Introduction

This is the report for the Deep Learning project in which we try several models to answer mathematical questions.
The first thing we did was read the paper [1] in which the dataset is described. This allowed us to better understand the end goal of the authors and helped us formulate some ideas on how to approach the task.
We decided to implement some of the models discussed in the paper; in this way, we would have a baseline to compare with our implementation. In particular, we thought that it would be interesting to compare the results obtained by an LSTM model and a Transformer model because they both can work with sequence-to-sequence tasks but are based on different strategies.

Implementation and Limitations

To implement the models, we used, as specified by the requirements, pytorch lightning. All training and tests were carried out using the resources provided by the free version of Google Colab.
Before starting the analysis of the selected models, we should mention some limitations that influenced our work. Google Colab's free tier does not provide much computational power; this highly limited how many samples we were able to use and for how long we were able to train/test the models. Indeed, after some time of using the free GPU, the account that uses it would be temporarily suspended for exceeding the GPU availability time (circa four hours).
This means that we had to choose a relevant subset of the dataset to not exceed the available resources and we were unable to train as long as needed to reach the accuracy of papers [1] and [2].

The Dataset

The dataset has a total of 56 modules, each containing 2 million question-answer pairs and focused on a different area of interest (such as algebra, probability, calculus). Among the samples of a single module, it is possible to distinguish three types of training question: easy, medium, and hard. For testing each module, the dataset provides samples for performing two types of tests: Interpolation and Extrapolation. They, respectively, test how the model learned to solve similar questions and its ability to generalize.
In our case, due to the limitation mentioned above, we did not work with all 56 modules, but we selected three of them (numbers_round, calculus_differentiate, polynomials_evaluate) for a total of 6 million samples. In this way, our models are learning how to solve multiple different problems and cannot specialize on a single one, a result that would be in contrast with the paper's [1] goal of training a model that solves multiple problems from multiple areas of expertise.

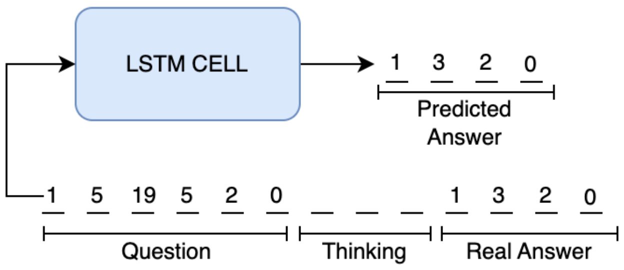
In our implementation, the dataset is loaded using the class *MathDataModule* that uses Hugging Face's *load_dataset* function together with custom loading scripts in which we define the data source and the splits. Once the samples have been loaded, we extract relevant information such as the dictionary containing all possible characters and their corresponding indexes, the maximum length of questions, and the maximum length of answers. Using these metadata, we encode the dataset so that it can be used by the model. For each sentence (question or answer), we turn each character into the corresponding index according to the vocabulary, add start and end of line tokens, and add padding to reach the maximum detected lengths. Check out an encoding example in Appendix A. We also provide a method to encode new questions not present in the dataset and use this class to create and return *Dataloaders* that we use in training, test and validation steps.

Metrics

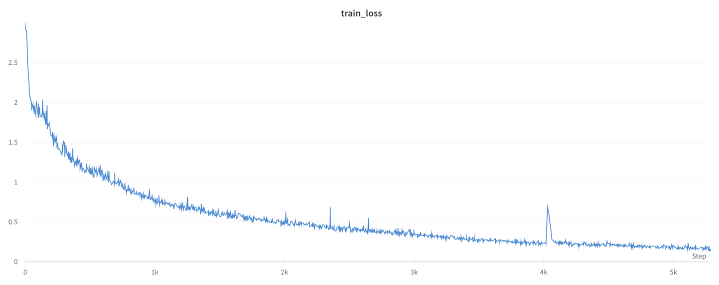
The metric that we use to assess the performance of the models is the accuracy of getting the correct answer to the question. More in detail, we followed the same criterion proposed in the dataset paper [1] in Paragraph 2.4. This means that the predicted answer is correct if and only if it matches character-for-character the correct answer. This also means that the model must be able to correctly predict the end of string token, otherwise the predicted answer would be wrong.
We report the precise accuracy values obtained by each model in its specific section.

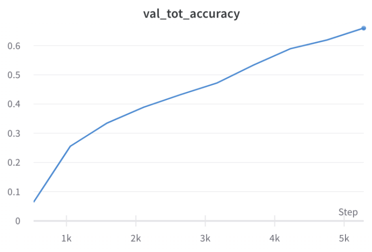
Baseline 1: LSTM

The first model that we implement for this task is inspired by the "Simple LSTM" presented in paper [1]. Its architecture is simple and straightforward because it is made up of a single LSTM cell. This cell sequentially processes the question and the answer, one character at a time.
Furthermore, as suggested in the paper [1], we introduced 16 "thinking steps" before outputting the answer so that the model can correctly process the question. To do so, we pass a zero input to the LSTM to update the hidden state and the cell state.



