Matthew Flynn
CS 231

Project 2

Introduction

For Project 2, we implemented a sequence-to-sequence transformer model that is trained to recover original sentences from preprocessed text.

Transformers are fundamental models in modern NLP, with its encoder component spawning the BERT family of models, and its decoder component leading to popular applications like ChatGPT. Because of this, it is important to understand the inner workings of these models deeply, and this project aims to do that by implementing the full transformer model for an autogressive token generation task.

To do this, the first step is implementing the remaining components of the model, such as the training loop and dataloader. Then, a grid search is conducted to find the best hyperparameters to use for the different experiments, including varying positional encoding strategies, decoding strategies, and model architectures. These experiments closely analyze these different components.

Methods

The transformer architecture builds on and expands earlier deep learning strategies.[1] The core contribution of this architecture is its attention mechanism, which is partly inspired by earlier attention mechanisms from the RNN family of models. However, the attention, as implemented with the query, key, and value matrices in transformers, is fully parallelizable. This allows the training and inference of these models to be much quicker with distributed strategies, as different tokens can attend to one another in parallel.

Transformer

At its core, the transformer architecture is broken into two components: the encoder and decoder. These two components vary in their role and implementation details.

For attention. The encoder employs multi-head self-attention across the whole input sequence, while the decoder, which is an autoregressive generator, employs a masked multi-head self-attention mechanism so that it cannot cheat and see the next token when generating. This enables the encoder to encode contextual information from the whole input sequence and

[1] The canonical paper that introducer the transformer is titled, “Attention Is All You Need” https://arxiv.org/abs/1706.03762

attend to each token in parallel, while its masked counterpart can only attend to previous tokens in the sequence.

For encoders, it consists of some *N* transformer blocks, each of which consists of, in order, multi-head self-attention, add and norm, a feed-forward neural network, and then a final add and norm. This is completed *N* times in a recurrent fashion, before its output is passed to the decoder. All the weights of these blocks are learnable.

The attention blocks for the decoder are somewhat deeper. Of course, the initial attention is masked, to avoid cheating at the autoregressive step. As the transformer is ultimately using matrices, this is accomplished by adding negative infinity to the (usually) upper triangle of the matrix, so that the decoder cannot attend to those positions or tokens. This passes through an add and norm step, with its output being combined with the output of the encoder block for cross-attention. The remaining part of the decoder transformer block mirrors that of the decoder. Likewise, the weights for each part are learnable.

The type of attention used in the original paper was scaled dot-product attention, although later models use different types of attention.

At the end of the *N* decoder blocks, the next token is finally predicted by being passed through a final feed-forward network before the softmax function turns the output into a probability distribution to pick the most probable next token. This token is appended to the input, and the cycle restarts. Temperature values can be applied to chose a value from the *Top-K* next tokens, to provide some variety in model generation. Beam search can also be applied to avoid greedy decoding.

Notably, transformers have no concept of position for the input tokens. Position is important for natural language processing though, so some type of position encoding is added to the inputs to both the encoder and decoder components to give the model this information. For the original transformer, this was accomplished by adding fixed, sinusoidal values to the inputs of both the encoder and decoder components. Learnable encodings appeared in later transformer architectures.

This process is summarized below, as detailed in the original paper:

