DSA4212: Transformer Architecture Assignment

Farrel D. Salim	Kevin Christian	Nixon Widjaja	Wilson Widyadhana
A0236449W	A0219722E	A0236430N	A0244106R

1 Background

Transformers [7] have seen many uses ever since their inception. In this assignment, we attempt to implement such an architecture using mainly JAX [1] and Flax [3], a high performance numerical computing framework and deep learning library respectively. We will be evaluating the performance of our implementation on various tasks pertaining to sequences.

2 Implementation

In implementing a transformer, we have to implement several critical components, including embedding, multi-head attention, positional encoder, encoder, and decoder classes [7]. We have used several resources in order to aid us in our implementation of the transformer, including [6] [4] [5]. We have also referenced the JAX and Flax documentation to gain inspiration on the implementation of our functions [1] [3].

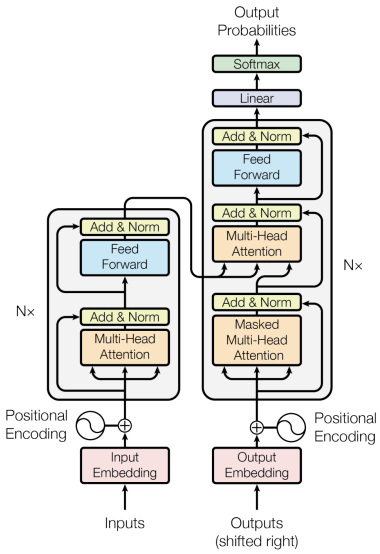

Figure 1: Transformer Architecture, taken from [7].

Figure 1, taken from [7], shows the complete architecture we are aiming to recreate. As we can see above, the transformer's architecture can be broken down into various smaller models.

2.1 Embedding

An embedding layer is used to convert input and output tokens into dense vector representations of size d_{model} that the model can work with. [7] also suggests that the data should be multiplied by $\sqrt{d_{\text{model}}}$. In our model, this layer is implemented as the `Embedding` class in `transformers.py` using the `flax.linen.embed` module.

2.2 Positional Encoding

To let the model learn about sequence ordering, it is common to add some positional encoding to the current vector representation. In this project, we use the sine-cosine positional encoding that used by [7], where given a token's position *pos*, its encoding will be a vector of size d_{model} as follows:

$$\text{PE}(pos) = [\text{PE}(pos, 1), \text{PE}(pos, 2), \ldots, \text{PE}(pos, d_{\text{model}})]$$

Furthermore, each dimension's value of PE(*pos*) will be:

$$\text{PE}(pos, i) = \begin{cases} \sin((\frac{pos}{10000})^{i/d_{\text{model}}}) & \text{if } i \text{ is even} \\ \cos((\frac{pos}{10000})^{i-1/d_{\text{model}}}) & \text{if } i \text{ is odd} \end{cases}$$

We referenced the `PositionalEncoding` function in [6] for our implementation of this model. Our implementation for this model can be seen in the `PositionalEncoder` class in `transformers.py`.

2.3 Combining Embdedding and Positional Encoding: Preprocessing

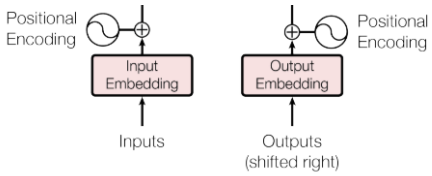

Figure 2: Preprocessing taken from [7].

In our implementation, we have also created the `Preprocessing` class, which will preprocess the input using the previously implemented `Embedding` and `PositionalEncoder` classes to allow it to become usable for further computation.

2.4 Attention

Below are the key equations that relate to the attention mechanism, which we have taken from [7] and [8] for ease of reference (with some modifications). Here, we use $X \in \mathbb{R}^{L \times d}$ as the input sequence, where L is the length of the input sequence. We also denote d to be the hidden state dimension and h to be the number of heads in the multi-head attention. In the standard definition for attention, we have that $Q = XW^q \in \mathbb{R}^{L \times d_k}, K = XW^k \in \mathbb{R}^{L \times d_k}, V = XW^v \in \mathbb{R}^{L \times d_v}, W_Q \in \mathbb{R}^{d \times d_k}, W_K \in \mathbb{R}^{d \times d_k}, W_V \in \mathbb{R}^{d \times d_v}$.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \in \mathbb{R}^{L \times d_v}$$

In multi-head attention, however, we have $W_i^Q \in \mathbb{R}^{d \times d_k/h}, W_i^K \in \mathbb{R}^{d \times d_k/h}$, and $W_i^V \in \mathbb{R}^{d \times d_v/h}$. Below we list the equations for multi-head attention [8]:

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W^O \in \mathbb{R}^{L \times d}$$
$$\text{where head}_i = \text{Attention}(Q_i, K_i, V_i)$$

In our implementation, instead of directly using the projection matrices W^q, W^k, W^v, and W^o, we use `flax.linen.dense` with the `bias_init` parameter initialized as all zeros, following the approach used in [5]. Our implementation for this model can be seen in the `Attention` class in `transformers.py`, which supports both masked and unmasked attention.

2.5 Feed Forward Network

In [7], this network is defined to implement the equation below.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

In other words, this is just a multi-layer perceptron (MLP) with one hidden layer and ReLU as its activation function. In this project, this network is implemented as the `FeedForwardNetwork` class in `transformers.py` using `flax.linen.dense` module and `flax.linen.relu` function.

2.6 Combining Attention and Feed Forward Network: Encoder

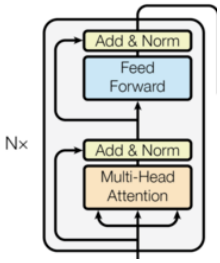

Figure 3: Encoder architecture, taken from [7]

In [7], an encoder is defined as a stack of $N = 6$ identical layers, with each layer having a multi-head attention and a feed-forward network, as can be seen in 3. In our implementation, we created `SingleEncoder` class, which represents a single layer of encoder, using the previously implemented `FeedForwardNetwork` and `Attention` classes. Additionally, we employ layer normalization in `SingleEncoder` class with the help of the `flax.linen.LayerNorm` module. Furthermore, we have also implemented the `Encoder` class in `transformers.py`, which consists of identical layers of `SingleEncoder`s, followed by a final normalization. This final normalization in our `Encoder` class follows the implementation of `Encoder` in [6].

2.7 Combining Attention and Feed Forward Network: Decoder

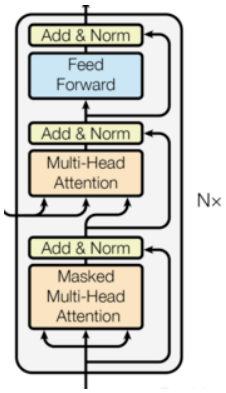

Figure 4: Decoder architecture, taken from [7]

In [7], a decoder is defined as a stack of $N = 6$ identical layers, with each layer having two multi-head attentions and a feed-forward network, as can be seen in 4. In our implementation, we created a `SingleDecoder` class, representing a single decoder layer, using the previously implemented `FeedForwardNetwork` and `Attention` classes, as well as `flax.linen.LayerNorm` for normalization. After implementing the `SingleDecoder` class, it is easy to create a multiple-layer `Decoder` class. Note that similar to the `Encoder` class, we also use a final normalization in our `Decoder` implementation, following the implementation in [6].

2.8 Logits Generator

This is simply the combined linear and softmax layer, which is used to get the final probability from the decoder's output.

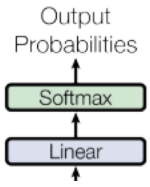

Figure 5: Generator taken from [7]

In our model, this layer is implemented as the `LogitsGenerator` class using `flax.linen.dense` and `flax.linen.log_softmax`. Note that we referenced [6], which uses the `log_softmax` function instead of the regular `softmax` function in their transformer's final layer.

2.9 Combining all together: Transformer

After creating the aforementioned classes, we can simply combine them together to create the `Transformer` class. In our implementation of `Transformer` class, we use the following models:

- Two `Preprocessing` instances: one for preprocessing the input before encoding and one used for preprocessing the output before decoding.
- One `Encoder` instance used for encoding the input sequence.
- One `Decoder` instance used for decoding the output sequence and combining them with the previously encoded input sequence.
- One `LogitsGenerator` instance as the final layer to get the probability of each token appearing as an output.

We also implement the `decode` function in `transformer_train.py`, referenced from the greedy decoding section of [6], to allow our transformer generate output sequence $(y_1, y_2, \ldots, y_{L_{\text{output}}})$ given input sequence $(x_1, x_2, \ldots, x_{L_{\text{input}}})$. More specifically, the `decode` function works as follows:

Algorithm 1: `decode`$(x_1, x_2, \ldots, x_{L_{\text{input}}})$

Denote V_{output} as the set of possible tokens for output sequence;
Start with initial output $\hat{y}_1 \in V_{\text{output}}$;
for $i \in \{1, 2, \ldots, L_{output} - 1\}$ **do**
  Run the transformer with $(x_1, x_2, \ldots, x_{L_{\text{input}}})$ and $(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_i)$ as the input to the encoder and decoder respectively to get $P \in \mathbb{R}^{i \times |V_{\text{output}}|}$;
  Consider the $r \in \mathbb{R}^{|V_{\text{output}}|}$, the last row vector of P. It shows the probability for next output token;
  Choose $\hat{y}_{i+1} \in V_{\text{output}}$ which corresponds to the largest probability of appearing in r;
end
Output $(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{L_{\text{output}}})$;

2.10 Testing the Transformer

In order to check correctness of our implementation, we test our models on three sequence-related tasks: the **sequence copy**, **sequence reversal**, and **sequence sorting** tasks.

To help train the model, we refer to the `Batch` class and `subsequent_mask` function from [6]. In computing the weight updates for our model, we will be using JAX's auto-differentiation capabilities. In order to more easily use specific optimisers and compute new weights, we also use the Optax library [2]. We also refer to [5] for the decision to use `optax.softmax_cross_entropy_with_integer_labels`

as our loss function. Furthermore, the `create_train_state` and `train_step` functions in Flax quick-start [3] also helped us in implementing the training step.

Shown below is our simplified training procedure for each batch, where we use multiple input and output sequences to update the transformer's parameters:

Algorithm 2: `train`(input-sequence, output-sequence)

Denote $(x_1, x_2, \ldots, x_{L_{\text{input}}})$ as the input-sequence;
Denote $(y_1, y_2, \ldots, y_{L_{\text{input}}})$ as the target output-sequence;
Denote V_{output} as the set of possible tokens for output sequence;
Run the transformer to be trained with $(x_1, x_2, \ldots, x_{L_{\text{input}}})$ and $(y_1, y_2, \ldots, y_{L_{\text{output}}-1})$ as the input to the encoder and decoder respectively to get $P \in \mathbb{R}^{(L_{\text{output}}-1)\times|V_{\text{output}}|}$;
Calculate `loss` = Softmax-Cross-Entropy$(P, (y_2, y_3, \ldots, y_{L_{\text{output}}}))$;
Optimize transformer's weights using Optax so that `loss` is minimized.

3 Results

After training the model for 50 epochs, we attained a reasonably good result for the **sequence copy** and **sequence reversal** tasks, with a 0.86 and a 0.78 accuracy, respectively. We have also attained a somewhat reasonable result for **sequence sorting task**, with a 0.43 accuracy. The cross-entropy losses during the training are available in the Appendix.

We can see from the accuracy and cross-entropy loss functions that the model performs quite well in all three tasks, showing the transformer's ability to discern complex patterns. However, the transformer model still vulnerable to overfitting, as the cross-entropy loss for the **sequence sorting task** is the lowest during training 8, even though its accuracy is the worst out of the three tasks. We believe that the model can be further improved by using more regularization techniques such as including dropout layers or using other learning rate schedulers mentioned in [7].

4 Conclusion and Future Work

We conclude that transformers are very powerful sequence modelling architectures, which are able to learn complex patterns given enough training data and time. Given more time for this project, we would like to test our transformer implementation on more complex tasks such as basic arithmetic. Moreover, we would like to see the impact that different hidden dimension sizes, number of layers, regularization techniques, or learning rate schedulers might have on our model's performance.

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[2] DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020.

[3] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2023.

[4] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. Opennmt: Open-source toolkit for neural machine translation. In *Proc. ACL*, 2017.

[5] Philip Lippe. Tutorial 6 (jax): Transformers and multi-head attention.

[6] Alexander Rush, Vincent Nguyen, and Guillaume Klein. The annotated transformer, Apr 2018.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[8] Lilian Weng. The transformer family version 2.0. *lilianweng.github.io*, Jan 2023.

5 Appendix

Below are the graphs of cross-entropy loss during training:

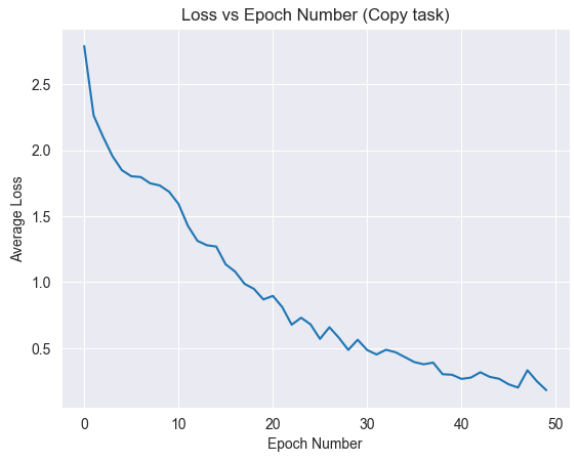

Figure 6: Cross-Entropy Loss for Copy Sequence Task

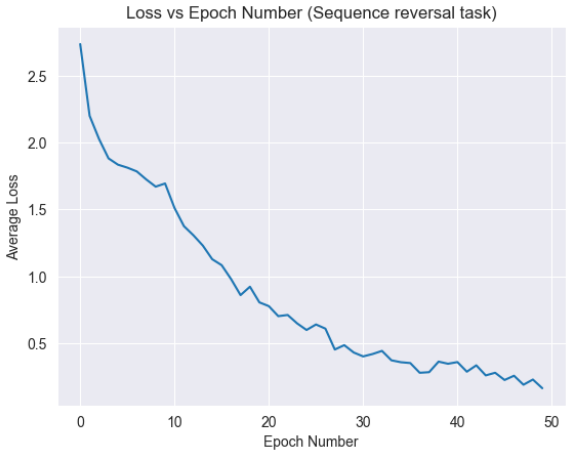

Figure 7: Cross-Entropy Loss for Sequence Reversal Task

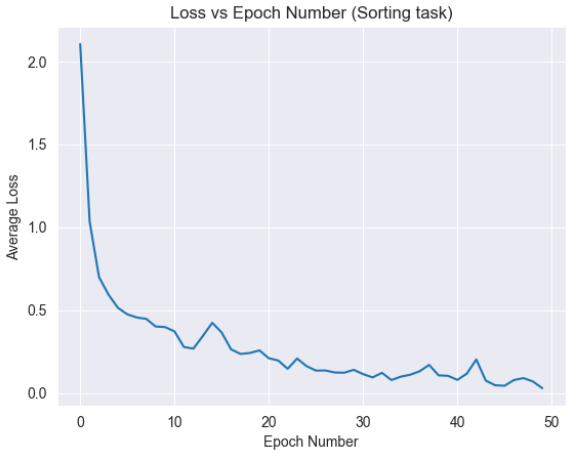

Figure 8: Cross-Entropy Loss for Sequence Sorting Task