During training, we also introduced teacher forcing to improve the results. In this way, when computing a character of the answer, the model receives as input the previous correct character with a fixed probability instead of the predicted one.
We trained this model for 10 epochs using the Adam optimizer, the cross-entropy loss function, a batch size of 1024, a learning rate of 6×10^{-4} and other hyperparameters as suggested in paper [1]. We obtained an accuracy of 0.560 for the interpolation test set and 0.806 for the extrapolation test set.

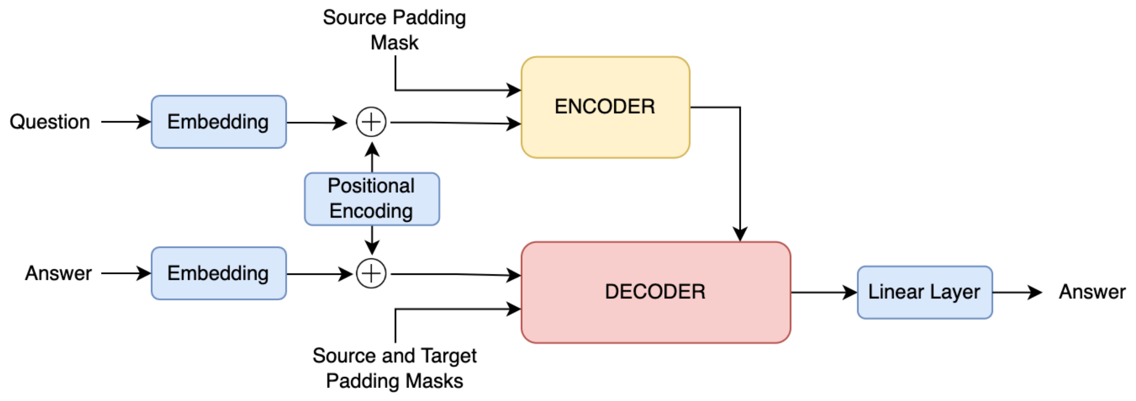




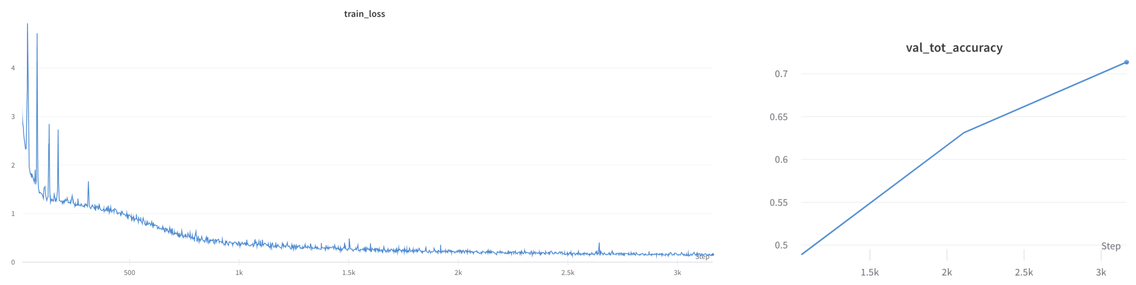
LSTM: on the left the loss during training, on the right the accuracy on the validation set at the end of every epoch

Baseline 2: Transformer

The second model we chose to implement is a standard Transformer that follows the architecture described in the paper "*Attention Is All You Need*" (Vaswani et al.)[3]. We used common values for the hyperparameters such as 8 heads for each multi-head attention component and 6 layers for both the encoder and the decoder. We also tested the model using some dropout layers as suggested in the paper [3] but we obtained worse results, so we decided to disable it.
Moreover, the weights of the embedding layer and the final linear layer are shared to ensure better decoding, as in paper [3]. In addition, we use sinusoidal positional encodings to take into account the order of the sequences.
In addition to the usual mask needed in the masked multi-attention mechanism, we also implemented padding masks to ensure that the padding at the end of each question and answer is not considered when computing the attention scores.



We trained this model for 3 epochs using the Adam optimizer, the cross-entropy loss function, a batch size of 512, a learning rate of 1×10^{-4} and other hyperparameters as suggested in paper [1]. We obtained an accuracy of 0.663 for the interpolation test set and 0.913 for the extrapolation test set.

