Solving High-School Probability Questions with Transformers

Introduction

Our task was to solve simple probability questions that middle to high-school students may encounter. The problem is a sequence to number problem.

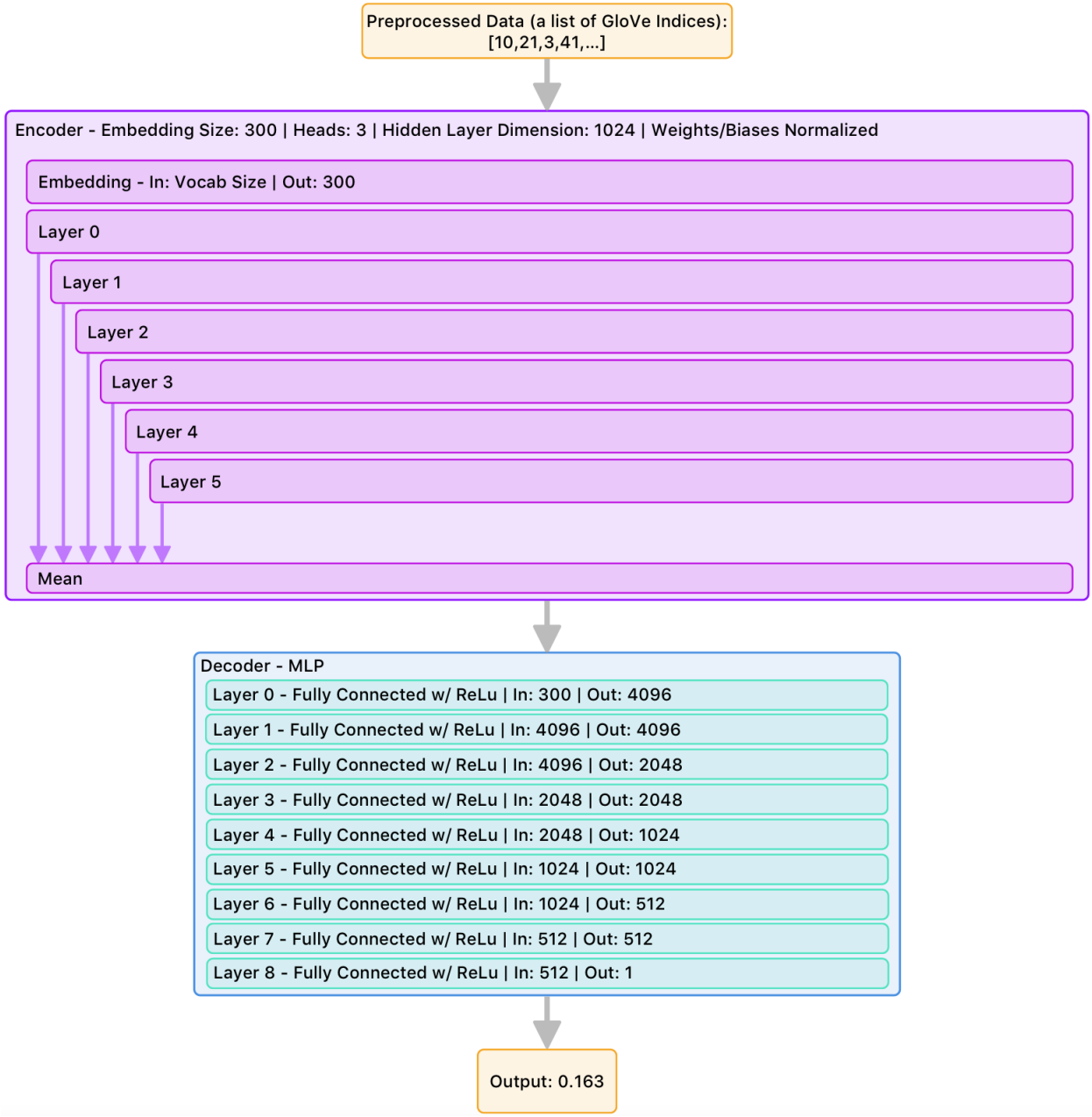
Example:

- Question 1:
  - What is prob of picking 1 b and 1 p when two letters picked without replacement from tpppbbpbbb?
  - Solution: 0.444
- Question 2:
  - Four letters picked without replacement from {y: 6, b: 1, u: 4}. What is prob of picking 2 y and 2 u?
  - Solution: 0.273

We implement a transformer model that takes in a sequence representing the sentence and outputs a numerical value (eg. 0.123) equivalent to the fraction answer that is provided by our dataset.

