Question 1

Self-attention is a crucial mechanism in modern deep learning architectures, especially in models designed for processing sequential data, such as the Transformer. Its primary purpose is to capture dependencies between elements in a sequence, regardless of their distance from one another. Let's explore its purpose and functioning in detail.

Purpose of Self-Attention

1. **Capturing Long-Range Dependencies**: Traditional sequential models like RNNs and LSTMs struggle with long-range dependencies due to their sequential nature and issues like vanishing gradients. Self-attention allows the model to directly relate any two tokens in a sequence, enabling it to learn relationships over long distances more effectively.
2. **Parallelization**: Unlike RNNs, which process inputs sequentially, self-attention can compute all attention scores simultaneously. This leads to significant improvements in training speed, making it possible to leverage modern hardware efficiently.
3. **Dynamic Contextualization**: Self-attention provides a mechanism for dynamically weighting the importance of different elements in the sequence. Each element can attend to all other elements, allowing the model to adjust its focus based on the context, leading to more nuanced representations.
4. **Handling Variable Input Lengths**: Self-attention can easily handle input sequences of varying lengths without requiring changes to the model architecture. This flexibility is especially beneficial in natural language processing (NLP) tasks.

How Self-Attention Works

Self-attention involves three main components derived from the input sequence: **queries (Q)**, **keys (K)**, and **values (V)**. Here's how it facilitates capturing dependencies:

1. **Linear Transformations**: For an input sequence of token embeddings, each token embedding is linearly transformed into a query, key, and value vector:

   $$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

   where X is the matrix of input embeddings and W^Q, W^K, and W^V are learned weight matrices.
2. **Attention Scores Calculation**: The attention scores between tokens are computed using the dot product of the query vector of one token with the key vectors of all tokens:

   $$\text{Attention Score}(Q_i, K_j) = Q_i \cdot K_j$$

   This results in a score matrix that indicates how much focus each token should have on every other token in the sequence.
3. **Softmax Normalization**: To convert the attention scores into a probability distribution, a softmax function is applied to the scores for each query:

   $$\alpha_{ij} = \frac{e^{\text{Attention Score}(Q_i, K_j)}}{\sum_k e^{\text{Attention Score}(Q_i, K_k)}}$$

Here, α_{ij} represents the attention weight of token j with respect to token i.

4. **Weighted Sum of Values**: Finally, each token's representation is updated by taking a weighted sum of the value vectors, where the weights are the attention scores:

$$\text{Output}_i = \sum_j \alpha_{ij} V_j$$

This output retains the information from the entire sequence while emphasizing the most relevant parts based on the learned attention weights.

Multi-Head Attention

To capture different types of relationships and dependencies in a more nuanced way, self-attention is often implemented in a multi-head fashion. This involves:

1. **Multiple Attention Heads**: Instead of a single set of Q, K, and V transformations, multiple sets (or heads) are used. Each head learns to focus on different aspects of the data.
2. **Concatenation and Final Transformation**: The outputs of all heads are concatenated and passed through a final linear transformation, allowing the model to combine various perspectives on the input sequence.

Conclusion

Self-attention fundamentally transforms how sequences are processed by allowing for direct relationships between all elements, capturing dependencies efficiently and flexibly. Its ability to learn contextual relationships dynamically, while also facilitating parallel processing, makes it an essential component in state-of-the-art models for tasks ranging from machine translation to text summarization and beyond. The development of the Transformer architecture, which relies heavily on self-attention, has propelled significant advancements in the field of NLP and deep learning as a whole.

Question 2

Transformers utilize positional encodings alongside word embeddings because, unlike recurrent neural networks (RNNs), they do not inherently process sequences in a temporal order. Positional encodings provide the model with information about the position of each token within the input sequence, allowing it to understand the order and relative positioning of words or elements.

Why Positional Encodings Are Necessary

1. **Lack of Sequential Order**: Transformers process input tokens in parallel, which means they do not maintain a natural sequence order as RNNs do. Without positional encodings, the model would treat each token independently, losing critical information about the structure of the sequence.
2. **Understanding Context**: The meaning of a word often depends on its position in a sentence. For instance, "The cat sat on the mat" has a different meaning than "The mat sat on the cat." Positional encodings help the model differentiate between such contextual nuances.

Incorporation of Positional Encodings

In the transformer architecture, positional encodings are added directly to the word embeddings before they are fed into the encoder or decoder layers. The process typically involves:

1. **Generating Positional Encodings**: For each position pos in the input sequence and each dimension i of the embedding, sinusoidal positional encodings are computed using the following formulas:

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

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

   where d_{model} is the dimensionality of the embeddings. This results in a unique encoding for each position that varies smoothly across dimensions.
2. **Adding Positional Encodings to Embeddings**: The positional encoding vector for each token is added to its corresponding word embedding:

$$\text{Input} = \text{Word Embedding} + \text{Positional Encoding}$$

   This combined representation is then passed through the transformer layers.

