt_data = batching(train_fr_indices, 32)  # (batch_size, seq_length)
```

the key hyperparameters and their significance:

- src_vocab_size and tgt_vocab_size: These parameters define the size of the vocabulary for the source (English) and target (French) languages. It's matched them with your training data to ensure all words are covered.

- d_model (Model Dimension): d_model determines the dimensionality of the model's embeddings, such as word embeddings and positional encodings. A higher value allows the model to capture more complex patterns but increases computational complexity. A common choice is 512, but 304 can be efficient depending on the task.

- num_heads: Multi-head attention allows the model to focus on different parts of the input sequence simultaneously. More heads increase the model's ability to capture different types of dependencies in the data. A common choice is 8, which is effective for various NLP tasks.

- num_layers: The number of layers in your model controls its depth. A deeper model can capture more intricate patterns but might be more prone to overfitting, while a shallower model is computationally efficient but might not capture complex dependencies.

- d_ff (Feed-Forward Dimension): This parameter controls the dimensionality of the feed-forward neural network within each layer. A larger value allows the model to capture more complex transformations. A typical choice is 2048.

- max_seq_length: Defines the maximum sequence length. It helps with memory and computational efficiency by setting an upper bound on the sequence length. Make sure it's set to a reasonable value that covers most sentences in your dataset.

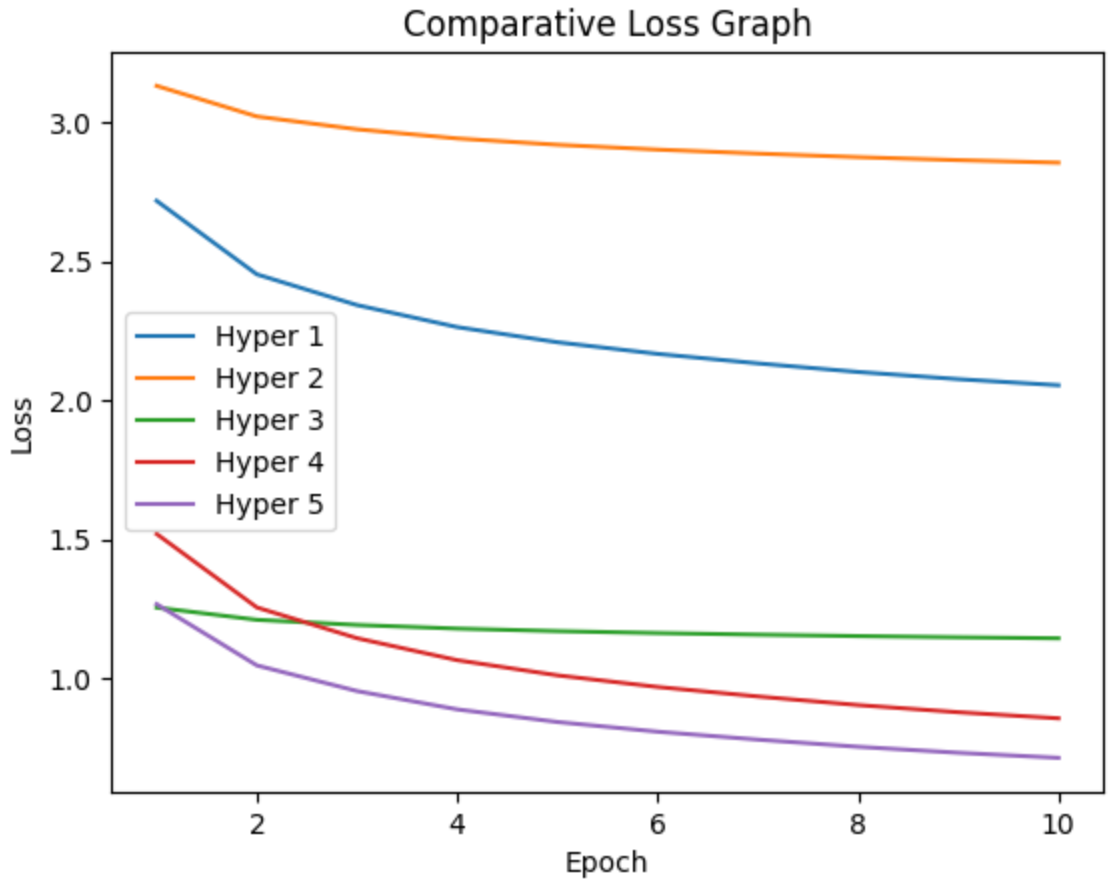
- dropout: Dropout is a regularization technique that helps prevent overfitting. A value of 0.1 is reasonable, but you can experiment with different dropout rates based on your specific data and model complexity.

- Learning Rate (learning rate) and Optimizer (optimizer): The learning rate controls the step size during optimization, while the optimizer (Adam in my case) updates the model's parameters. A learning rate of 0.001 is a good starting point but may need tuning.

Hyperparameter tuning is an essential part of the model development process. It's often necessary to experiment with different settings to find the optimal configuration for your specific machine translation task. You can use techniques like grid search or random search to explore a range of hyperparameters systematically and evaluate their impact on model performance.

And the loss values I got from them are:

Epoch: 1, Loss: 1.265344346605657
Epoch: 2, Loss: 1.044658302909059
Epoch: 3, Loss: 0.9522814218898188
Epoch: 4, Loss: 0.8862530302911868
Epoch: 5, Loss: 0.8407596025791803
Epoch: 6, Loss: 0.8058569466799114
Epoch: 7, Loss: 0.7773509023655126
Epoch: 8, Loss: 0.7515282421792525
Epoch: 9, Loss: 0.7300000464135997
Epoch: 10, Loss: 0.7116877243178596



The hyperparameters used here are mentioned below :