When our model predicts a number, our accuracy function will check that the solutions (evaluated from fraction to a decimal eg. 7/16 = 0.4375) from the dataset and our predictions are within 3 decimals to each other. We do this to simplify the task as otherwise our model might be wrong due to being off in some less significant decimals. This also simplifies the problem by turning it from a sequence prediction to a scalar value prediction. We do however experiment with this and see if we are able to make it accurate to more decimal places without a huge impact on accuracy.

Model Figure



Model Parameters

The total number of trainable parameters is 40,522,705. We start off with a simple embedding encoder layer (nn.Embedding). The encoder has 117000 weight parameters since our input layer is the size of vocabulary and our output dimension is embedding dimension (d_model). We found that the ideal embedding dimension was 300. We then have 6 Transformer Encoder Layers. Each of these layers is made of a feedforward network where d_hid (1024) is essentially

the dimension of the feedforward network. All the layers have multi-head attention (3) to learn information from multiple subspaces at all positions. The parameters of these layers are divided into self attention projection in and out dimensions, 2 linear layers and 2 normalization layers. The number of parameters in each of these sublayers remains the same throughout the 6 layers of the encoder. During training, we found the optimal numbers for transformer encoder layers, hidden dimension in each layer, and number of heads for multi-head attention models were 6, 1024, and 3 respectively. Finally, we have our decoder layers. Whilst training, we found that our model needed multiple decoder layers to make our model complex enough to learn probability. Therefore, we have 9 decoder layers where the first layer essentially increases the dimension of the output and then, we proceed to bring the dimensions down to 1 whilst applying the Relu function after each layer to introduce non-linearity. The weights of the decoders are therefore the input dimension multiplied by the output dimension (eg. decoder layer weight parameters = 4096 * d_model (300) = 1228800). The number of parameters for the bias of each layer is simply the output dimension of the layer.

The table for these parameters can be found in the Appendix.

Model Examples

Excellent performance:
Question - Three letters picked without replacement from {f: 7, g: 6, b: 4}. Give prob of picking 1 b and 2 g.
Model Prediction - 0.0882353
Actual Answer - 0.0882353 (3/34)

Average performance:
Question - What is prob of picking 1 b and 1 p when two letters picked without replacement from tpppbbpbbb?
Model Prediction - 0.4448
Actual Answer - 0.4444 (4/9)

Poor performance:
Question - Four letters picked without replacement from {y: 6, b: 1, u: 4}. What is prob of picking 2 y and 2 u?
Model Prediction - 0.1329
Actual Answer - 0.2727 (3/11)

Data Source

To implement this model, we used the DeepMind dataset from Github. More specifically, we used the "probability__swr_p_level_set.txt" dataset, which is further explained below.

Data Summary

Citation: https://identifiers.org/arxiv:1904.01557 (Free to use under Apache License 2.0)

This dataset contains 666,666 samples (in the format we desire) which was more than sufficient for our model to understand sequences of words, learn probability, and produce a numerical output as an answer to the question.

The labels in the dataset are simple to process to numerical values which simplifies the problem from a sequence to sequence question to a value prediction question. Moreover, since the dataset is pre-generated by an algorithm it is unlikely to have numerical mistakes in the final answers.

All 666,666 samples follow the same format: 2 lines per data point with the first being the question and the second being the solution.

