Deep Learning Lab **2021**

Student: Naga Venkata Sai Jitin Jami

Mathematical Problem Solving with Transformers

1. Problem Setups and Preliminaries

Properties of data can be found in table 1

	Training Set	**Validation Set**
Character vocabulary size	33	33
Number of questions	1999998	10000
Number of characters	63898109	339578
Average length of question	32	34
Total number of words	14531838	73641

Table 1: Text data properties

2. Dataloader

2.1. Q1

For the implementation, the input (question) and output (answer) have different vocabularies by default.

2.2. Q2

The `unk` token is returned by `Vocabulary.get_idx` when the user is looking for the token for a character if its not part of the `Vocabulary` and the `extend_vocab` flag is set to `False`.

3. Model

A Seq2Seq Transformer model was implemented using sub-modules from `torch.nn` for embedding layers, encoding layers, decoding layers. A positional encoding function was already provided. We first pass the `source` and `target` through the `embedding` layer. We then pass the `source_embedding` through the encoder to get the `memory` (also known as `context`). We then use this `memory` and `target_embedding` to get the final output. We also use the encoder and decoder methods for encoding and decoding in greedy search. The implementation can be found in the attached `python` file under class name `Seq2SeqTransformer`.

4. Greedy Search

The implementation can be found in the attached `python` file under function name `greedy_decode` and `translate`.

4.1. Q1

In the greedy search algorithm, we use the trained transformer to get the `memory` data out of the `source` (which is the essentially the question). We use `memory` and the `decode` function to get the first character of our `target` (the character token with the highest probability amongst the tokens in `target_vocab`). We then use this first character and `memory` to get the next character and so on while we concatenate the prediction to the previous prediction.

4.2. Q2

The greedy implementation takes advantage of the `encode` and `decode` function in the Seq2Seq transformer which uses `nn.TransformerEncoderLayer` and `nn.TransformerDecoderLayer`. We use the `encode` function to learn the context vector which is then used by the `decode` function to generate the logits. Once the probability distribution was present, implementing greedy search was straightforward where we pick the character with the highest probability in the vocabulary.

4.3. Q3

A stopping mechanism was implemented to terminate greedy search once the `'<eos>'` tag is predicted. The termination of search once we reach maximum length is implemented in the `translate` function, where the prediction length is capped at maximum target length with the flag `max_len`.

4.4. Q4

A batch mode evaluation implementation was carried out in the `batch_greedy` function where the greedy search is carried out batch by batch. The evaluation moves to the next batch only once we finish all the questions in this batch. There is also an argument to set the number of batches you want to carry out greedy search on, by default this is set to 10. The function also prints the question and prediction for predictions that are correct.

5. Accuracy Computation

Accuracy of the model was calculated using the `translate` function which uses greedy search. The accuracy test was run on the first 100 batches, so a total of 6400 questions in the training and validation datasets. The `target` and predictions were converted to strings and compared, if they were equal that would be counted as a successful prediction. A score was calculated for both training and validation datasets after every epoch. The implementation can be found in the attached `python` file under function name `Accuracy_Computation`.

6. Training

6.1. Loss Function

Cross-Entropy loss function was used in the training of the transformer because the Soft-max classification layer is built into the `torch` function.

6.2. Training Code

The Seq2Seq transformer had separate `encode` and `decode` functions that were used in the training pipeline especially for the validation section of the pipeline. Training losses/accuracy, validation losses/accuracy were tracked after every epoch. Some basic model predictions (questions from the validation set) are printed after every 5 epochs.

6.3. Gradient Accumulation

Gradients are accumulated for every 10 batches. So `optimizer.step()` was called only if the batch number was divisible by 10. This way gradients accumulated over 10 batches or 640 questions during training.