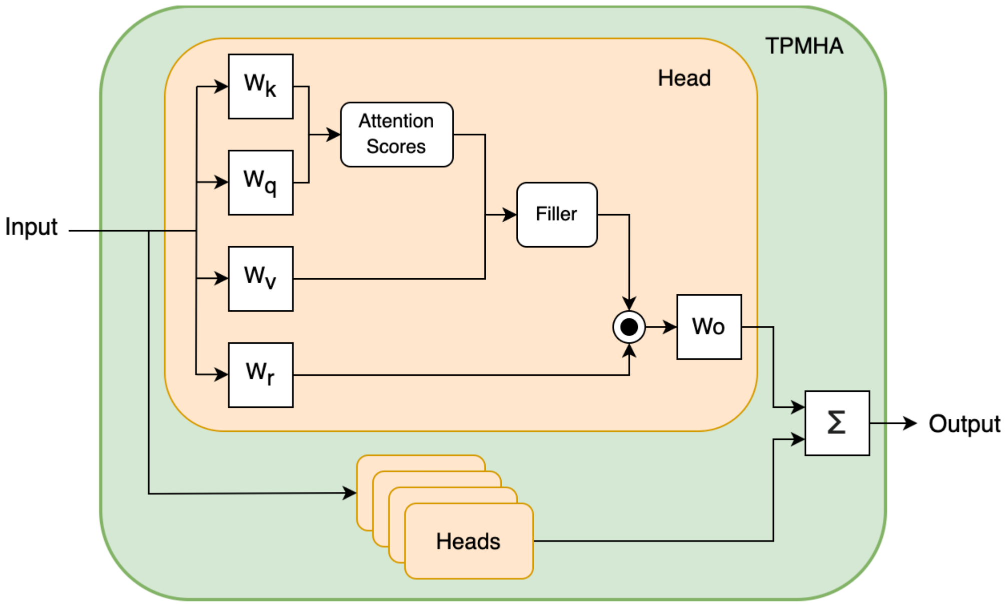


Transformer: on the left the loss during training, on the right the accuracy on the validation set at the end of every epoch

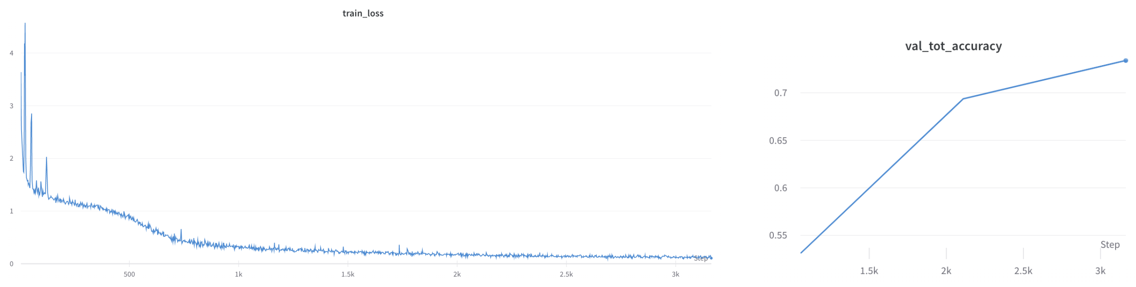
SOTA: TP-Transformer

The third and final model we implemented is the current state-of-the-art for the Mathematics Dataset: the Tensor-Product Transformer [2]. The architecture of this model is the same as the standard transformer except in the multi-head attention, which is replaced with a Tensor-Product multi-head attention (TPMHA).

In this model, there is a new *Role* vector in addition to the Key, Query, and Value vectors. We call *Filler* the result of the standard attention head. Each new head then binds that filler to its role via the tensor product and applies an affine transformation. Finally, the results of each head are summed to form the Tensor-Product Representation of the structure with multiple heads. To control dimensionality, the paper [2] suggests using pointwise multiplication, which is a contraction of the tensor product.



We trained this model for 3 epochs using the Adam optimizer, the cross-entropy loss function, a batch size of 512, a learning rate of 1×10^{-4} and other hyperparameters as suggested in paper [2] (embedding size 256 and dimension of the feedforward networks 2048). We obtained an accuracy of 0.684 for the interpolation test set and 0.957 for the extrapolation test set.



TP-Transformer: on the left the loss during training, on the right the accuracy on the validation set at the end of every epoch

Conclusions

Looking at the experimental results, we found that the Transformer and TP-Transformer models are slower in completing an epoch, but they need fewer of them to reach similar levels of accuracy with respect to the LSTM. Moreover, as we thought, the TP-Transformer performed better than the other models, both in the Interpolation and the Extrapolation test sets. However, we believe that our results are highly influenced by the lack of appropriate computational resources. We are pretty sure that we would get even better results if additional training were performed.

	LSTM	**Transformer**	**TP-Transformer**
Interpolation	56.02%	66.27%	68.42%
Extrapolation	80.59%	91.27%	95.72%

Table 1: Results.

References

[1] David Saxton et al. *Analysing Mathematical Reasoning Abilities of Neural Models.* 2019. DOI: 10.48550/ARXIV.1904.01557. URL: https://arxiv.org/abs/1904.01557.

[2] Imanol Schlag et al. *Enhancing the Transformer with Explicit Relational Encoding for Math Problem Solving.* 2019. DOI: 10.48550/ARXIV.1910.06611. URL: https://arxiv.org/abs/1910.06611.

[3] Ashish Vaswani et al. *Attention Is All You Need.* 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

Appendix A: Encoding

In the table, there is an example of how a pair of questions and answers is encoded. To each sentence we add the encoded representation of the start-of-line token (1), the end-of-line token (2), and the padding token (0). The encoding of these tokens was chosen by us and is always the same.

	ORIGINAL	**ENCODED**
Question	2+2	1 5 19 5 2 0 0
Answer	4	1 3 2 0