Example:

- Question 1:
    - What is prob of picking 1 b and 1 p when two letters picked without replacement from tpppbbpbbb?
    - Solution: 4/9
- Question 2:
    - Four letters picked without replacement from {y: 6, b: 1, u: 4}. What is prob of picking 2 y and 2 u?
    - Solution: 3/11

For more information on the dataset, here are some statistics on the samples and nature of data:

- Average Input Length (# of words): 19.788
- Average Output Value: 0.163

During training, we found that using our entire dataset required large amounts of time and as such, we only used 10% of it (66,667 samples). Here is the information about the specific points we used:

- Average Input Length (# of words): 19.777
- Average Output Value: 0.162

As seen, our smaller set is comparable to the entire set as it outputs almost equal average values. Moreover, we ran the same code on the specific training, validation, and testing set:
Training Set:

- Average Input Length (# of words): 19.765
- Average Output Value: 0.162
- Average Input Length (after tokenization): 29.765
- Max Input Length (after tokenization): 45

Validation Set:

- Average Input Length (# of words): 19.77
- Average Output Value: 0.16
- Average Input Length (after tokenization): 29.773
- Max Input Length (after tokenization): 45

Testing Set:

- Average Input Length (# of words): 19.814
- Average Output Value: 0.163
- Most Common Output Value: 0.1
- Average Input Length (after tokenization): 29.704
- Max Input Length (after tokenization): 45

*Tokenization explained in the Data Transformation section.

Data Split

A straightforward 60% (training), 20% (validation), 20% (testing) data split was used. The dataset used was large and already randomized meaning that the question types were split evenly amongst the dataset and not grouped together. Thus, given that similar format of questions appeared frequently, the data split included random samples in all sets meaning all sets were representative of our data.

Data Transformation

Initially, the data was in the format of a large text file where each 1-line question was followed by a 1-line answer. First, we split the data into the train, test and validation sets as explained above. We converted the fractional answers to numerical ones in this step while adding data points to the respective sets. Next, we converted the sets into json files with a question solution pairing. Given the large size of our data, we also created smaller datasets that could be used for error detection, simple training and tuning and would require less time while still being representative of our entire dataset.

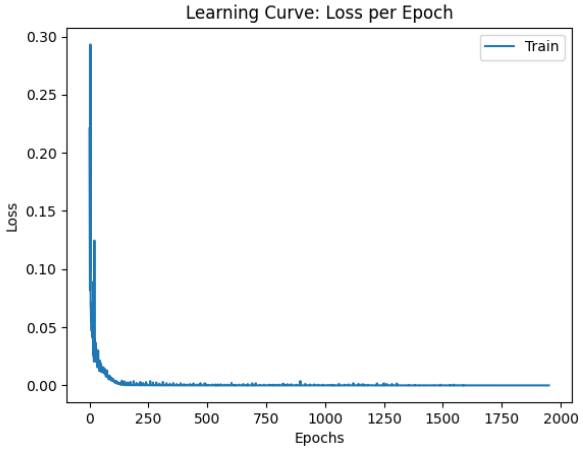
Then, we loaded the json files into our Google Colab notebook and tokenized the data with GloVe indices through TabularDatasets in PyTorch. Our parse_data function acted as the

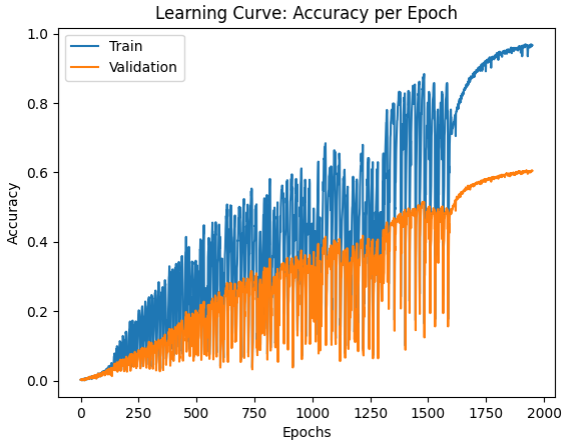
tokenizer in this case and more specifically, it added tokens based on unseen words in the dataset. For instance, a question contained the following: “without replacement from tpppbbpbbb” where ‘tpppbbpbbb’ does not appear in the GloVe dataset and as such was split into separate letters and then tokenized. We built a vocabulary based on our questions dataset and then formed BucketIterators to be used to train the model with batches.

Training Curve

The training curves are based on a tolerance of 1e-3 (essentially, checking that our model predicts values that are within 3 decimal places of the answer). We have tried the model on different tolerance thresholds, however, we saw it most fit to include the loss and accuracy of the model with 1e-3 threshold as justified in the quantitative measure section.

We use the MSE loss function as it is the most appropriate to find differences between the target numerical values and predicted numerical values.





Note also that since we are dealing with such a low loss, there may be floating point precision issues. The loss is around 1e-7 after epoch 250 and reaches 1e-8 after approximately 1500 epochs.

Next, we look at the training and validation accuracies of the model. Throughout training, we did introduce learning rate decay and change in dropout as identified in the next table. We can see that the training set accuracy is 96.695% whereas the validation set plateaus at around 60.508%.

Epochs	Learning Rate	Dropout
0-750	1e-4	0
750-1250	1e-5	0

1250-1500	1e-6	0
1500-1950	1e-7	0.1

Hyperparameter Tuning

The first thing we tried doing was overfitting our model to a smaller dataset of 40 samples (question-solution pairs). This was done to see if our model had any noticeable bugs and to get a feel of how the basic hyperparameters like learning rate would impact the model and training time. This taught us that a learning rate of 1e-3 or 1e-4 was pretty good for the early stages of training as it learned quickly while not having a super jagged curve. However, as our model got better we knew we wouldn't be able to use this learning rate as it would lead to too much variance.

We had many parameters we could play with including number of layers in the encoder/decoder, number of hidden units in the encoder/decoder, learning rate, dropout, weight decay, and batch size.
As the model took a long time to learn (our final model took 20 hours to train) we had to be smart in the way we tested different hyperparameters.

We started by making the encoder have 6 layers and 500 hidden units per layer. This seemed like good values given the complexity of our input and what we had seen other people do for different types of applications online. We gave our decoder 4 layers with each mapping an input of size 300 to an output of size 300 except for the last one which mapped to an output of size 1. We ran this with a learning rate of 1e-4 and batch size 32 and quickly found that this model was not complex enough as it wouldn't learn so we decided to increase the number of layers by 1 and also increased the size of the layers. We did this by increasing the size of the decoder layers and testing the model several times and ended up with:
Model_1:
- Encoder: 300 x 500 x 6 x 3
- Decoder:
  - Layer1: 300 x 1500
  - Layer2: 1500 x 1000
  - Layer3: 1000 x 600
  - Layer4: 600 x 300
  - Layer5: 300 x 1

We then ran our model again and it did learn but it took a while to run. This is where we started to play with the batch size going from 32 all the way to 256 as that was the maximum the GPU's VRAM would allow. We stuck to 256 as that led to the fastest training due to parallelization.

We then ran the model for a bit but it ended up getting stuck at 55% validation accuracy. Once again we increased the decoder complexity.

Model_2:

- Encoder: 300 x 1024 x 6 x 3
- Decoder:
  - Layer1: 300 x 4096
  - Layer2: 4096 x 2048
  - Layer3: 2048 x 2048
  - Layer4: 2048 x 1024
  - Layer5: 1024 x 1024
  - Layer6: 1024 x 512
  - Layer7: 512 x 512
  - Layer8: 512 x 1

This model had considerably more parameters and so we decided not to increase the encoder complexity much as the model would take way too long to train. We thought using dropout initially would complicate the training and so decided to train the model without dropout and to add it at later epochs by the use of checkpointing.

We decided to train this model with batch size 256 and learning rate 1e-4 as that would lead to less noise than 1e-3. We let it run for 750 epochs where it started to get noisy. For this reason we used the checkpoint and trained the model again but with learning rate 1e-5. This went well for 500 epochs where the model got noisy again so we dropped the learning rate to 1e-6. This went for another 250 epochs. From here we wanted our model to generalize well and not over rely on some information so we started testing dropout and batch normalization.

We quickly ran some tests on dropout vs batch normalization on a smaller set. We found that batch normalization wasn't even able to overfit the data so we were stuck with dropout.

Furthermore we tried 50% dropout but our model started predicting gibberish so we dropped dropout until 10% which gave more stable predictions and would still allow the model to generalize. As such we ran another 450 epochs with learning rate 1e-7 and dropout 10% where we saw the curve start to flatten. This model was at 96.695% training accuracy and 60.508% validation accuracy which was pretty good! Since we had already tuned our model for multiple days, the curve was pretty flat, and we were happy with the accuracy, so we decided this was a good place to stop.

