DSA4212:
Transformer Architecture

Group Composition:

1. Lim Jing Rui, A0233959N, jingrui.lim@u.nus.edu

1 Introduction

Transformers are highly effective models for sequence-based tasks due to their self-attention mechanisms, which allow them to capture complex dependencies in input data. This report presents the development of a basic transformer model for sequence reversal, followed by enhancements that enable it to perform text generation. The core architecture of the transformers are documented, explaining its functioning, and detailing the improvements.

2 Simple Transformer for Sequence Reversal

The initial implementation is a simplified transformer model designed to reverse sequences of integers. This section explains how the core components of the transformer architecture were set up and trained to achieve this objective.

2.1 Core Components

The model includes fundamental transformer components, such as self-attention, multi-head attention, feed-forward layers, and positional encoding.

2.1.1 Self-Attention and Multi-Head Attention

The model's attention mechanism allows each token in the sequence to attend to every other token, capturing relationships within the sequence. Self-attention calculates:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V are query, key, and value vectors, and d_k is the key dimension. Multi-head attention enables the model to process these relationships across multiple perspectives by splitting the embedding dimension across heads (Vaswani et al., 2017).

2.1.2 Feed-Forward Layers

The feed-forward layer in the transformer model is implemented as a two-layer fully connected network with a ReLU activation function between the layers. This layer provides additional non-linearity and enhances the model's ability to learn complex transformations within each position of the sequence, independently of others. The feed-forward network is defined as:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

where W_1 and W_2 are weight matrices, and b_1 and b_2 are biases. Each position in the sequence passes through the same feed-forward network independently, enabling the model to learn transformations for individual token representations that are contextually informed by the attention layer (Vaswani et al., 2017).

2.1.3 Positional Encoding

Because transformers lack an inherent sequential structure, positional encoding is added to the embeddings to inform the model of token positions in the sequence. This is implemented as:

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

(Vaswani et al., 2017).

2.2 Training Procedure

The simple transformer model was trained on randomly generated integer sequences and their reversed sequence to learn sequence reversal. A fixed vocabulary size and random sequences allowed the model to focus solely on learning the reversal pattern. Cross-entropy loss was used, and the model output was generated in a single forward pass without sequence-based constraints.

3 Improvements

While effective for reversing sequences, the simple transformer model was limited in its capability to handle more complex language generation tasks. Below, we list key improvements implemented to transition the model into a versatile text generator, suitable for NLP tasks.

3.1 Vocabulary and Text Preprocessing

In order for the transformer to perform NLP tasks, vocabulary creation, normalization, and encoding functions were added. Padding (<PAD>) was added to process varying sequence lengths. Unknown (<UNK>) tokens were later added as many words that were used in testing was not found in the training data set due to its size. These tokens replaced any such "out-of-vocabulary" words.

3.2 Enhanced Positional Encoding

Since sequence lengths were now varied, positional encoding was modified to handle this by creating new encodings if the sequence exceeded maximum length. This made the model dynamically adaptable.

3.3 Decoder Layers, Cross-Attention, and Generation Function

While the encoder layers effectively capture information from the input sequence, they alone cannot generate a new sequence based on that input. Unlike in sequence reversal which is a simple pattern matching task without the need for context, a decoder is essential for sequence-to-sequence tasks, as it enables the model to produce an output sequence that is contextually aligned with the input.

Initially, there was an attempt at sequence generation by modifying only the encoder output, but results could not be produced. Adding decoder layers with an autoregressive generation loop allowed the model to generate tokens sequentially. Additionally, cross attention ideally allows the model to capture key contextual information from the encoder output. Experiments were done with top-k sampling and temperature settings in the generate function, and settling on the current parameters to produce more coherent results.

3.4 Model Hyper-parameters

- **Embedding Size, Attention Heads, Hidden Dimensions, Number of Layers:** Settled on 256, 8, 256, 8 respectively. This allowed the model to produce the most acceptable results based on manual testing.

Current Model Output

```
Prompt: Hi, how you doing
Generated: hi hope doing well doing im doing well
```

  - **num_heads:** 8
  - **num_layers:** 8
  - embed_size: 256, hidden_dim: 256, batch_size: 4

Test Model Output

```
Prompt: Hi, how you doing
Generated: hi how how do how how doing how
```

- **num_heads:** 4
- **num_layers:** 4
- embed_size, hidden_dim, batch_size unchanged

Figure 1: Simplified comparison of outputs with different hyper-parameters

- **Learning Rate:** After testing learning rates between 0.01 and 0.00001, 0.00005 was chosen as it allowed stable, incremental learning without overshooting optimal points.

