Report

Question 1:
What is the purpose of self-attention, and how does it facilitate capturing dependencies in sequences?
Answer:Self-attention is a mechanism designed to capture dependencies between words (or tokens) in a sequence, regardless of their positions. Its main purposes are:

- **Capturing Context and Relationships:** Understanding the relationship between words in the entire sequence.
- **Efficiently Learning Long-Term Dependencies:** Modeling dependencies without being constrained by sequence order, which allows for more flexible context understanding.

How Self-Attention Works

Self-attention works by computing a weighted sum of all tokens in a sequence, where the weights reflect how much focus each word should give to others. This is done through the following steps:

1. **Input Representation:**
   Each word in the sequence is transformed into a vector (embedding).
2. **Creating Queries, Keys, and Values (Q, K, V):**
   For self-attention, three vectors are derived for each token:
   - **Query (Q):** Represents the word for which attention is being calculated.
   - **Key (K):** Represents the words being attended to.
   - **Value (V):** Contains the information of each word.
3. These vectors are calculated as follows:
   - Q = X * W_Q
   - K = X * W_K
   - V = X * W_V
4. where:
   - X is the input embedding matrix.
   - W_Q, W_K, and W_V are learned weight matrices.
5. **Calculating Attention Scores:**
   The attention score for each word pair is calculated using the dot product between the query and key vectors:
   - score = (Q * K^T) / sqrt(d_k)
6. where:
   - d_k is the dimension of the key vectors.

7. **Applying Softmax to Obtain Weights:**
   The scores are passed through a softmax function to obtain normalized attention weights:
   - Attention Weights = softmax((Q * K^T) / sqrt(d_k))
8. These weights determine how much each word attends to every other word.
9. **Computing Weighted Sum of Values:**
   A weighted sum of value vectors is computed using the attention weights:
   - Output = Attention Weights * V
10. This results in a context vector for each word, incorporating information from all other words in the sequence.

Capturing Dependencies

By allowing each word to attend to every other word, self-attention captures both local and global dependencies within the sequence, regardless of the positional distance between tokens.

Multi-Head Self-Attention

Transformers use **multi-head attention** to capture different aspects of relationships between words. Multiple attention heads are computed in parallel, and their outputs are concatenated and linearly transformed to capture richer information.

Advantages of Self-Attention

1. **Parallel Computation:** Enables parallel processing of sequences, making it faster than sequential models like RNNs.
2. **Flexible Contextual Understanding:** Allows focusing on both local and long-range context effectively.
3. **Scalable to Longer Sequences:** Handles longer sequences efficiently, making it suitable for various NLP tasks.

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
Answer:Transformers use positional encodings to inject information about the order of tokens in a sequence. Unlike RNNs or LSTMs, which inherently process sequences in a step-by-step manner and thus retain sequential information, transformers process all tokens in parallel. This parallelization makes them highly efficient but also causes them to lack an inherent sense of token order.

To address this, **positional encodings** are added to the word embeddings to provide a sense of position in the sequence. Without positional encodings, the transformer would treat a sequence as a "bag of words," ignoring word order, which is crucial for understanding context.

How Positional Encodings Are Incorporated into the Transformer Architecture

Positional encodings are added to the input embeddings before they are fed into the transformer layers. Here's the process:

1. **Word Embeddings and Positional Encodings:**
   Each word in the input sequence is represented as a vector (embedding). A positional encoding vector of the same dimension is generated for each position in the sequence.
2. **Addition of Embeddings and Encodings:**
   For each word, its positional encoding is added to its embedding:
   - Input = Word Embedding + Positional Encoding
3. **Feeding into Transformer Layers:**
   The combined vectors are then fed into the transformer layers (self-attention and feed-forward layers). This addition allows the model to incorporate positional information along with word meaning.

Recent Advances in Various Types of Positional Encodings

While the traditional transformer architecture uses **sinusoidal positional encodings**, which are predefined and not learned, recent advances have explored other types of positional encodings:

1. **Learned Positional Encodings:**
   Unlike sinusoidal encodings, which are fixed functions of position, learned encodings treat positions as parameters to be learned during training. This

approach allows for more flexibility, as the model can learn positions based on the data it sees, potentially improving performance on some tasks.