Another thing we had tried was using a positional encoder so that the position of the words in the sentence would be kept but found that this negatively impacted accuracy. While training we were able to get training accuracy to increase but validation accuracy would stay flat which is a sign of

overfitting. When solving these questions we only care about the variables and numbers and not the words so using a positional encoder adds noise and affects the prediction. For example, “what is the prob of picking 1 a from aaaabbbba” and “what is the prob of picking 1 a and 1 b from aaaabbbba” both have a similar format but adding positional encoding makes the questions different, thus our neural network struggles to generalize.

Quantitative Measures

Our get_accuracy function is the reference point for this section. We check that the model predictions are within 1e-3 (accurate to 3 decimal places) of the final answer (which is the numeric label evaluated from the fraction). The 1e-3 tolerance threshold was seen as the most fit since the intended use for high-school students generally requires answers to the nearest 2 or 3 decimal places.

The loss metric used was MSE as it was the most reasonable choice to calculate the difference between two numerical values (solution to problem and our prediction). We don’t use cross-entropy loss as our problem is a regression problem and not a classification problem. We want to lower the distance between the ground truth value and our model’s output.

Additionally, given the time and resource constraints (in terms of GPU RAM), it was harder to increase performance to the same levels for 1e-4 and lower tolerance levels.

Quantitative and Qualitative Results

The model had an accuracy of 60.97% on the test set. We also tested the model with difference tolerance levels on the test set and the results are stated below.

Tolerance	Accuracy
1e-2	87.544
1e-3	**60.967**
5e-4	49.104
1e-4	17.802

The test performance was identical across small and large datasets as the format of questions was similar with different values.

As an extreme analysis, we compared the performance of our model to a baseline model that would predict the most common answer (or the mode answer which was 0.1) for each question as identified in the data summary. The baseline model has an accuracy of 2.857% on the test set (with tolerance 1e-3), significantly lower than our model.

Upon further analysis of the results, we found that the model performed well with shorter lengths of tokenized inputs. The best 100 predictions were made for an average length of 19.78 (approximately 10 lower than average) whilst the worst 100 predictions were on an average length of 35.96 after tokenization. Additionally, the length of tokenized sentences directly correlates with the size of the probability pool. For instance, probability of 1d and 1u from "vzmnudmnz" (size: 9) had a difference of 5.79e-7 from the correct answer whereas probability of 2s from "ssssssszssssss" (size: 13) was off by 0.006. This is logical given the nature of the problem; increasing the number of variables and items within a probability question makes the problem difficult to solve as there are generally more possibilities to consider.

Justification of Results

Given the difficulty of the problem where the model essentially had to learn probability functions, the 60.967% test accuracy is reasonable for the scope of the project. Additionally, we were advised to try a simpler problem that concerned arithmetic instead but due to our interest in the problem, we decided to go ahead with probability.

To improve the performance of the model even further, we would need to increase the model's complexity very significantly through more encoder and decoder layers with higher dimensions. However, given time and resource constraints, this seemed like an unachievable task. Additionally, with enough computational resources, we would also be able to use the full dataset rather than just 10% of it. With these improvements, we hypothesize the test accuracy would be improved.

To justify our results we checked the accuracy had we always predicted the mode of the outputs which turned out to be 1/10.

Tolerance	Accuracy
1e-1	70.06570065700657
1e-2	7.438574385743857
1e-3	2.8575285752857527
1e-4	2.812528125281253

As seen, with a tolerance of 1e-3 predicting the mode gives an accuracy of under 3%. This means our model did approximately 21 times better.

Given that the dataset is a popular mathematics dataset, previous work has been conducted on a similar problem. As shown, ML experts from Facebook and Google were able to devise a transformer model for a sequence to sequence prediction (they predicted fractions whereas we predicted numerical values) that had 78% test accuracy. However, our transformer model was able to beat their other approaches such as the simple and attentional LSTMs.