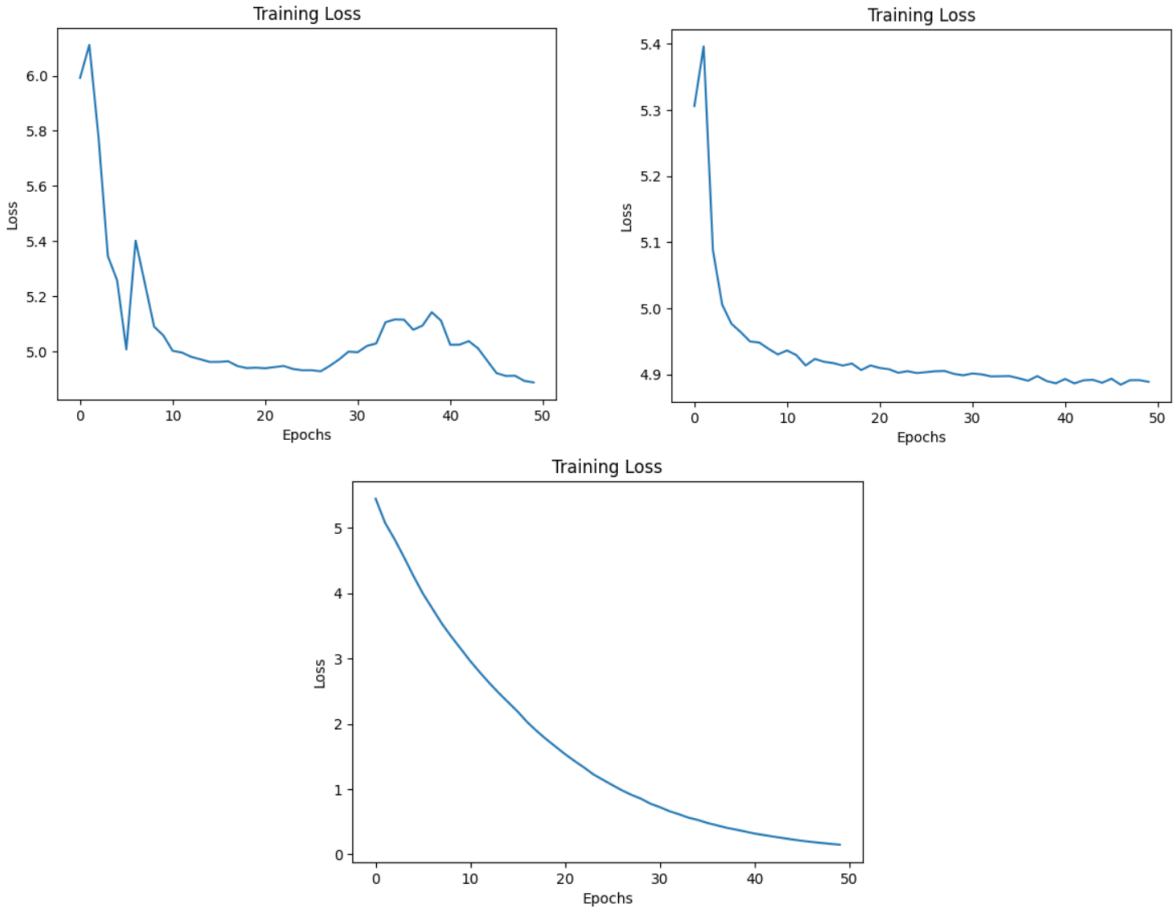

Figure 2: Comparison of Learning Rates: 0.01 (top left) vs. 0.001 (top right) vs 0.00005 (bottom)

- **Number of epochs:** The decision to cut number of epochs to 50 was made after realisation

that the loss plateaued after 50, and performance actually started to degrade.

4 Training and potential improvements

4.1 Dataset

The reason this model was trained on such a small dataset was due to time complexity constraints. Per 10 epochs, it took an average of 60 seconds, and it took roughly 100 epochs for the loss to stabilise. There was attempt to incorporate a 3000 row dataset but even using the first 100 rows only, it took an average of 120 seconds per 10 epochs. It was inefficient for regular testing and experimentation. As such, the decision was to train the model on a specific task (greetings). Testing with a few diverse sentences did not yield particularly better results.

- **Current dataset:** 271.3544 seconds (About 50 seconds per 10 epochs)

4.2 Improvements

In the future, a larger dataset would allow the model to learn more diverse patterns and generate more accurate and meaningful text. An end-of-sequence (EOS) token was not implemented to test the model's ability with the dataset, but it is concluded that training data was insufficient. Admittedly, the model is not yet robust enough but it met the goal of generating a response to a prompt.

5 Conclusion

Through this iterative development process, a functioning transformer model was created and deep insights into the practical considerations of implementing and optimizing transformer architectures were gained. While there's room for improvement, particularly in scaling to larger datasets and more complex tasks, the current implementation provides a solid foundation for further development and experimentation.

References

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Lukasz, K., Polosukhin, I. (2017). Attention is all you need. *Proceedings of the 31st International Conference on Neural Information Processing Systems, 6000–6010. Presented at the Long Beach, California, USA. Red Hook, NY, USA: Curran Associates Inc.*

Sarkar, A. (2023, April 26). Build your own transformer from scratch using pytorch. *Medium. https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb*