Project Report: Question answering on mathematics dataset

Abstract—**The authors of the paper proposed a model to solve math problems through deep neural networks. The task is to solve mathematical problem with a large range of problem types without any algebric formulas or notion. The structure consists in a encoder-decoder network. The main novelty introduced from the paper is the Tp-attention mechanism, a new Self-Attention mechanism, which explicitly encodes the relations between each Transformer cell and the other cells from which values have been retrieved by attention.**

I. Introduction

In this report we will give a brief introduction of the model, focusing in particular on the TP-Attention mechanism, then we will illustrate how we implemented the model. Just like the authors of the paper, we used the DM_Math dataset. For the implementation we used pytorch lightning. The dataset is made of pair question-answer like the following: *Factor d**3/5 - 125473*d**2/5 + 3073711536*d/5 - 4249998289728*, and the model must generate a result like *(d - 95665)*(d - 14904)**2/5*.

II. Dataset

Just a brief introduction on the Dm_Math Dataset. It has 3 different training classes, which differs by the complexity of the problem: training easy, training medium, training hard. For the test instead it have 2 classes: extrapolate and interpolate that refer to the mathematical task to be solved to get the answer.

A. Dataset loading

The DM-MAth Dataset is composed of 56 types of maths problem, for more than 120 million of pair question answer. Working with such a large dataset we faced two problems, both related to Google Colab free plan limits: the first one is about the ram occupation: even uploading 1/3 of the Dataset was enough to saturate Colab's RAM and make the system crash. The second one is about on Colab running time limit. The authors of the paper trained the model on the whole dataset, for a total of 1 million steps with a batch size of 1024. Unfortunately for us would not be possible to get even near to these number. Thus our choice was to make the whole dataset available but to leave the user free to choose the percentage of the dataset to utilize. In out tests we used 0.5 % of the whole dataset for training and 10% of the dataset for testing. We run the training for 2 epochs. This allowed us to complete the training and the testing of our model in about 140 minutes.

B. Loader and iterator

The model doesn't take in input words but works with token. Our vocabulary counts 73 token, one token more with respect to the vocabulary used by the authors of the paper. The token are both symbols and items such as $< SOS >$, $< EOS >$, $< PAD >$, $< UNK >$. With respect to the original vocab we added the $< UNK >$ token. Of course in our case the source vocabulary is the same as the target vocabulary. For our model we defined a max input sequence lenght of 200 token. In the dataset pre-processing, we take care for each batch to make all the sentences belonging to the same batch of the same length adding padding token. Later we get an iterator which returns batches of a specified batch size.

III. Trivial Baseline

As a trivial baseline we choose to implement a simple Neural Network with 200 input neurons units, so the input length of the network is fixed at 200 tokens. This is the reason why before feeding the network with any sequence we have to pad the data to make any sequence have a length of 200. The input layer has an output of 500 units, the hidden layer has 500, 2000 as input, output dimensions. For both layers we used ReLU as activation function. For what concerts the output layer, we have an input size of 2000 and an output size of max_output_len*73 We choose to set the maximum length of the output sequence to 35, so we have in output $35*73 = 2555$ units, these values represents the 35 outputs token, each one with the associated values. To get the probability associated with each token we applied a softmax to the last dimension of the output. Since we applied the softmax manually in the last layer we had to choose a loss function which didn't take care of applying the softmax , so we used negative log likelihood loss, with the option to ignore the padding value, otherwise the model would have focuses on the padding tokens. In the end the output of a single iteration of the model has the shape: [batch_size,out_seq_length,vocab_size]. We got the predicted token for each item of seq_lenght buy applying the argmax function in the last dimension. At the end we get the output with shape [batch_size, out_seq_len]. In the computation of the accuracy we took care to cut the output sequence, which dimension is a fixed 35, to the dimension of the true labels. In this way we avoided that the padding and other unuseful token could influence the accuracy measurement.