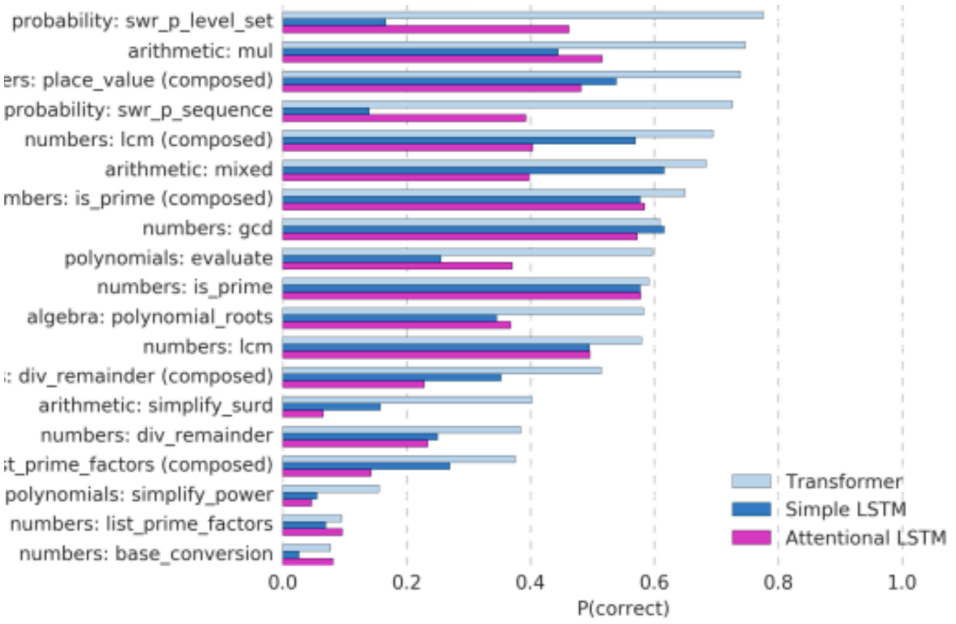


Figure 4: **Interpolation** test performance on the different modules.

For the use case, high-schools most commonly require students to round answers to 2 decimal places. With this threshold, our model has 87.544% accuracy. Essentially, students using our model for their intermediary steps in assignments can expect a grade of A, which we believe to be satisfying and reasonable.

Ethical Considerations

Given our predictive model will aim to solve and give direct answers to probability questions, it can be exploited by high-school students to cheat on assignments and gain an “unfair advantage” over their peers. The model is intended to be used as a tool for intermediary computations of probability by high-school and university students who already have access to online tools.

However, if we consider the opposite end of the spectrum - teachers or professors, our model

will benefit them by saving them time and could even be used for autograding purposes where the teachers don't need to find answers for their questions.

Additionally, we aim to mitigate the "cheating" risks associated with the model by predicting numerical values instead of fractional values as expected by probability assignments. However, this can also be seen as a limitation to the model since it does not provide exact values in the form of fractions and rather, provides numerical values rounded off to 3 decimal places.

Authors

The work was broken down evenly and the majority was done in 3 meetings involving all 3 members. The first meeting was designed for project plan and initial discussions regarding approaches and data sources. The second meeting was set up solely for data collection and processing, wherein all 3 members took a task each: formatting data into json for easy access, parsing data with GloVe embeddings and splitting data into train, validation and test sets. In the last 3-4 hour long meeting, everyone was working on the transformer model. Training and get_accuracy functions were split evenly and discussions on model parameters and layers were done together. Everyone was constantly involved with debugging and overfitting to a small dataset.

Ugo Dos Reis: contribution to all meetings, debugging, writing model and introduction sections in report
Jiachen Sun: contribution to all meetings, debugging, hyperparameter tuning (training on large dataset)
Ayush Oza: contribution to all meetings, debugging, writing data and results section in report

Appendix

Modules	Parameters
transformer_encoder.layers.0.self_attn.in_proj_weight	270000
transformer_encoder.layers.0.self_attn.in_proj_bias	900
transformer_encoder.layers.0.self_attn.out_proj.weight	90000
transformer_encoder.layers.0.self_attn.out_proj.bias	300
transformer_encoder.layers.0.linear1.weight	307200
transformer_encoder.layers.0.linear1.bias	1024
transformer_encoder.layers.0.linear2.weight	307200
transformer_encoder.layers.0.linear2.bias	300
transformer_encoder.layers.0.norm1.weight	300
transformer_encoder.layers.0.norm1.bias	300
transformer_encoder.layers.0.norm2.weight	300

