Deep Learning Project Report

Luigi Antonelli - Florin Cuconasu - Luca Gaudenzi

February 2023

1 Dataset

The analysis was conducted on the **Mathematics Dataset** [1], which consists of many different types of mathematics problems, also organized by difficulty. The dataset is constituted by question and answer pairs that can be generated through a script by the authors; nonetheless, we decided to use the same pairs used in the original paper [1]. Thus, the dataset is structured into 56 modules, which cover different categories of mathematical problems, each of 2 million training samples. Due to the high number of samples (112 million considering all the 56 modules) and our limited computational power, we decided to focus only on 3 different modules: *algebra__linear_1d*, *probability__swr_p_level_set*, *numbers__is_prime*. The choice was done by considering the modules where the results of the TP-architecture [2] were of different scale:

- *algebra__linear_1d* with accuracy $> 95\%$
- *probability__swr_p_level_set* with accuracy in range $[75\%, 85\%]$
- *numbers__is_prime* with accuracy in range $[55\%, 60\%]$

Therefore, the dataset we used contains 6 million training samples and 30 thousand test samples. We used the test data of the *interpolation* directory, containing the same type of problems of the training. We did not test the model on the *extrapolation* data because we did not train the model on all the different mathematical tasks; whereas, as stated in the original paper, this test set was introduced to measure the ability of the model to algebraically generalize on much more difficult problems.

1.1 Implementation Details

Each sentence is analysed as a sequence of characters, hence the vocabulary is composed by character symbols. (Saxton et al., [1]) reported a vocabulary size of 95, as they considered lowercase and uppercase characters and also symbols that never appear in the train and test data. Instead, we opted to reduce the vocabulary to only lowercase characters and the symbols that only appear in the train and test data (using the function `get_vocabulary_from_files`). However, we also added the special token $< unk >$, i.e., unknown, to manage the cases in which during inference

the model may receive out-of-vocabulary tokens. Our final vocabulary size is of 58 symbols, also including the special tokens ($< bos >$, $< eos >$, $< pad >$, $< unk >$).

For what concerns the `Mathematics_Dataset` class, we decided to pre-process questions and answers on the fly when they are accessed through the `__getitem__` method. We noticed that storing sentences that had already been pre-processed and then loaded when instantiating the dataset object was not doable for the Colab RAM, as it required more than 40 GB.

We tested the performance of the `__getitem__` method on a smaller batch of data, comparing our approach and the one loading the data already pre-processed. The difference is not so significant since our approach requires about 200 - 300 μs and the other 40 - 60 μs. Moreover, we tried to efficiently manage the dataset loading using multiple workers (setting the parameter `num_workers = 2` of the `DataLoader`) and pinning the memory (`pin_memory=True`) to minimize the data transfer between CPU and GPU.

2 NON-SOTA

2.1 Transformer:

The first baseline that we implemented is the vanilla Transformer from "Attention Is All You Need" [3]. In our code it's the `pl.LightningModule` class `Transformer`. The architecture is composed by an encoder and a decoder. The encoder takes in input the embeddings X of the input sequence (trainable embeddings `nn.Embedding`) to which we add the positional embeddings.
Positional embeddings are necessary because, unlike RNNs, Transformers take in input the whole sequence at once and the Multi-Head attention layer is permutation equivariant: $P\,\mathrm{MHA}(X) = \mathrm{MHA}(PX)$, where P is a permutation matrix. With positional embeddings E instead $P\,\mathrm{MHA}(X + E) \neq \mathrm{MHA}(P(X + E))$. In other words, without positional embeddings Transformers would perceive the input sequence as a set of (unordered) tokens and only with these embeddings they can understand ordering. In this case we are using the sinusoidal positional embeddings proposed in "Attention Is All You Need". There are also other variants, for example learnable positional embeddings, that can be concatenated to the input so that $P\,\mathrm{MHA}(X||E) \neq \mathrm{MHA}(P(X||E))$.
The modified input is then passed through a series of encoder blocks
(`TransformerBlock` in our code) composed by a MHA layer, layer normalization, a feed forward network and another layer normalization. There are also two residual connections right before each layer normalization (Add&Norm). In the encoder (`TransformerEncoder`), each MHA uses the same input for query, key and value and a padding mask to let the attention focus only on tokens different from the padding token that is used to make every sequence of the same length. In practice, this mechanism consists in setting the unnormalized attention scores for padding tokens to $-\infty$ so that their normalized attention scores (after the softmax) will be zero. We took inspiration from "The Annotated Transformer" [4] to implement the Multi-Head Attention layer in an efficient way. In fact, instead of using H (number of heads) self-attention layers, we used a single `nn.Linear(embedding_dim,`

`embedding_dim)` respectively for W_q, W_k and W_v. This is possible as long as the size of the embeddings is a multiple of H (`embedding_dim = num_heads * dim_head`) and `dim_head` $= d_q = d_k = d_v$ so that query, key and value all have the same dimension (d_q must be equal to d_k by default, otherwise we couldn't compute the scaled dot product attention).
The decoder (`TransformerDecoder`) instead is composed by a series of decoder blocks `DecoderBlock` composed by a masked MHA, a layer normalization after a residual connection and a `TransformerBlock`. During training Transformers use teacher forcing which consists in passing to the decoder the whole output sequence shifted to right Y (embeddings + positional embeddings of the output shifted to right) so that the decoder can learn from the correct label at each time step instead of relying on the prediction it made for the previous time step. This practice combined with the fact that Transformers can handle entire sequences in a single forward pass requires a causal mask: the prediction of the $(i+1)$-th token, made for the subsequence containing the first i tokens of the output, can only depend on every token up to the i-th and not on future tokens $j > i$. Similarly to the padding mask, the causal mask makes the normalized attention scores that refer to forbidden tokens equal to zero. This is the reason why the first MHA in a `DecoderBlock`, where Y = query = key = value, uses a causal mask combined with a padding mask (in the function `create_trg_mask`). The second MHA in a decoder block (included in the `TransformerBlock`) uses as query the result of layer normalization right after the masked MHA and as key and value the output of the encoder. This layer allows to attend to the memory created by the encoder to answer the queries coming from the output sequence. The residual connection after the second MHA adds the query back to the attention value. The decoder blocks end with a feed forward network and another Add&Norm just like an encoder block.
After the `TransformerDecoder` there is a final linear layer for classification over |vocabulary| tokens because the Transformer predicts the next word of the output.
During inference of course teacher forcing is not used and the decoder is queried a number of times equal to the length of the predicted output sequence: we start with $< bos >$ (beginning of sequence token) and compute the argmax over the logits of the final linear layer to understand which token in the vocabulary is the most likely to be the next one after $< bos >$. We repeat the same operation by using at each iteration an output sequence longer by one token until the $< eos >$ (end of sequence) token is predicted or the maximum length for the output is reached.
As a form of regularization, we are using a Dropout layer

- on the normalized attention scores,
- after each Multi-Head attention,
- after the non-linearities in the feed forward networks,
- after each feed forward network.

2.2 Seq2Seq with GRU:

The second baseline that we implemented is GRU for solving sequence to sequence tasks. In our code it's the `pl.LightningModule` class `GRUEncoderDecoder`. As for the Transformer baseline, the architecture is composed by an encoder and a decoder, with the first taking in input the embeddings X of the input sequence (computed with `nn.Embedding`). Unlike Transformer, positional embeddings and causal masks are not required because RNNs process sequences token by token and update an hidden state to remember previous information.
X is then given in input to the encoder (class `GRU` in our code). The encoder is a multi-layer GRU RNN, and it is very similar (in the way in which it is called and in what it returns) to PyTorch GRU implementation (`torch.nn.GRU`). The class `GRU` allows to specify a mask for each input sequence; this mask is used to block the update of the `hidden_state` at timestamp `t` when the `t`-th element of the tensor `mask` is false. The reason of this functionality is that we are forced to pad the input sequences in order to do the computation in batches: RNN allows to consider variable length sequences, but sequences of the same batch must have the same length and it is likely to have input sequences containing more padding than meaningful tokens. Therefore the padding could significantly affect the computation of the last hidden state of the encoder, and consequently the computation in the decoder. In order to mitigate this problem the `mask` is used to block the GRU from updating the `hidden_state` by using padding information.
Our implementation of multi-layer GRU RNN also consists of a class `GRUCell`. As can be inferred from the name, this class implements a GRU cell; it is worth noting that, unlike the PyTorch implementation, the update of the hidden state is computed differently. In both implementations (ours and the PyTorch one), we have that:

$$h_t = (1 - z_t) \cdot n_t + z_t \cdot h_{t-1},$$

but whereas in ours:

$$n_t = \tanh\left(W_{in}x_t + b_{in} + W_{hn}(r_t \cdot h_{t-1}) + b_{hn}\right),$$

in PyTorch GRU:

$$n_t = \tanh\left(W_{in}x_t + b_{in} + r_t(W_{hn} \cdot h_{t-1} + b_{hn})\right).$$

Also the decoder is a multi-layer GRU RNN, hence encoder and decoder share the same class (class `GRU`). In the encoder the initial hidden state of each layer is initialized as a tensor of zeros, conversely the initial hidden state of the decoder is equal to the last hidden state of the encoder. The component of the last hidden state associated to the last layer of the encoder's GRU is concatenated to the embedding of the token used by the decoder to predict the next one. For this last we were inspired by **Simple Seq2Seq machine translation using GRU based encoder-decoder architecture** [5].
Unlike Transformers, we have adopted probabilistic teacher forcing for training GRU. Similarly to the deterministic teacher forcing, the probabilistic one consists

in passing to the decoder the whole output sequence shifted to right; additionally the probabilistic version does not use always the correct label in order to predict the next token, conversely with a certain probability p (for our training we decided to use $p = 0.50$) it uses the token generated at the previous timestamp (so that it simulates what happens at inference time where the whole output sequence is not available).
One more difference from Transformer baseline is that after the decoder we don't use a single linear layer for classification over |vocabulary| tokens; conversely we apply a feed forward network composed by two linear layers with a ReLU activation function in the middle.
At inference time teacher forcing is not used, therefore to predict the next token the decoder uses only already predicted tokens (the first token is always $< bos >$). As for Transformer, the decoder is queried until either each sequence of the batch has reached $< eos >$ or the maximum length for the output is reached. As a form of regularization, we are using a Dropout layer:

- after the ReLU of the final feed forward network
- to the input (but not to the input hidden state) of each GRU cell
- to the output (but not to the final hidden state) of the multi-layer GRU RNN

The last two Dropout layer applications are inspired by "**Recurrent Neural Network Regularization**" [6].

2.3 Seq2Seq with LSTM:

The third baseline that we have implemented is LSTM for solving sequence to sequence tasks. In our code it's the `pl.LightningModule` class `LSTMEncoderDecoder`. This baseline has a lot of in common with **Seq2Seq with GRU**:

- the final feed forward network (with associated dropout)
- the concatenation of the last hidden state associated to the last layer with the output tokens' embeddings
- the use of probabilistic teacher forcing

Some of the differences are:

- encoder and decoder are multi-layer LSTM RNN and not GRU RNN
- use of PyTorch LSTM (not implemented from scratch)
- absence of the mask functionality used in the GRU baseline

In general the code used for LSTM and GRU is more or less the same, therefore we don't explain it again.
As a form or regularization, we use:

- as in the GRU baseline, a Dropout layer after the ReLU of the final feed forward network
- the dropout parameter of PyTorch LSTM class.

3 SOTA: TP-Transformer

We implemented the TP-Transformer (Tensor product Transformer) from the paper **"Enhancing the Transformer With Explicit Relational Encoding for Math Problem Solving"** [2] which proposes a slight modification of the original Transformer architecture from "Attention Is All You Need" [3].
The paper underlines how a Transformer can be considered as a kind of Graph neural network: when we issue a query for the i-th token in the sequence and compute the attention on other tokens it's like we are building an information-flow graph where there is a directed edge e_{ji} from the j-th token to the i-th token having a weight equal to the normalized attention score a_{ij} and label h, where $h \in \{1, \ldots, H\}$ and H is the number of heads.
The paper introduces a new affine transformation to substitute these discrete labels $\{1, \ldots, H\}$ of the edges with a relation vector built starting from the query. The attention value is called "filler" and the relation vector is called "role" in the sense that the latter embeds the structural role that the former "fills". The attention value and the relation vector are multiplied element-wise before performing a final affine transformation like in the standard Multi-Head Attention. Therefore the output of the TP-Multi-Head Attention layer is a contraction (the diagonal) of the tensor product of a structure with H constituents.
Our implementation in PyTorch Lightning of this model shares the underlying architecture of the vanilla `Transformer` but allows to create a TP-Transformer by passing as parameter `tp_attention = True`. There is a slight difference between our implementation and the one proposed in the paper: the authors of the paper consider Transformers with an additional layer normalization at the beginning of each Transformer block (both in the encoder and in the decoder). We decided to skip this operation because in the paper it is considered external to the TP-Multi-Head Attention layer and because "Attention Is All You Need" didn't use it. Moreover, we did a further check on the official repository [7] and even their implementation of the vanilla Transformer included the additional layer normalization.

4 Results

The results that are showed in this section were computed by tracking the metrics via TensorBoard. In particular, we tracked the training loss and the accuracy of both test and training sets. In this way, we could understand if the model started overfitting, hence when the training accuracy grew much faster than the test accuracy.

Since the dataset is quite big (about 6M samples), even though we have reduced it a lot by considering only 3 modules, we could not train the model using hyper-parameters comparable to the one used in the paper [2]. The main differences with respect to the hyper-parameters used in the paper are: a small *number of encoder and decoder blocks* for the Transformers, i.e, 3 blocks instead of 6; *hidden size* of 512 instead of 2048; *batch size* of 128 instead of 1024; a much small number of training steps, i.e., 350k of the TP Transformer against 1M. These choices were not only enforced by the fact that Colab does not have a sufficient RAM and GRAM for

managing those quantities, yet mainly for the reason that our model was trained on a much smaller dataset (6M vs 112M), so those quantities might not have been a good fit for our architecture. The hyper-parameters of the best models can be seen in Table 1. We noticed that a good impact on the model's performance was the *hidden size*, hence its increase had a good impact on the test accuracy. Instead, for what concerns the dropout value we found that it did not impact so much the accuracy, nevertheless, for the Vanilla Transformer a little percentage of dropout (0.3) improved the score. The optimizer we used was Adam, with a learning rate = $1 \cdot 10^{-4}$, beta1 = 0.9, beta2 = 0.995 as reported in the paper [2]; even though we tried other configurations as well, those remained the best.

We decided not to train the GRU model, as after some tests we noticed that it required too much time, i.e., about 7 hours for a single epoch. The reason may be that our implementation is not so optimized, especially due to the loop over the input tokens. We looked at the PyTorch implementation and in fact it uses a TorchScript script for that part, also optimized in C++. Therefore, for what concerns the RNNs we will solely present the results of the LSTM.

4.1 Comparisons

In Figures 1 and 2 we can see the performance of the two Transformers, where the yellow line refers to the TP version and the blue one to the Vanilla. We limited the graphs to the same portion of steps, so the two trends can be compared. As we can observe the TP performed better than the Vanilla. Indeed, in Figure 1 the training loss of the TP decreased faster from the beginning without overfitting too much the training set: the training accuracy at the end of the third epoch of the Vanilla Transformer was 57.5%, instead the TP one was 62.1%, but respectively with 53.7% and 56.1% of test accuracy. Moreover, in support of this we can look at the test accuracy trend (Figure 2). The sort of oscillating trend is due to the type of testing samples that are inferred in that particular step. For instance, when the model is inferring samples from the *algebra__linear_1d* module we can notice some "hills", since the Transformer models are relatively good at that kind of questions, especially when they are of type *easy*. On the contrary, the LSTM model is not that good at that type of questions, as we can see in Figure 3, where we have "reverse hills" in those portions. However, the LSTM model seems to be better on the other two modules, in particular on the *probability__swr_p_level_set* module where it reached the best accuracy, also surpassing the TP Transformer (Table 2).

An important observation is that all the models performed overall comparable to the TP model of the paper [2] on the *numbers__is_prime* module, meaning that it is quite a hard task. Indeed, the answers to this type of questions are simply boolean answers, i.e., if the number is prime or not, thus the models somewhat understand some patterns but it is still not enough for these kinds of problems.

Hyper-parameter	TP	Vanilla	LSTM
Epochs	7	3	4
Batch size	128	128	128
Num heads	8	8	-
Embedding dim	256	256	128
Hidden size	512	512	128
Dropout	0.0	0.3	0.0
Gradient Clipping	0.5	0.5	0.5
Teacher Forcing Ratio	-	-	0.5
Num Blocks	3	3	-
Num Cells	-	-	2

Table 1: Hyper-parameters of the best models. TP refers to the TP-Transformer and Vanilla to the Vanilla Transformer.

Model	Steps	ACC	Algebra	Probability	Prime Numbers
LSTM	185k	49.8%	38.5%	**51.6%**	54.9%
Vanilla Transformer	140k	53.7%	58.3%	49.8%	54.9%
TP Transformer	350k	**59.7%**	**75.1%**	50.7%	**59.3%**

Table 2: Accuracy scores on test set. Models were trained on a Colab Tesla T4 GPU.

5 Conclusions

On the basis of our experiments and of the mathematical modules that we considered, the new proposed Transformer architecture performs slightly better overall than the Vanilla architecture. Nevertheless, we suppose than with more training of the Vanilla Transformer and maybe with more parameters, it could reach similar performance to the TP Transformers. Instead, the LSTM model suffers a bit from the absence of the attention mechanism, especially in the *algebra__linear_1d* module, where it may help to understand that the same variables in two different sides of an equation actually refer to the same variable. Apart from this, our LSTM performs similarly to what reported in paper [1] in the other two modules.

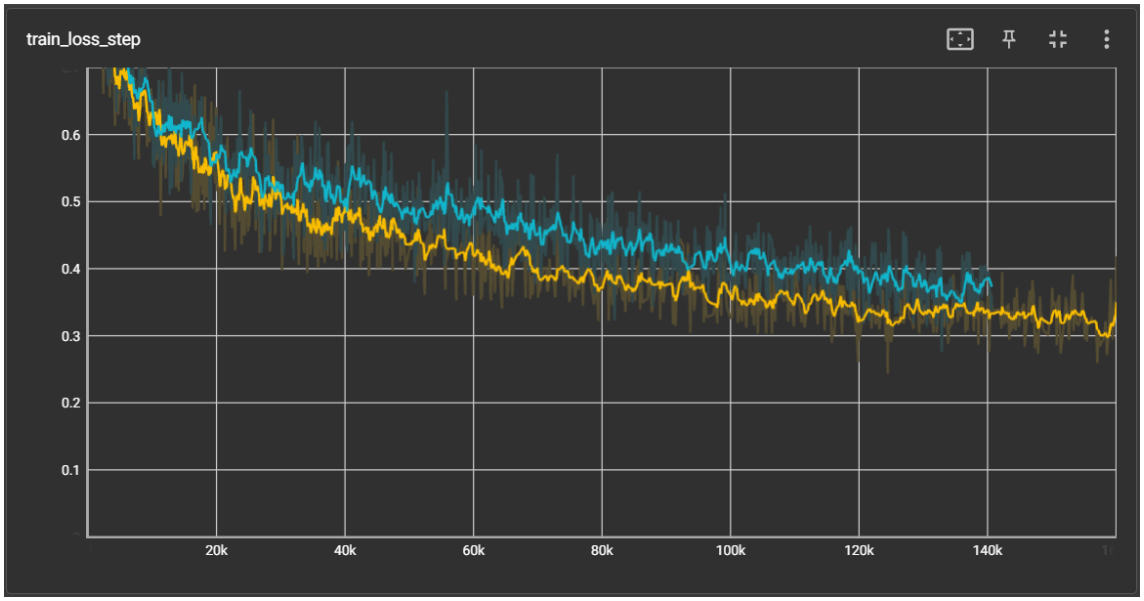

Figure 1: Training loss comparison on the first 3 epochs TP (yellow) vs Vanilla (Blue) Transformers.

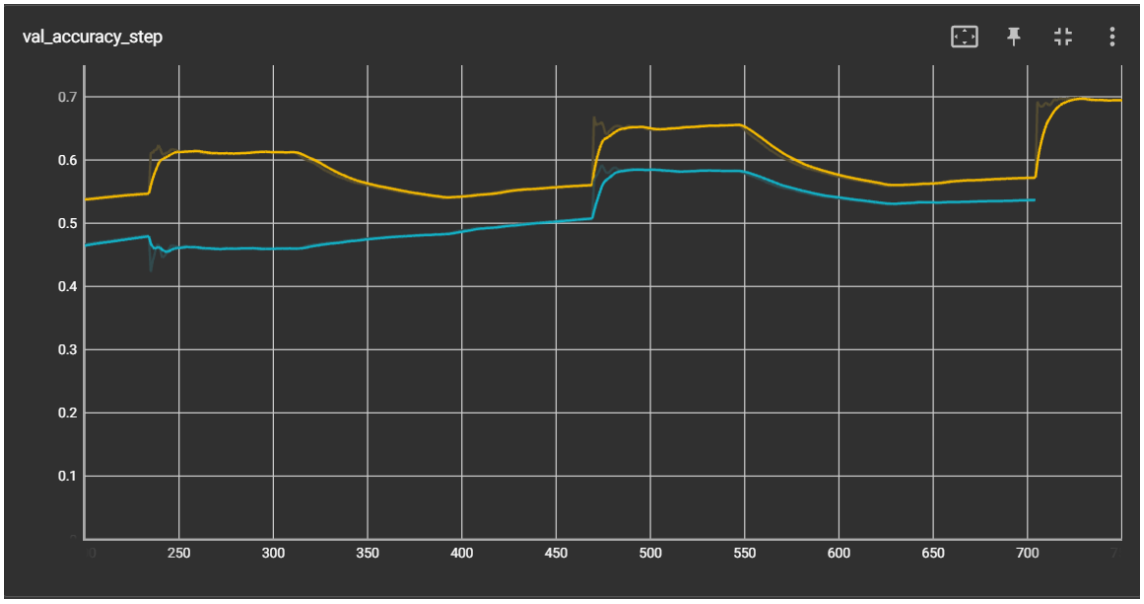

Figure 2: Test accuracy comparison on test steps TP (yellow) vs Vanilla (Blue) Transformers.

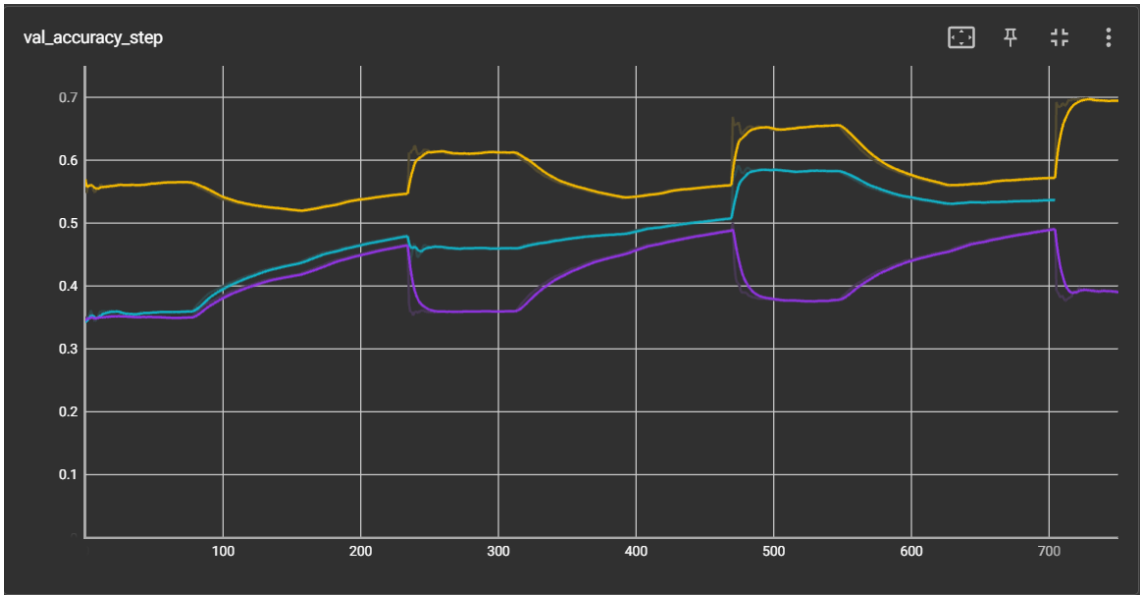

Figure 3: Test accuracy comparison on test steps TP (yellow) vs Vanilla (Blue) Transformers vs LSTM (Purple).

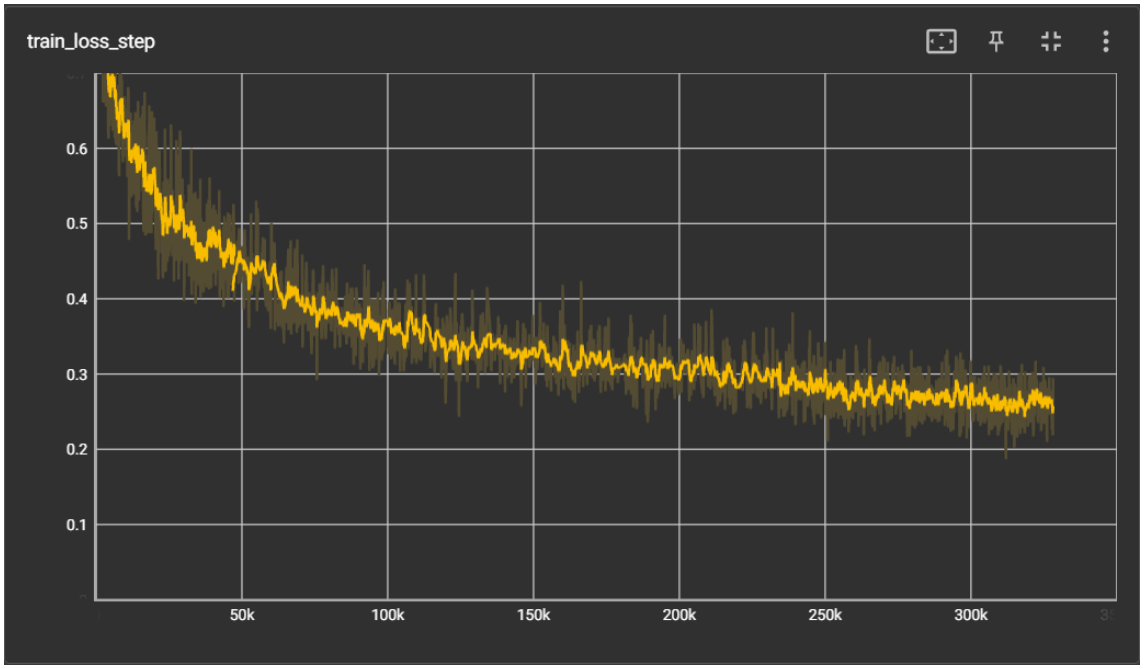

Figure 4: Train loss TP Transformer (Finishes with 0.2648).

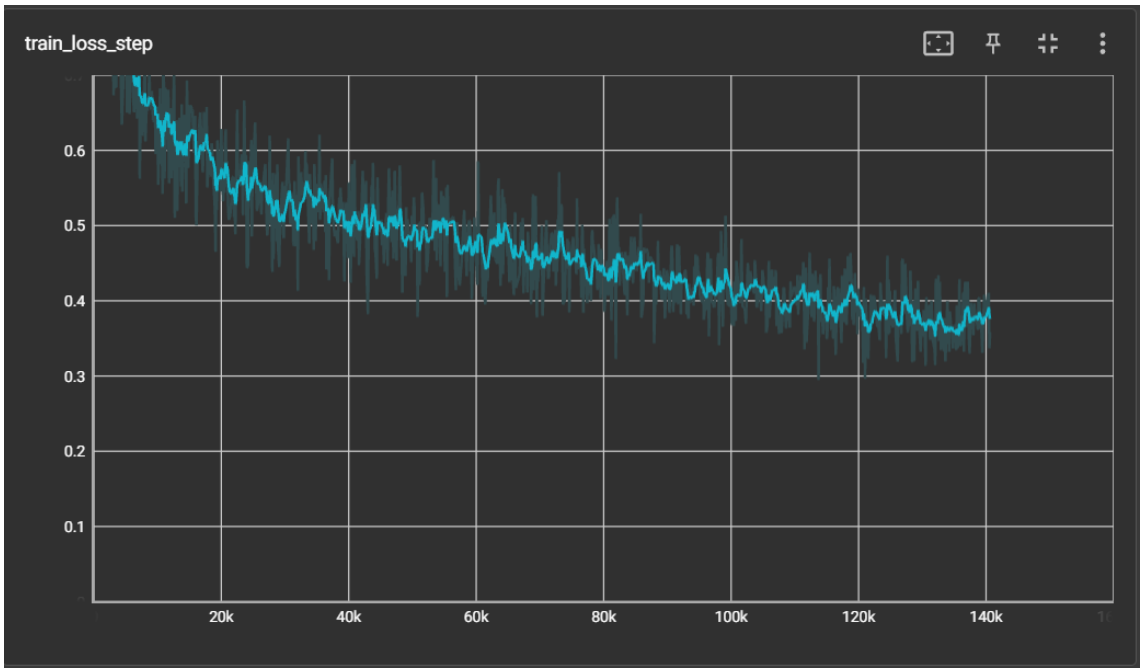

Figure 5: Train loss Vanilla Transformer (Finishes with 0.3941).

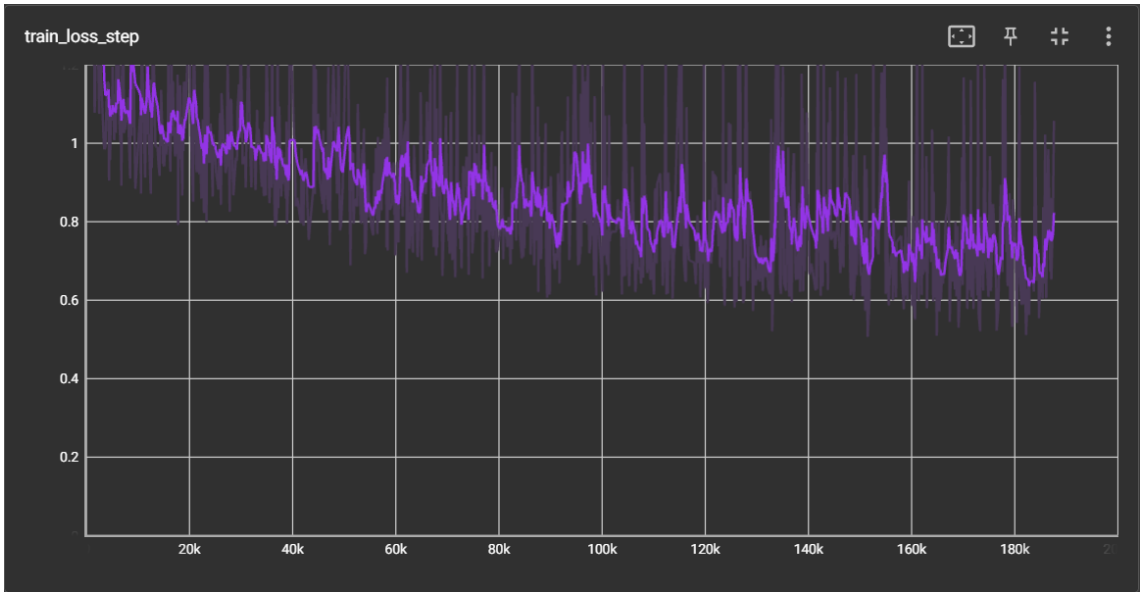

Figure 6: Train loss LSTM (Finishes with 0.746).

References

[1] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing Mathematical Reasoning Abilities of Neural Models, 2019.

[2] Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. Enhancing the Transformer with Explicit Relational Encoding for Math Problem Solving, 2019.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need, 2017.

[4] Alexander Rush. The Annotated Transformer. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 52–60, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[5] Pritam Chanda. Simple SEQ2SEQ machine translation using GRU based encoder-decoder architecture, Feb 2021.

[6] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent Neural Network Regularization, 2015.

[7] ischlag. TP-Transformer, Mar 2021.

[8] georgeyiasemis. Recurrent-Neural-Networks-from-scratch-using-PyTorch, Mar 2021.