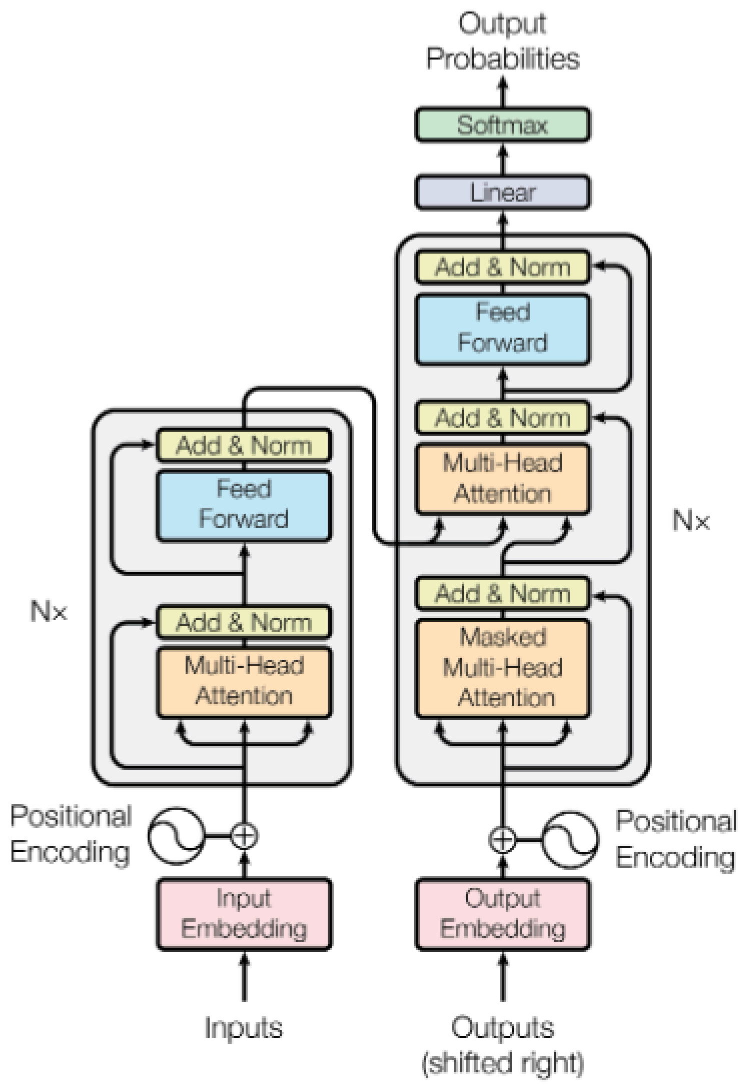


Figure 1: The Transformer - model architecture.

Implementation

For the implementation, I focused on creating flexible module components so that it was simple to call different configurations of the model from the command line. This simplified running many model configurations on a GPU cluster via slurm's *sbatch*.

Besides implementing the loading and encoding of the input data, which follows traditional PyTorch conventions, the only notable implementation detail was the use of abstracting all training components as methods on a Trainer object implemented with the builder pattern. This simplified calling a pipeline and managing the state of a training run to simplify reporting and logging of a run.

Hyperparameters

After implementing the model, a grid search was conducted to find optimal hyperparameters. To limit the total number of experiments to run, only some of the hyperparameters were varied in the grid search:

- Batch Size: [32, 64, 128]
- Number of Encoder and Decoder Layers: [1, 2, 4]
- Model Dimensions: [128, 256]
- Number of Attention Heads: [2, 4, 8]
- Feed Forward Network Dimensions: [256, 512]

The combination of these hyperparameters was run on the Student GPU Cluster, and the total number of results were 103 due to some crashes, detailed below. Because of the already large number of experiments in this grid search, the following hyperparameters were held constant:

- Learning Rate: 1e-4
- Epochs: 4
- Max Target and Source Length: 50
- Dropout: 0.1

These models are also autoregressive, so a decoding strategy at inference time is needed. For this preliminary grid search, only greedy decoding was used.

Inputs

The data for this project is synthetically generated, and the model must learn to autoregressively generate the reconstructed original, target sentence, given a tokenized and lemmatized source sentence. It was from this pair that the model was to learn in a supervised fashion.

These source and target sentences are given as a list of json objects, with each object containing a pair of source and target sentences. An example sentence pair from the train set is provided below:

```
{
   "src": "22 : 63 men that hold jesus mock him , smite him .",
   "tgt": "22 : 63 and the men that held jesus mocked him , and smote him ."
 }
```

This data was parsed and encoded for use with PyTorch dataloaders for efficient training and evaluation loops.

We were provided with train, dev, and test sets for training and evaluation.

Metric

For all experiments, the key metric is the average BLEU score, which is calculated with the generated sequence against the gold sequence at the testing stage in the pipeline. The total, cumulative BLEU score for all of the test set is average to obtain the reported value. We used the BLEU score module from NLTK, and I applied smoothing method 2 (Laplace).

Results

I obtained the following results from the initial grid search and the following experiments.

Grid Search

There were 103 experiments in the grid search, summarized below. Only the varied hyperparameters are included in this summary.

To keep the table compact, the header shorthands below correspond to the following hyperparameters:

- BS: Batch Size
- NumED: Number of Encoder and Decoder Layers
- Dim: Model Dimensions: [128, 256]
- NumAH: Number of Attention Heads
- FFDim: Feed Forward Network Dimensions
- Avg BLEU: Average BLEU Score’

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
32	4	256	2	512	**0.8147**
32	4	256	2	256	0.8121
32	4	256	4	512	0.8103
32	4	256	8	512	0.8081
32	2	256	2	512	0.8039
32	4	256	8	256	0.7995
32	4	256	4	256	0.7992
32	2	256	4	512	0.7988
32	2	256	2	256	0.7967
32	2	256	4	256	0.7899

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
32	2	256	2	512	0.7833
32	2	256	8	256	0.7798
64	4	256	2	512	0.7791
64	4	256	2	256	0.7756
64	4	256	4	512	0.7667
64	4	256	8	512	0.7661
64	2	256	2	512	0.7628
64	4	256	4	256	0.7593
32	1	256	4	256	0.7592
32	1	256	2	512	0.7587
32	1	256	2	256	0.7582
32	1	256	4	512	0.7582
32	1	256	2	256	0.7565
64	2	256	2	256	0.7564
64	4	256	8	256	0.7540
32	1	256	8	512	0.7535
32	1	256	8	256	0.7493
64	2	256	4	512	0.7483
64	2	256	4	256	0.7393
64	2	256	8	512	0.7300
64	2	256	8	256	0.7178
128	4	256	2	512	0.7137
128	4	256	4	512	0.7102
32	4	128	4	512	0.7083
32	4	128	2	512	0.7054
128	4	256	2	256	0.7050

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
32	4	128	2	256	0.6976
128	2	256	2	512	0.6964
128	4	256	8	512	0.6954
32	4	128	8	512	0.6947
32	2	128	2	512	0.6938
128	4	256	4	256	0.6861
32	4	128	8	256	0.6840
32	4	128	4	256	0.68111
128	4	256	8	256	0.6795
128	2	256	2	256	0.6788
32	2	128	4	512	0.6723
32	2	128	2	256	0.6706
128	2	256	4	512	0.6672
128	2	256	8	512	0.6664
128	2	256	4	256	0.6591
128	1	256	2	512	0.6543
32	2	128	4	256	0.6516
128	2	256	8	256	0.6483
128	1	256	4	512	0.6449
32	1	128	4	512	0.6422
32	2	256	8	256	0.6412
128	1	256	4	256	0.6403
32	1	128	2	512	0.6383
128	1	256	2	256	0.6375
64	4	128	4	512	0.6370
64	4	128	2	512	0.6360

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
64	4	128	2	256	0.6352
32	1	128	2	256	0.6342
32	1	128	4	256	0.6335
128	1	256	8	512	0.6303
64	4	128	8	512	0.6276
32	1	128	8	512	0.6231
128	1	256	8	256	0.6190
64	4	128	8	256	0.6133
64	4	128	4	256	0.6132
32	1	128	8	256	0.6097
128	4	128	2	512	0.5702
128	4	128	4	512	0.5625
128	4	128	8	512	0.5600
128	4	128	4	256	0.5481
128	4	128	2	256	0.5459
128	4	128	8	256	0.5365
128	2	128	2	512	0.5345
128	2	128	8	512	0.5234
128	2	128	5	512	0.5182
128	2	128	2	256	0.5120
128	2	128	4	256	0.5083
128	2	128	8	256	0.4973
128	1	128	4	512	0.4968
128	1	128	2	256	0.4900
128	1	128	8	512	0.4865
128	1	128	2	512	0.4843

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
128	1	128	4	256	0.4783
128	1	128	8	256	0.4491

Due to the combination of the hyperparameters, there should have been more. However, many crashed on the grid for unknown reasons, apparently due to possible resource contention when slurm was allocating resources. I reran the full set of batch size 32, which proved to be the most effective batch size, and so I stopped the reruns there in order to save computational resources.

As the table summary shows, the best hyperparameter configuration is as follows:

- Batch Size: 32
- Number of Encoder and Decoder Dimensions: 4
- Model Dimensions: 256
- Number of Attention Heads: 2
- Feed Forward Network Dimensions: 512

This configuration attained an average BLEU score of 0.8147, and it will be used as the baseline for the positional encoding experiment.

Experiment 1: Positional Encoding Strategies

The first experiment is comparing the already-implemented fixed sinusoidal positional encodings with learnable positions encodings, as available in PyTorch.

Encoding	BS	NumED	Dim	NumAH	FFDim	Avg BLEU
Sinusoidal	32	4	256	2	512	0.8147
Learnable	32	4	256	2	512	**0.8318**

The learnable embeddings, all else equal, show a performance gain of 0.0171 points for the average BLEU score. This is relatively significant, given that the maximum BLEU score is one. With only changing this single module in the model, performance immediately increased.

Experiment 2: Decoding Algorithms

There are two types of decoding algorithms to experiment with, both of which were already implemented. These are: greedy, which immediately picks the next best token; and beam search, which searches ahead some *k* possible paths, where *k* is a hyperparameter. The intuition is that the next best token may not be the best overall, once future possible tokens are taken into account. However, this significantly increases computational costs.

Because Experiment 1 shows that learnable position encodings are better than fixed, sinusoidal ones, so that model is used as the baseline here.

Strategy	Beam Width	BS	NumED	Dim	NumAH	FFDim	Avg BLEU
Greedy		32	4	256	2	512	0.8318
Beam	3	32	4	256	2	512	0.8337
Beam	5	32	4	256	2	512	**0.8342**
Beam	10	32	4	256	2	512	**0.8342**

Another important consideration when comparing greedy and beam search is how long inference takes.

Strategy	Beam Width	Inference Time	Avg BLEU
Greedy		1.49 Hours	0.8318
Beam	3	4.68 Hours	0.8337
Beam	5	12.01 Hours	**0.8342**
Beam	10	22.07 Hours	**0.8342**

This experiment shows that while beam search does provide very marginal performance increases (0.24 average BLEU gained), it is at great computation cost. It also shows that increased beam width does not always increase performance, as a beam width of 5 and 10 had the same performance at two decimal places, and a beam width of three only lagged behind by 0.05 average BLEU.

Experiment 3: Model Architecture Variants

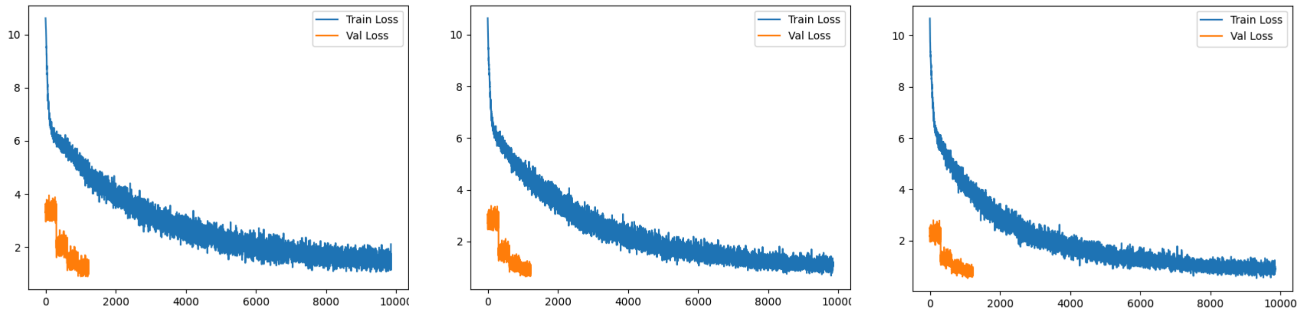
For model architecture experiments, I experimented with the number of attention heads and the encoder/decoder depth. Because the learnable position encodings performed better than the fixed sinusoidal encodings, that model was loaded as a baseline for experiments. Because of that, these experiments and scores differ from their equivalent in the original grid search. The same default hyperparameters were used as noted above.

Number of Attention Heads

The below table experiments with, all else equal, how the number of attention heads impacts performance.

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
32	4	256	2	512	0.8292
32	4	256	4	512	**0.8312**
32	4	256	8	512	0.8256

These experiments show that for this seq2seq task, increasing the number of attention heads does not always increase performance. The best performing model here has four attention heads, which appears to be the sweet spot for attention heads, all else equal. Two attention heads only lagged behind by an average BLEU score of 0.2, and that was the number of attention heads suggested in the original grid search with fixed sinusoidal position embeddings. This experiment is consistent with that, although the four attention heads beats it slightly with four attention heads. Ultimately, the best score performed similarly but did not beat the learnable encoding baseline, although the number of attention heads swapped from 2 to 4.



The loss curves summarize this experiment. As the average BLEU score is not significantly different, they appear almost identical. From left to right, this shows the loss curve for 1, 2, and 4 encoder and decoder layers over the training steps.

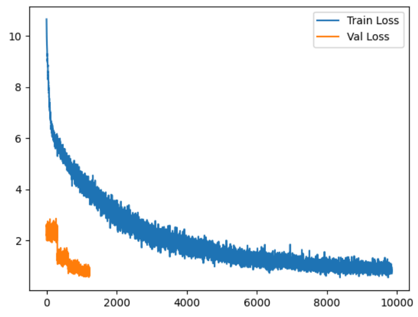
Number of Encoder and Decoder Layers

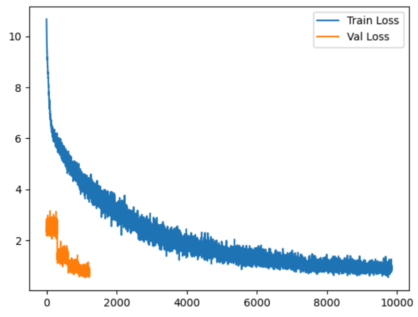
I also experimented with the depth of the encoder and decoder module layers. Again, all other hyperparameters were held equal.

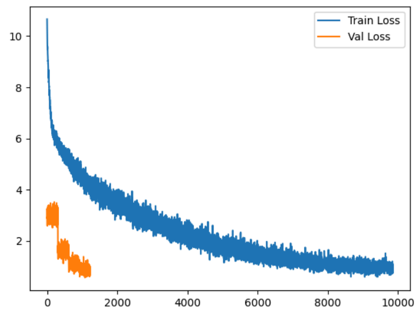
BS	NumED	Dim	NumAH	FFDim	Avg BLEU
32	1	256	2	512	0.7837

BS	NumED	Dim	NumAH	FFDim	Avg BLEU
32	2	256	2	512	0.8159
32	4	256	2	512	**0.8313**

These experiments show that, unlike the number of attention heads, increasing the depth of the encoder and decoder modules does clearly increase performance. However, like the number of attention heads, the best score did not differ significantly from the learnable encodings baseline.







These loss curves, like above, do not show significant differences in the loss curves, as the average BLEU scores and learning did not differ significantly. From left to right, these charts have 2, 4, and 8 attention heads over the training steps.

Example Generated Outputs

Throughout the many experiments, many examples of generated outputs on the test set were logged. To show an illustrative example, the below was taken from the model trained with learnable position encodings, but otherwise with the optimal hyperparameters obtained in the grid search.

Prediction	['"', 'long', 'time', ',', 'indeed', '!"']
Gold	['"', 'long', 'a', 'time', ',', 'indeed', '!"']
BLEU Score	0.8464
Prediction	['}', 'the', 'history', 'time']
Gold	['}', 'and', 'times']
BLEU Score	0.25

Prediction	['enter', 'rosse', '.']
Gold	['enter', 'rosse', '.']
BLEU Score	1.0

These three examples show a range of performance. And as with autoregressive models, a beginning sequence token is injected to start the generation with the source encoding inputs as context. And while only short examples are included here due to space constraints, this pattern is consistent with longer inputs.

In summary, the model is capable of performing generally well, but missing smaller intermediate tokens while being able to capture the rest of the generation pipeline, such as the first example missing ‘a’. Generation can also be overall very confused, only capturing the single first token correctly, such as the second example. There appears to be some path dependency involved in generation that if the earlier a token is wrong in the sequence, the more likely later tokens are to be wrong, in a cascading error pattern. This is likely primarily a problem with a greedy decoding strategy, as beam search will look ahead to hopefully alleviate this issue. Finally, the model is also capable of entirely reconstructing the gold sequence without any error, such as the last example.

And while these examples show that the model is not perfect, it can perform well at this task and produce plausible outputs, even when that is not reflected in the BLEU score.

Analysis

The transformer architecture is rich and dynamic, and there are many components and modules to experiment with.

The first step is always to conduct a grid search over the available hyperparameters of the baseline model architecture to find the optimal hyperparameters. This grid search showed the most amount of variance in scores, with the best combination achieving an average BLEU score of 0.8147, while the worst performing model achieved only 0.4491. The model seemed most sensitive to batch size, with the smallest batch size used in the experiments (32) consistently performing the best, and the largest consistently performing the worst (128). Future experiments should increase the range of smaller batch sizes (e.g., 1, 2, 8, 16) to see if the trend continues with smaller batch sizes increasing performance. This is likely to increase training time, though, so there is a tradeoff.