Recent Advances in Positional Encodings

While traditional sinusoidal positional encodings are effective, several recent approaches have been proposed to enhance or replace them:

1. **Learned Positional Embeddings**: Instead of using fixed sinusoidal functions, some models use learned positional embeddings, where each position has a corresponding trainable vector. This allows the model to adapt positional information based on the training data.
2. **Relative Positional Encodings**: These encodings focus on the relative distance between tokens rather than absolute positions. This approach can improve performance in tasks like translation or language modeling, where understanding the relationship between words is crucial. They allow the model to generalize better across different sequence lengths.
3. **Rotary Positional Embeddings (RoPE)**: This method encodes positional information using complex numbers, enabling the model to capture relative positions more effectively. RoPE maintains the properties of the original embeddings while enhancing the ability to generalize to unseen sequences.
4. **Learned Attention Mechanisms**: Some newer architectures incorporate attention mechanisms that can adjust based on positional information dynamically, making the model more flexible in how it attends to different parts of the sequence.

Differences from Traditional Sinusoidal Positional Encodings

- **Static vs. Dynamic**: Sinusoidal encodings are static and predetermined, while learned positional embeddings can adapt to the specific patterns in the training data.

- **Absolute vs. Relative**: Traditional sinusoidal encodings provide absolute positional information, while methods like relative positional encodings focus on the relationships between tokens, offering more nuanced understanding.
- **Complexity and Generalization**: Approaches like RoPE leverage complex representations that can better capture relationships in various contexts, potentially improving performance on specific tasks compared to sinusoidal encodings.

In summary, positional encodings are essential for enabling transformers to understand the order of input sequences, and recent advances have provided more flexible and effective methods for incorporating positional information into model architectures.

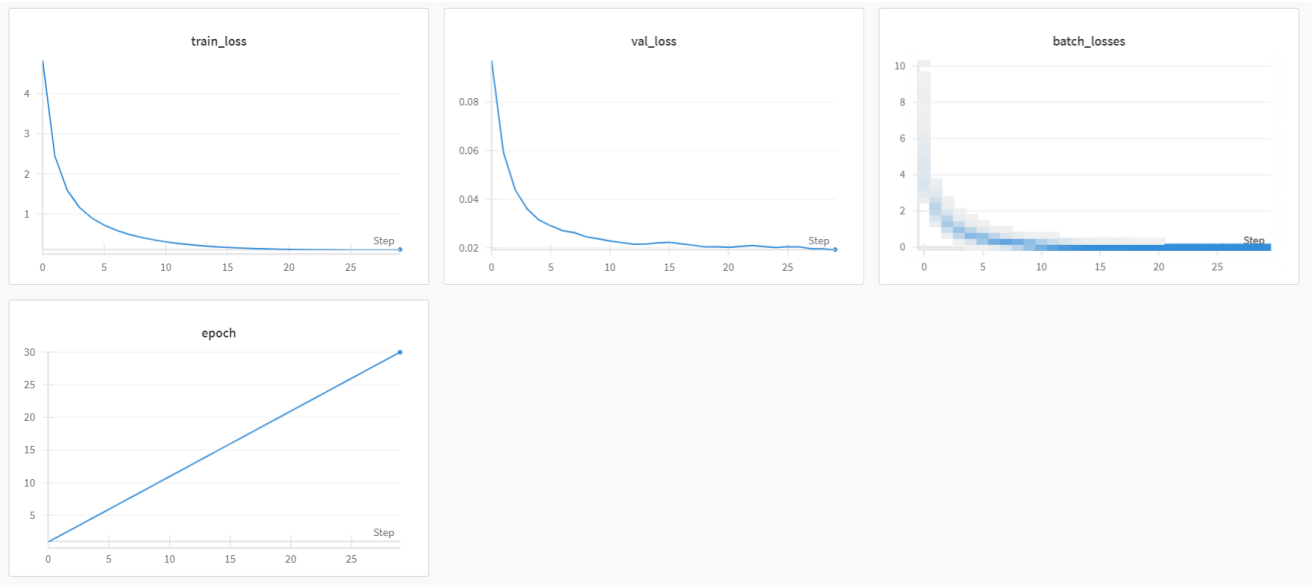
Hyperparameters

The best hyperparameters are as follows:

512 '--d_model' = 'Dimensionality of the model'
8 '--num_heads' = 'Number of attention heads'
6 '--num_layers' = 'Number of layers in the model'
2048 '--d_ff' = 'Dimensionality of the feed-forward layer'
100 '--max_seq_length' = 'Maximum sequence length for input'
0.1 '--dropout' = 'Dropout rate'
32 '--batch_size' = 'Batch size for training'
10 '--num_epochs' = 'Number of epochs for training'
0.0001 '--learning_rate' = 'Learning rate for the optimizer'

Loss graphs and Eval metrics

All the losses (along with model params) have been logged to wandb. The best model's graph is shown below



BLEU score was chosen as the eval metric.
BLEU scores have been written to testbleu.txt, The mean BLEU score is : 0.34401817512232635
(actual translated sentences are also reported in the test_translated_full.txt file)

Analysis

We see that we get a reasonable translation (and bleu score) with multiple layers compared to a single layer...we also see an improvement with having multiple heads, however as number of layers keep increasing, we dont get increasing results, the same is true for the number of heads. We also see that the model performs equally (if not better) even without initialising the embeddings to a pre-trained one