7. Experiments

7.1. Training Pipeline

The training and model parameters used for the main experiment are in table 2

Batch Size	64
Effective batch size	640
Optimizer	Adam
Learning Rate	0.0001
Gradient Clipping	0.1
Embedding Size	256
Attention Heads	8
Feed Forward Dimensions	1024
No. of encoders	3
No. of decoders	2
No. of epochs	20

Table 2: Training and Model parameters

7.2. Training Curves

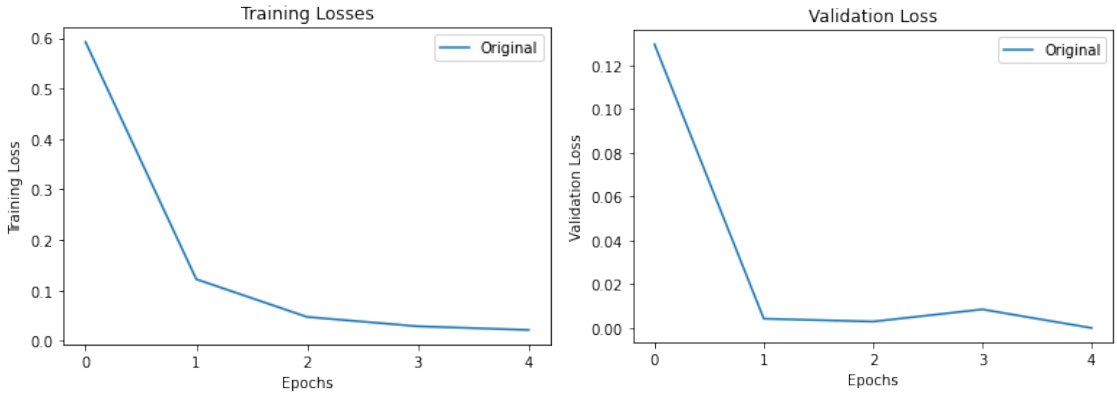
Training was carried out over 20 epochs but acceptable accuracy was reached after 5 epochs. In figure 1, we can see the training losses calculated by the function `Train_epoch`, validation losses calculated by the function `evaluate`, the training accuracy and validation accuracy calculated by the function `Accuracy_Computation` over the first 100 batches. Apart from the calculating the performance metrics during training, the prediction of the model on random questions picked out from the validation set and printed during training. The following is the final output after training for 20 epochs. The `translate` function was used for this.

```
Question: What is the ten thousands digit of 62795675?
9
Question: What is the hundred thousands digit of 82923295?
9
Question: What is the tens digit of 70750657?
5
```

As we can see, the predictions are accurate for the questions picked out randomly from the validation set.

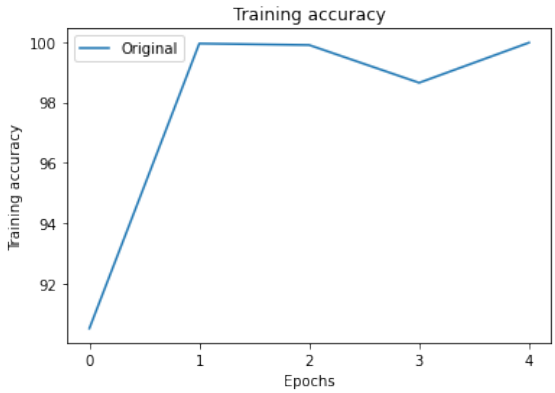
7.3. Hyper-Parameter Tuning

Hyper-parameter tuning was carried out by reducing the model parameters `NHEAD` (number of attention heads) and `FFN_HID_DIM` (Feed Forward Dimensions). The reduced parameters can be

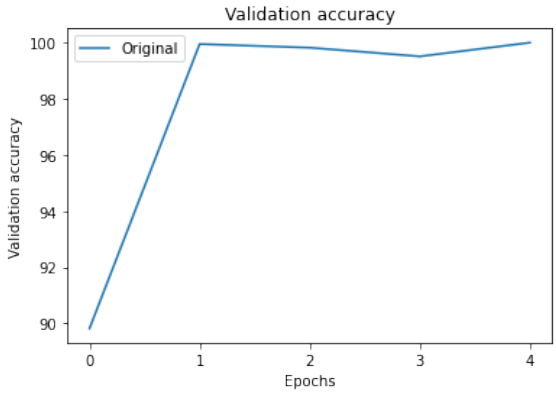


(a) Training Losses

(b) Validation Losses



(c) Training Accuracy



(d) Validation Accuracy

Figure 1: Base model performance plots

found in table 3. Training was carried out over 20 epochs but acceptable accuracy was reached after 5 epochs. The plots can be found on figure 2