2. **Relative Positional Encodings:**
   Instead of encoding absolute positions, relative positional encodings capture the distance between tokens. This approach is particularly useful for tasks where relative positioning is more important than absolute positioning (e.g., handling long sequences effectively).
3. **Rotary Positional Encodings (RoPE):**
   Rotary encodings introduce rotational transformations to embeddings to incorporate relative positional information. They improve model performance on long-range dependencies and have been used in models like GPT-NeoX.
4. **Efficient Positional Encodings for Long Sequences:**
   As transformers are extended to handle longer sequences, new types of positional encodings have been developed to efficiently handle long-term dependencies without performance degradation. These include sparse or hierarchical encodings, which reduce the computational complexity of positional information.

Differences from Traditional Sinusoidal Positional Encodings

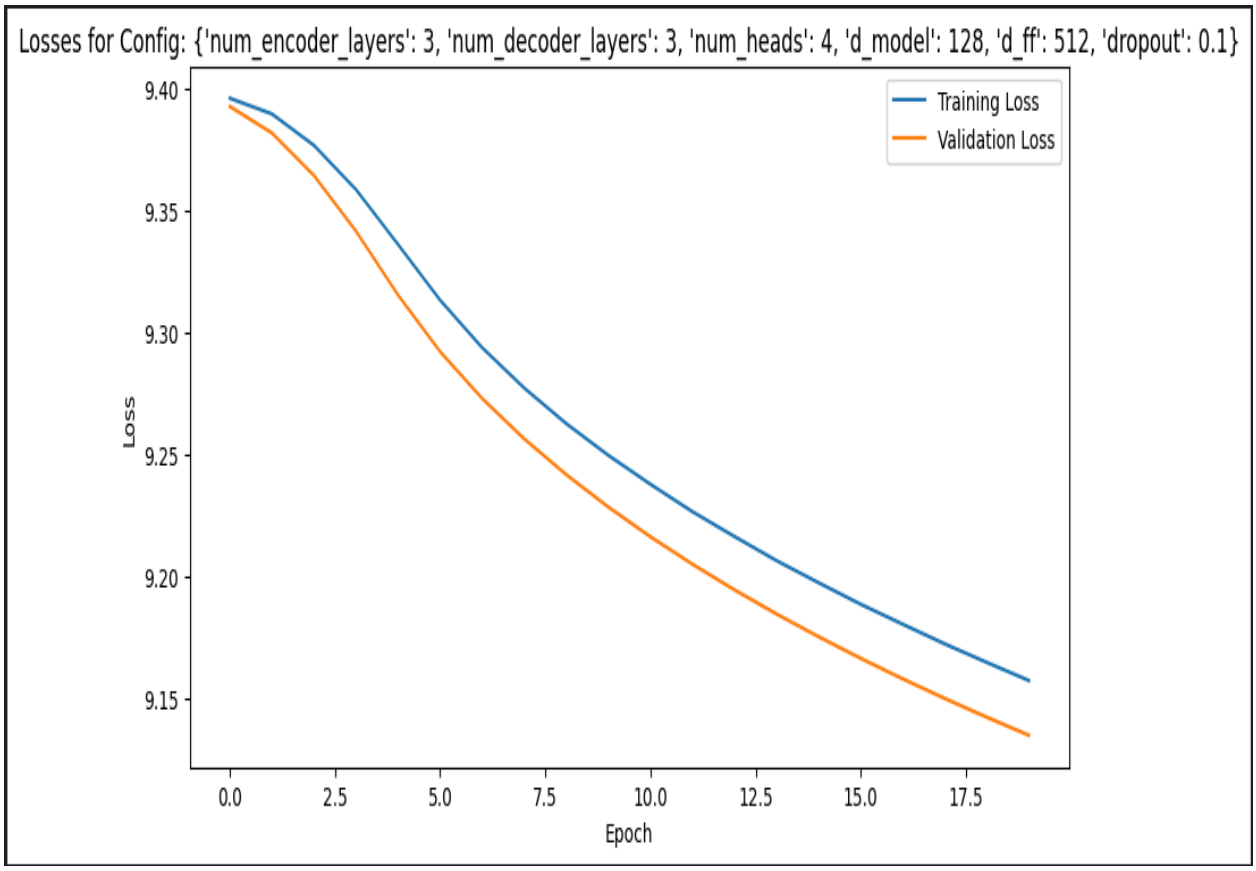
- **Fixed vs. Learnable:** Sinusoidal encodings are fixed and predefined based on a mathematical function, while many recent advances involve learnable parameters that adapt based on training data.
- **Absolute vs. Relative Positioning:** Traditional encodings are based on absolute positions in a sequence. Newer approaches, like relative positional encodings, focus on the distance between tokens, offering greater flexibility.
- **Improved Scalability:** Some recent encodings are designed to be more efficient for long sequences, reducing the computational burden while maintaining or improving performance on sequence tasks.