transformer_encoder.layers.0.norm2.bias	300
transformer_encoder.layers.1.self_attn.in_proj_weight	270000
transformer_encoder.layers.1.self_attn.in_proj_bias	900
transformer_encoder.layers.1.self_attn.out_proj.weight	90000
transformer_encoder.layers.1.self_attn.out_proj.bias	300
transformer_encoder.layers.1.linear1.weight	307200
transformer_encoder.layers.1.linear1.bias	1024
transformer_encoder.layers.1.linear2.weight	307200
transformer_encoder.layers.1.linear2.bias	300
transformer_encoder.layers.1.norm1.weight	300
transformer_encoder.layers.1.norm1.bias	300
transformer_encoder.layers.1.norm2.weight	300
transformer_encoder.layers.1.norm2.bias	300
transformer_encoder.layers.2.self_attn.in_proj_weight	270000
transformer_encoder.layers.2.self_attn.in_proj_bias	900
transformer_encoder.layers.2.self_attn.out_proj.weight	90000
transformer_encoder.layers.2.self_attn.out_proj.bias	300
transformer_encoder.layers.2.linear1.weight	307200
transformer_encoder.layers.2.linear1.bias	1024
transformer_encoder.layers.2.linear2.weight	307200
transformer_encoder.layers.2.linear2.bias	300
transformer_encoder.layers.2.norm1.weight	300
transformer_encoder.layers.2.norm1.bias	300
transformer_encoder.layers.2.norm2.weight	300
transformer_encoder.layers.2.norm2.bias	300
transformer_encoder.layers.3.self_attn.in_proj_weight	270000
transformer_encoder.layers.3.self_attn.in_proj_bias	900
transformer_encoder.layers.3.self_attn.out_proj.weight	90000
transformer_encoder.layers.3.self_attn.out_proj.bias	300
transformer_encoder.layers.3.linear1.weight	307200
transformer_encoder.layers.3.linear1.bias	1024
transformer_encoder.layers.3.linear2.weight	307200
transformer_encoder.layers.3.linear2.bias	300
transformer_encoder.layers.3.norm1.weight	300
transformer_encoder.layers.3.norm1.bias	300
transformer_encoder.layers.3.norm2.weight	300
transformer_encoder.layers.3.norm2.bias	300
transformer_encoder.layers.4.self_attn.in_proj_weight	270000
transformer_encoder.layers.4.self_attn.in_proj_bias	900
transformer_encoder.layers.4.self_attn.out_proj.weight	90000
transformer_encoder.layers.4.self_attn.out_proj.bias	300
transformer_encoder.layers.4.linear1.weight	307200
transformer_encoder.layers.4.linear1.bias	1024
transformer_encoder.layers.4.linear2.weight	307200
transformer_encoder.layers.4.linear2.bias	300

transformer_encoder.layers.4.norm1.weight	300
transformer_encoder.layers.4.norm1.bias	300
transformer_encoder.layers.4.norm2.weight	300
transformer_encoder.layers.4.norm2.bias	300
transformer_encoder.layers.5.self_attn.in_proj_weight	270000
transformer_encoder.layers.5.self_attn.in_proj_bias	900
transformer_encoder.layers.5.self_attn.out_proj.weight	90000
transformer_encoder.layers.5.self_attn.out_proj.bias	300
transformer_encoder.layers.5.linear1.weight	307200
transformer_encoder.layers.5.linear1.bias	1024
transformer_encoder.layers.5.linear2.weight	307200
transformer_encoder.layers.5.linear2.bias	300
transformer_encoder.layers.5.norm1.weight	300
transformer_encoder.layers.5.norm1.bias	300
transformer_encoder.layers.5.norm2.weight	300
transformer_encoder.layers.5.norm2.bias	300
encoder.weight	117000
decoder.weight	1228800
decoder.bias	4096
layer1.weight	16777216
layer1.bias	4096
layer2.weight	8388608
layer2.bias	2048
layer3.weight	4194304
layer3.bias	2048
layer4.weight	2097152
layer4.bias	1024
layer5.weight	1048576
layer5.bias	1024
layer6.weight	524288
layer6.bias	512
layer7.weight	262144
layer7.bias	512
layer8.weight	512
layer8.bias	1

Total Trainable Params: 40522705