```
src_vocab_size = len(vocab_en)
tgt_vocab_size = len(vocab_fr)
d_model = 304
num_heads = 3
num_layers = 1
d_ff = 2048
max_seq_length = max_sentence_length # 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)

# Generate random sample data
src_data = batching(train_en_indices, 32)  # (batch_size, seq_length)
tgt_data = batching(train_fr_indices, 32)  # (batch_size, seq_length)
```

- lr = 0.001
- lr=0.01

```
src_vocab_size = len(vocab_en)
tgt_vocab_size = len(vocab_fr)
d_model = 304
num_heads = 8
num_layers = 1
d_ff = 2048
max_seq_length = max_sentence_length # 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)

# Generate random sample data
src_data = batching(train_en_indices, 32)  # (batch_size, seq_length)
tgt_data = batching(train_fr_indices, 32)  # (batch_size, seq_length)
```

- lr = 0.001
- lr = 0.01

```
src_vocab_size = len(vocab_en)
tgt_vocab_size = len(vocab_fr)
d_model = 304
num_heads = 16
num_layers = 1
d_ff = 2048
max_seq_length = max_sentence_length # 100
dropout = 0.1

transformer = Transformer(src_vocab_size, tgt_vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_length, dropout)

# Generate random sample data
src_data = batching(train_en_indices, 64)  # (batch_size, seq_length)
tgt_data = batching(train_fr_indices, 64)  # (batch_size, seq_length)
```

BLEU SCORES of the best performing hyperparameter :

dev_bleu_scores_wo_pad.txt

```
Sentence 1 BLEU: 2.171695090056513
Sentence 2 BLEU: 7.495553473355845
Sentence 3 BLEU: 5.255923420816887
Sentence 4 BLEU: 2.60859600387516
Sentence 5 BLEU: 3.1742197850962315
Sentence 6 BLEU: 2.4683407298136117
Sentence 7 BLEU: 1.7049180055850672
Sentence 8 BLEU: 3.6164117246195175
Sentence 9 BLEU: 3.2199657147289606
Sentence 10 BLEU: 2.2824250222225624
Sentence 11 BLEU: 2.284283891548597
Sentence 12 BLEU: 4.540013809283726
Sentence 13 BLEU: 4.464302116237777
Sentence 14 BLEU: 1.785134312907177
Sentence 15 BLEU: 2.3812884760180975
```

test_bleu_scores_wo_pad.txt