Hypeparameter Tuning

Model 1:

- Encoder Layers: 3
- Decoder Layers: 3
- Number of Attention Heads: 4
- Embedding Dimension (d_model): 128
- Feed-Forward Network Dimension (d_ff): 512
- Dropout Rate: 0.1

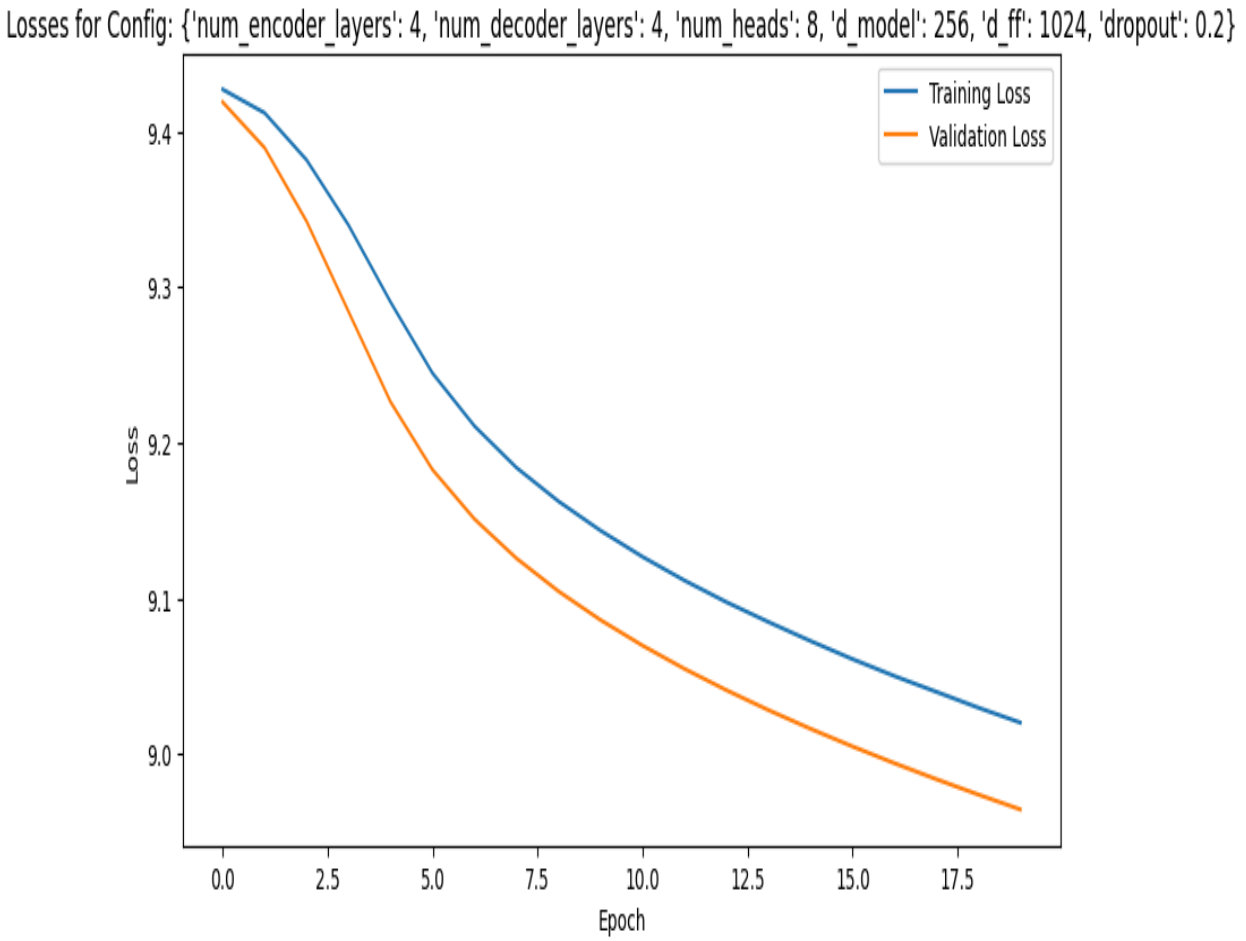
BLEU SCORE:.1655



Model 2:

- Encoder Layers: 4
- Decoder Layers: 4
- Number of Attention Heads: 8
- Embedding Dimension (d_model): 256
- Feed-Forward Network Dimension (d_ff): 1024
- Dropout Rate: 0.2

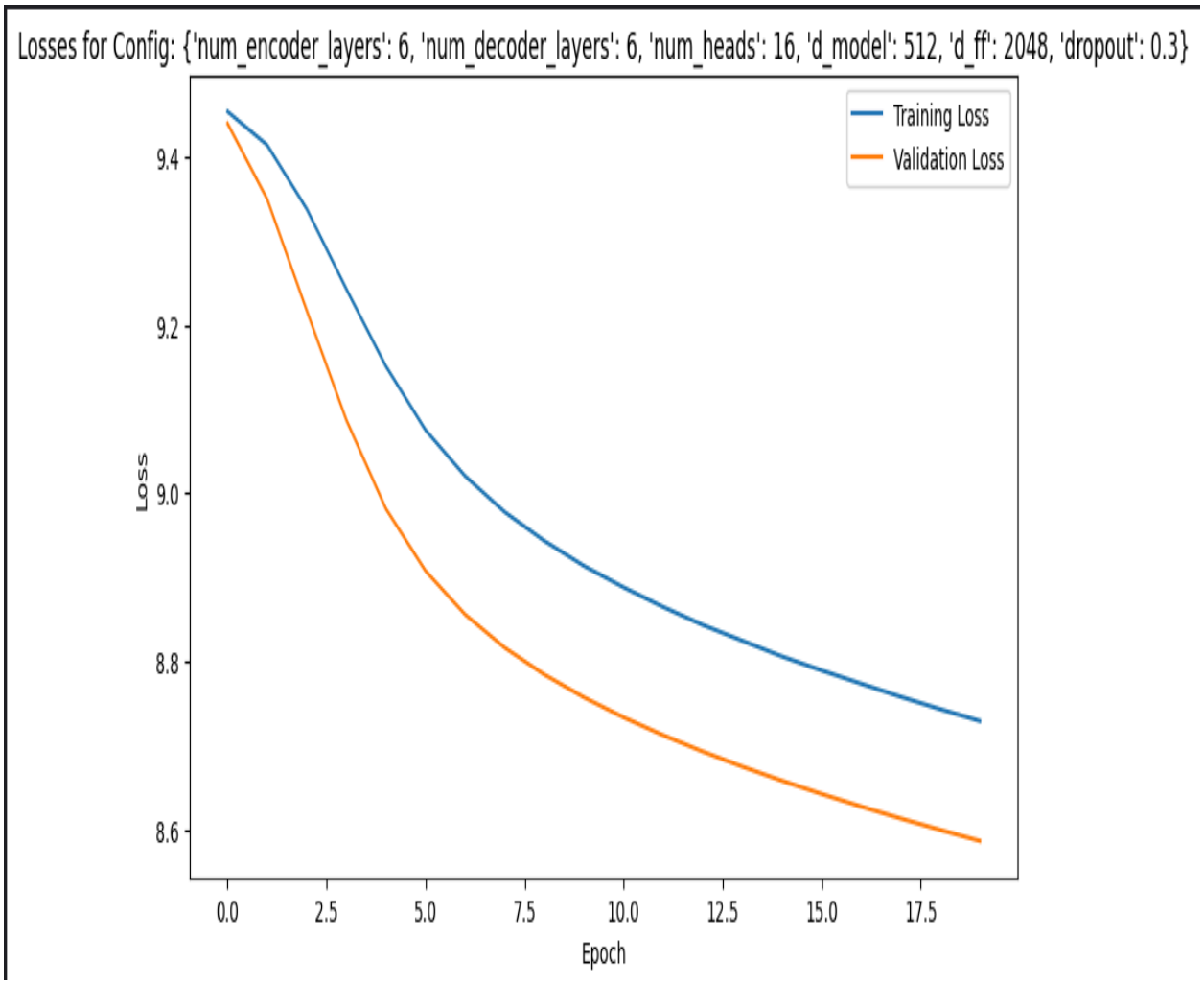
BLEU SCORE: .1656



Model 3:

- Encoder Layers: 6
- Decoder Layers: 6
- Number of Attention Heads: 16
- Embedding Dimension (d_model): 512
- Feed-Forward Network Dimension (d_ff): 2048
- Dropout Rate: 0.3

BLEU SCORE:.1651



Hyperparameters Chosen for the Transformer Model

1. **Number of Encoder and Decoder Layers:**
    - **Model 1:** 3 layers each
    - **Model 2:** 4 layers each
    - **Model 3:** 6 layers each

      The number of layers controls the model's capacity to learn complex representations. A larger number of layers generally enhances the model's ability to capture deeper relationships but also increases training time and the risk of overfitting.
2. **Number of Attention Heads:**
    - **Model 1:** 4 heads
    - **Model 2:** 8 heads
    - **Model 3:** 16 heads

      The number of attention heads allows the model to focus on different parts of the input sequence simultaneously. More heads can enable better parallelism and understanding of context, but also increase computation and parameter count.
3. **Embedding Dimension (d_model):**
    - **Model 1:** 128
    - **Model 2:** 256
    - **Model 3:** 512

      The embedding dimension defines the size of each token's vector representation. A larger dimension can capture more semantic information but requires more memory and can slow down training.
4. **Feed-Forward Network Dimension (d_ff):**
    - **Model 1:** 512
    - **Model 2:** 1024
    - **Model 3:** 2048

      This dimension determines the size of the hidden layer in the feed-forward network. Larger dimensions allow for more complex transformations but also increase the number of parameters, making the model more prone to overfitting.
5. **Dropout Rate:**
    - **Model 1:** 0.1
    - **Model 2:** 0.2
    - **Model 3:** 0.3

      Dropout is used to regularize the model and prevent overfitting by randomly dropping units during training. A higher dropout rate forces the

model to learn more robust features but may reduce learning capacity if too high.

Performance Differences Across Models

The BLEU score results show that **Model 2** performs the best, followed by **Model 1**, and finally **Model 3**. This behavior is consistent with models trained on a relatively small dataset of 30,000 sentences :

1. **Model Capacity vs. Data Size:**
    - **Model 2** strikes a balance between model complexity and the size of the dataset. With 4 layers and moderate embedding/attention dimensions, it effectively learns patterns without overfitting or underfitting.
    - **Model 1** has a smaller capacity, which may limit its ability to learn more complex relationships, but it benefits from reduced risk of overfitting on the small dataset.
    - **Model 3** is the most complex model, with more layers, heads, and larger embeddings. On a small dataset, this model likely overfits, capturing noise rather than generalizable patterns, leading to lower BLEU scores.
2. **Impact of Attention Heads and Layers:**
    - **Model 2's** 8 attention heads and 4 layers provide a good mix of focus across different sequence parts without overwhelming the model with too many parameters.
    - **Model 1's** fewer attention heads and layers may reduce its ability to capture multi-dimensional relationships, resulting in slightly lower performance.
    - **Model 3's** higher number of layers and attention heads increases computational complexity, making it hard for the model to generalize from limited training data.
3. **Dropout and Regularization:**
    - A **0.2 dropout rate in Model 2** effectively prevents overfitting while maintaining learning capacity.
    - A lower **dropout rate of 0.1 in Model 1** helps retain more information, which works for its smaller size.
    - A **higher dropout rate of 0.3 in Model 3**, combined with its large capacity, may lead to under-utilization of the model's full capability, thus hurting performance.

Conclusion

The optimal performance of **Model 2** illustrates the importance of balancing model complexity with data size. While a more complex model can learn deeper relationships, an excessive capacity for a small dataset can lead to overfitting and reduced generalization performance. A moderate model with a balanced number of layers, attention heads, and regularization often achieves better results when training data is limited.