With the hyperparameters from the baseline, the first experiment was modifying the position encoding strategy. The baseline model used the fixed, sinusoidal position encodings from the original transformer. That was compared with learnable position encodings, as provided by PyTorch. This simple modification showed immediate improvement in the model, as the average BLEU score improved from 0.8147 to 0.8318. From this experiment, it is clear that learnable position encodings are important for optimizing performance. Future experiments should experiment with other position encoding strategies, such as the RoPE strategy, as used in Llama models. This was used as the new baseline for experiments two and three.

Decoding algorithms was the hardest experiment to conduct as beam search (and varying its beam width hyperparameter) was very computationally expensive, with some experiments lasting almost almost a whole day using a 48gb GPU. However, beam search does ultimately perform better very slightly better than greedy decoding, with a beam width of ten improving performance from 0.8318 to 0.8342. This is a tradeoff as well, and if optimal performance is required, the computational resources may be worth it, especially after certain optimizations are made to beam search decoding. Perhaps the implementation here was not fully optimized, but even if so, the miniscule performance gains are likely not worth it.

The transformer is very flexible as well, and the different components and modules can easily be modified. For these final two experiments, the learnable position encodings were used as a baseline to differentiate these experiments from those obtained during the grid search, as the hyperparameters overlapped.

For the number of attention heads hyperparameter, it was varied between 2, 4, and 8. Intuitively, it may make sense that the larger architecture with 8 heads performed best. This experiment, however, showed that 4 performed best, followed closely by 2. The performance did not differ significantly from the baseline, with 0.8318 decreasing very slightly to 0.8312. This is de minimus, and future experiments should also experiment with 1 attention head.

Finally, for the number of encoder and decoder layers, the intuition that more layers and larger models is better holds true, with 8 layers, when compared to 2 and 4 layers. However, like with the number of attention head experiments, the best performing model performed very slightly worse than the baseline, decreasing from 0.8318 to 0.8313. Like above, this is de minimus, and future experiments should continue increasing the number of layers to see if performance continues to increase, e.g. 16 and 32 layers. This is, of course, more computationally expensive, but it should be ultimately cheaper than beam search.

Conclusion

This project shows the effectiveness of transformer models in learning new seq2seq tasks, and also how the model is sensitive to different hyperparameters and modules. Ultimately, beam search with learnable position encodings proved to be the best-performing model, but at great computational cost. For a relatively efficient model, finding the best hyperparameter configuration from a grid search with learnable position encodings is the best strategy.

There remains many possibilities for future work. While the grid search of the base model did include 103 hyperparameter combinations, there are many more that could be run. It would make sense to focus only on smaller batch sizes of 32, e.g. 2, 4, 8, 16, 32, and exploring a smaller number of attention heads (2, 4), with deeper encoder and decoder blocks (8, 16, 32). All runs should be with learnable position encodings, Depending on the task, there is computational tradeoff to this extended grid search, but it will help identify the optimal model configuration.

And as the transformer model is very dynamic with the possibility to swap out different modules, such as positional encoding strategies, attention types, residual connections, etc., there are many other variations to consider, before only considering hyperparameters.

This lack of other experiments with this project is ultimately a limitation, and it also does not compare the transformer architecture with other architectures. It is only compared with itself, and so it is not clear it is the best model overall for this specific seq2sec task. At a minimum, RNN, LSTM, and GRU models should also be included as they have shown good performance on similar seq2seq tasks, although they, of course, have their own limitations.

Another limitation is that the only task explored here was a seq2seq task that learned to reconstruct sentences. It may be useful to explore other tasks for EncoderDecoder transformers for a better comparison, such as a machine translation task.

The last limitation to note is that the primary metric used here is the average BLEU score across the test set. While it is outside of the scope of this paper to discuss the relative issues with the BLEU score, it may also be helpful to use a different but similar metric, such as a ROUGE score, to compare if one score better captures the performance of the model.

Overall, this was a very useful project in understanding the transformer model architecture more deeply, and learning how different hyperparameters and module components influence the model's performance on this seq2seq task.