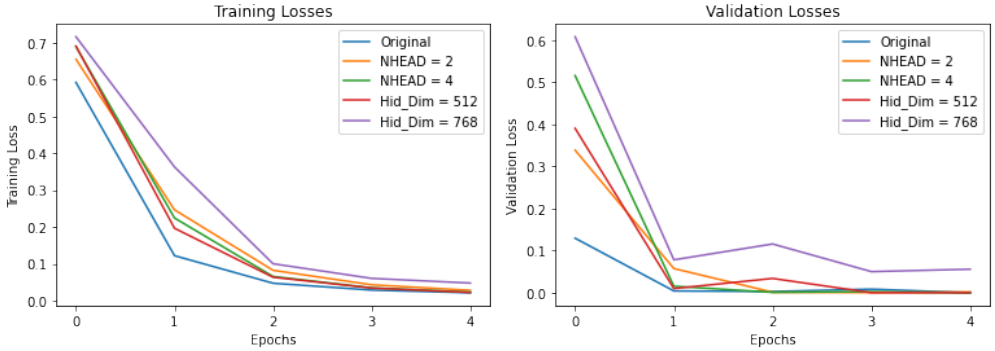
Case Tag	NHEAD	HID_DIM
Original	8	1024
NHEAD = 2	2	1024
NHEAD = 4	4	1024
HID_DIM = 512	8	512
HID_DIM = 768	8	768

Table 3: Hyper-parameter Tuning

We can see from the plots related to training loss (fig 2a) and validation loss (fig 2b) that not all models achieve peak performance. In both plots we can see that the original model achieves best performance fastest and the `HID_DIM = 768` takes the longest number of epochs to reach best performance. But in the end all models reach an acceptable level of accuracy as evidenced by the validation plots (fig 2c and 2d). But the main reason for reducing the size of the model is to see if any time can be saved for achieving the same performance. We can see from figure 2e that the average time across 20 epochs is pretty much the same for all the models with minor variations. So perhaps its prudent to go with the original model since it achieves best performance the fastest.

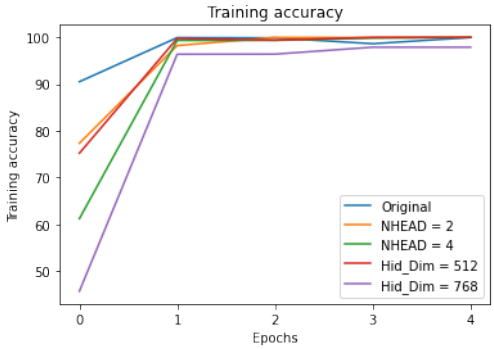
7.4. Experiment: Compare - Sort

All the above experiments were run on the dataset `numbers__place_value` that is part of the DeepMind mathematical dataset. Now experiments were run on a different part of the same dataset called `compare_sort`. Questions in this dataset were of the form:

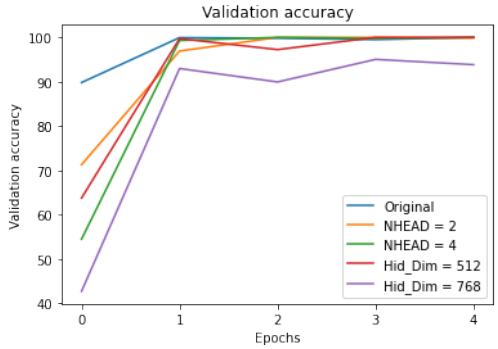


(a) Training Losses

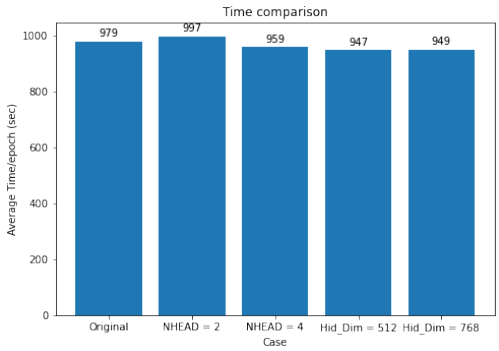
(b) Validation Losses



(c) Training Accuracy



(d) Validation Accuracy



(e) Time comparison

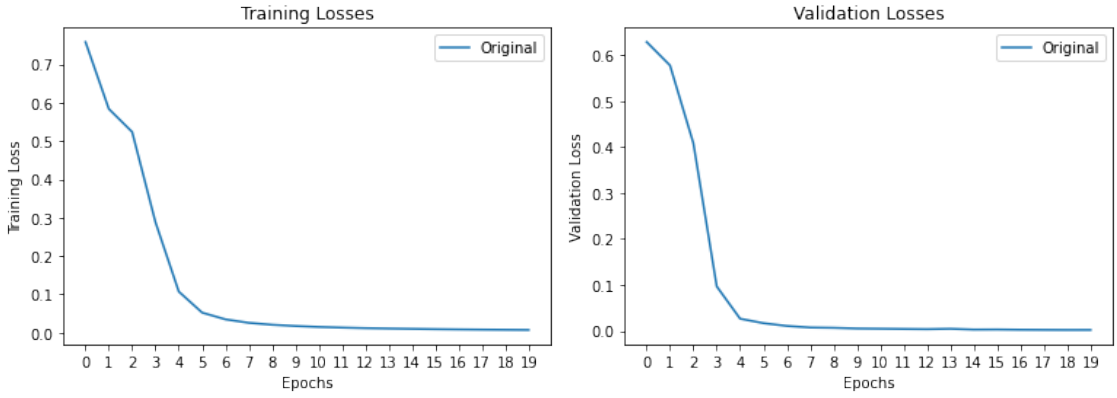
Figure 2: Hyper-parameter tuning

```
Question: Put 0.4, 5, 30, 50, -2, 16 in descending order.
50, 30, 16, 5, 0.4, -2
```

Hence, it is safe to say that this task is more complex than the previous one. The fundamental difference is the length of the answer which is varying according to the question and the target vocabulary is also larger than the previous case. However after training the model for 20 epochs, training and validation accuracy of 99% was achieved. The performance plots can be found on figure 3.

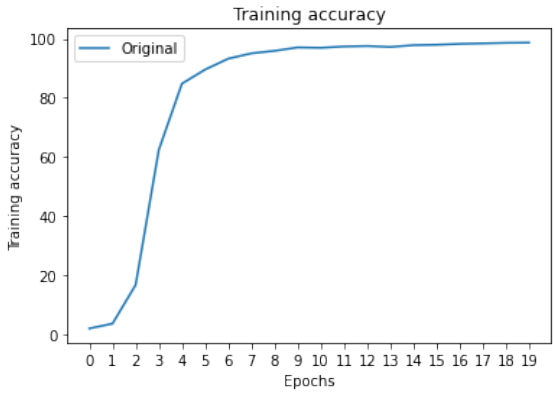
You can see that it takes much longer (in terms of epochs) to train the model to an acceptable accuracy. A similar test was run on this task where random questions were picked from the validation set to test its accuracy ourselves. The results are acceptable for the 3 questions that were picked out.

```
Question: Put 0.4, 5, 30, 50, -2, 16 in descending order.
50, 30, 16, 5, 0.4, -2
Question: Put 3/4, -217, -15/2, 2/5 in descending order.
3/4, 2/5, -15/2, -217
Question: Put 70, 102018, -5 in descending order.
102018, 70, -5
```

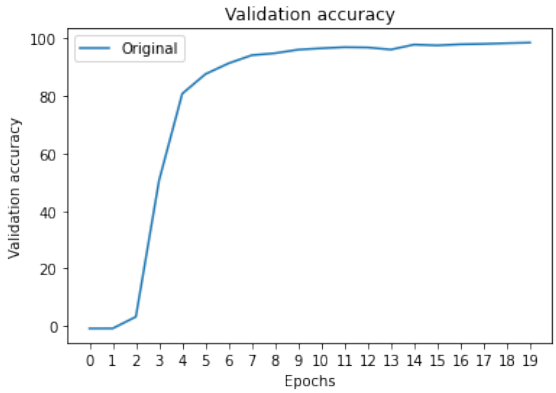


(a) Training Losses

(b) Validation Losses



(c) Training Accuracy



(d) Validation Accuracy

Figure 3: Base model performance plots for "Compare_sort" task

7.5. Experiment: Algebra - Linear 1D

A transformer model of similar structure and parameters was used on a third dataset called `algebra__linear_1d`, also part of the DeepMind mathematical dataset. As opposed to the 2 previous cases, this dataset was rather hard to train and predict with even after 20 epochs. Only a training accuracy of 45% and validation accuracy of 22% was achieved. Some correct answers predicted by the model from the validation dataset are as follows:

```
Question: Solve 49*l + 45*l - 125 - 63 = 0 for l.
2
Question: Solve -64*t + 1387 - 848 + 933 = 0 for t.
23
Question: Solve -21*v - 96 = 156 for v.
-12
```