IV. Transformer

The normal transformer is composed by a lattice of (t,l) cells, where the t is the sequence element of the input and l is the layer indices with a t=1,...,T and l=1,...L. All cells share the same topology and the cells of the same layer share the same weights. More specifically, each cell consists of an initial layer normalization (LN) followed by a Multi-Head Attention (MHA) sublayer followed by a fully-connected feed-forward (FF) sub-layer. Each sub-layer is followed by layer normalization (LN) and by a residual connection. The cell structure follows the original transformer by (Vaswani et al., 2017)

A. *Masks*

To get sequences of the same length in each batch, padding token have been added to short sentences, but we don't want the model to focus and compute attention on the padding tokens, so adding a target mask in the decoder attention was not enough, but we had to add also a source mask for the input of the model. The aim of this mask is to set to a very low number the values associated with padding token, in this way they do not influence attention computation.

V. TP-Transformer

The TP-Transformer presents a single difference with the normal transformer. The Multi Head Self Attention is replaced by a TP-SelfAttention described in the below paragraph. But for the rest it has the same structure of normal Transformer.

A. *Tp-Attention*

The representation of each constituent is built compositionally from two vectors: one vector that embeds the content of the constituent, the 'filler', the vector returned by attention and a second vector that embeds the structural role it fills, a relation conceptually labeling an edge of the attention graph. The vector that embeds a filler and the vector that embeds the role it fills are bound together by the tensor product to form the tensor that embeds the constituent that they together define. The relations here, and the structures they define, are learned unsupervised by the Transformer. In the model, the TP-Transformer, each head of each cell generates a key, value and query vector, as in the Transformer, but additionally generates a role-vector. The query is interpreted as seeking the appropriate filler for that role (or equivalently, the appropriate string-location for fulfilling that relation). Each head binds that filler to its role via the tensor product (or some contraction of it), and these filler/role bindings are summed to form the TPR of a structure with H constituents.

VI. Results

A. *Train Setting*

As said before, because of Google Colab running time limit we choose to train on the 0.05 % of the whole dataset. Just like the authors of the paper we used the Adam optimizer, with learning rate = 0.0001 , beta1 = 0.9, beta2 = 0.995. The authors trained with a batch size of 1024, unfortunately because of the GPU ram size of Google Colab we could only train using a batch of 256. Training on the 0.05 percent of the dataset, for 2 epochs, allowed us to train only for about 4k steps, with a batch size of 256, where the authors trained for 1000k steps using 1024 batches. We used the same approach for all the 3 models

	Parameters	Dataset %	Epochs
Trivial baseline	6.2M	0.05	2
Transformer	44.5M	0.05	2
Tp-transformer	49.2M	0.05	2

Fig. 1. Training details

We got the following results:

	Interpolate	Extrapolate
Trivial baseline	2.26	2.36
Transformer	8.36	10.61
Tp-transformer	8.37	10.61

Fig. 2. Accuracy results

As excepted using the trivial Neural network we get a really low accuracy of 2.26 on the interpolate and 2.36 on the extrapolate test dataset. Unfortunately even by using the Transformer or Tp-Transformer we could not get even near to the accuracy obtained from the authors of the paper. We got 8.36% and 10.61% for Transformer and 8.37% and 10.61% for Tp-transformer on the Interpolate and Extrapolate testing dataset respectively. We got these very low accuracy results probably because of the enormous difference in the training time between our model and the authors of the paper. We managed to train our model on about 2k batches of size 256 for 2 epochs, while the authors trained for 1k steps with batch size 1024. This means that we trained our model for only $\frac{1}{1000}$ of the time w.r.t the authors of the paper.

	Interpolate	Extrapolate
Trivial baseline	-0.27	-0.22
Transformer	3.30	3.30
Tp-transformer	3.31	3.31

Fig. 3. Loss results

For what concerns loss result with the trivial baseline we get a -0.27 and -0.22 loss values on the interpolate and extrapolate test dataset. With the Transformer or Tp-Transformer we got 3.30 and 3.30 for Transformer and 3.31 and 3.31 for Tp-transformer on the Interpolate and Extrapolate testing dataset respectively.

We can see that we got very similar value for Transformer and Tp-transformer, apparently not enough to justify the implementation of the Tp-attention. This in our opinion because the very low training time of the model does not allow to exploit the advantages given by the relaction vector.