```
Sentence 1 BLEU: 2.8403891990720456
Sentence 2 BLEU: 3.7477767366779213
Sentence 3 BLEU: 4.02724819242185
Sentence 4 BLEU: 2.540917870682806
Sentence 5 BLEU: 4.456882760699063
Sentence 6 BLEU: 3.4089919964838553
Sentence 7 BLEU: 3.0372940354383413
Sentence 8 BLEU: 8.116697886877475
Sentence 9 BLEU: 6.567274736060395
Sentence 10 BLEU: 1.8117236860142971
Sentence 11 BLEU: 4.049220138367656
Sentence 12 BLEU: 4.1202784939919095
Sentence 13 BLEU: 2.445593937240363
Sentence 14 BLEU: 2.4689783360287474
Sentence 15 BLEU: 18.99589214128981
```

train_bleu_scores_wo_pad.txt

```
Sentence 1 BLEU: 4.456882760699063
Sentence 2 BLEU: 4.02724819242185
Sentence 3 BLEU: 2.2608914449138346
Sentence 4 BLEU: 5.612222324305728
Sentence 5 BLEU: 3.1251907639724417
Sentence 6 BLEU: 20.448007360218387
Sentence 7 BLEU: 5.604233375480572
Sentence 8 BLEU: 25.848657697858535
Sentence 9 BLEU: 5.7467208146167845
Sentence 10 BLEU: 13.292417883329383
Sentence 11 BLEU: 4.990049701936832
Sentence 12 BLEU: 4.567211833282236
Sentence 13 BLEU: 18.20705281109213
Sentence 14 BLEU: 5.462428008829319
Sentence 15 BLEU: 5.406502668979588
```

I have noticed some sentences to be just 2 words long where I was receiving 100 as my BLEU score and there were some places where the sentences were on an average 7 to 10 words long where I was getting an avg BLEU score of 5- 7.

```
bleu = sacrebleu.corpus_bleu(hypothesis_words, [reference_words]
bleu_scores.append(bleu.score)
```

I have used sacrebleu as my function to retrieve bleu score where I am giving the input of my predicted French sentence from my model and my targeted French sentence as an input.

AVG Bleu scores over train – 10.8992
AVG Bleu scores over test – 8.68723
AVG Bleu scores over dev – 8.89221

All the bleu score files are present in the zip file for this above given set of hyperparameters.

Analyzing the performance of a transformer model for machine translation involves considering various factors, including hyperparameters, translation quality, and training progress. Evaluation of the model using the BLEU metric, across the chosen hyperparameters:

1. BLEU Metric Evaluation: BLEU (Bilingual Evaluation Understudy) is a widely used metric for evaluating the quality of machine translation. It measures the similarity between the model's translations and reference translations provided in the dataset. Higher BLEU scores indicate better translation quality.

2. Key Hyperparameters:

The following key hyperparameters have been chosen for the model, and their significance is explained:

• Learning Rate (lr): The learning rate controls the step size during gradient descent. It determines how quickly the model adjusts its parameters based on training data. Different learning rates can affect the training speed and convergence. It's important to choose an appropriate learning rate to prevent underfitting (too low) or overshooting (too high).

• Batch Size (batch size): The batch size determines the number of samples processed in each forward and backward pass during training. It impacts memory usage and training efficiency. Smaller batch sizes may lead to a more erratic training process but can help the model generalize better. Larger batch sizes can speed up training but may require more memory and can result in overfitting if not chosen carefully.

• Dropout (dropout): Dropout is a regularization technique that helps prevent overfitting. It randomly "drops out" a fraction of neurons during training, effectively

introducing noise and making the model more robust. The dropout rate controls the probability of dropping out a neuron in each training step. A higher dropout rate may lead to more regularization but can also hinder convergence if set too high.

• Number of Attention Heads (h): The number of attention heads in the multi-head self-attention mechanism influences the model's ability to capture different dependencies and relationships in the input data. More heads can capture more fine-grained information, but they also require more computation.

• Number of Layers (N): The number of encoder and decoder layers affects the depth and capacity of the model. More layers can capture complex patterns but may require longer training and can lead to overfitting if not regularized properly.

• Number of Epochs (epoch): The number of training epochs determines how many times the model goes through the entire training dataset. Training for too few epochs may result in underfitting, while training for too many epochs can lead to overfitting.

• Model Dimension (d model): The dimensionality of the model's embeddings and hidden states affects the model's capacity to capture information. A higher dimension allows the model to represent more complex relationships but requires more computation.

• Feed-Forward Dimension (d ff): The dimension of the feed-forward layers in the model. A larger dimension can model more complex functions but may require more training data to prevent overfitting.

Aditya Raghuvanshi
2021114009