REPRODUCIBILITY CHALLENGE: DEEP LEARNING FOR SYMBOLIC MATHEMATICS

Xiaoxuan Wu, Ruixue Guo, Zeliang Zhao & Wanyuan Lin
School of Electronics & Computer Science
University of Southampton
{xw5u21,rg6g21,zz8u21,wl7n21}@soton.ac.uk

ABSTRACT

It has been proved in *Deep Learning for Symbolic Mathematics* Lample & Charton (2019) that neural networks perform good on mathematical tasks. This project aims to re-implement parts of the experiments established in that paper and analyse its reproducibility. The code of our experiment is available at https://github.com/cora0305/COMP6248-Reproducability-Challenge.

1 INTRODUCTION

At a time when neural networks are widely used, people have relatively little attention on considering using it to solve mathematical problems such as symbolic calculations. Moreover, among existing studies on this topic, the majority of them treat it as arithmetic tasks like integer addition and multiplicationZaremba et al. (2014); Arabshahi et al. (2018). The original paper proposed the method to consider symbolic calculations as a target for NLP models and utilise sequence-to-sequence models (seq2seq) to solve this problem. Our team reproduced some key experiments and the details are as follows.

2 METHOD & ALGORITHM

In the original paper, the authors' main ideas are divided into three parts: generate data, train the model, and predict results. The first step is to generate data so that the deep learning model is a large number of known samples to do the training. The second step of training the model is to use the data from the first step to train the seq2seq model. The third step is to predict the outcome given a function or a differential equation, use the model that has been trained to predict the outcome, rank the multiple outcomes predicted, and choose the most likely one as the value for the symbolic calculation.

2.1 EXPRESSION AS TREES

Before using the seq2seq models to finish the task, it is necessary to convert mathematical expressions to trees and from trees to sequences, with operators and internal nodes, children, numbers, constants, and variables as leaves.

2.2 DATA GENERATION

Before generating the data, the authors placed the necessary restrictions on the scope of the data:

- expression with up to the $n = 15$ internal nodes
- $L = 11$ leaf values in $\{x\} \cup \{-5, \ldots, 5\} \backslash \{0\}$
- $p_2 = 4$ binary operators: $+, -, \times, /$
- $p_1 = 15$ unary operators:

$\exp, \log$, sqrt, $\sin, \cos, \tan, \sin^{-1}, \cos^{-1}, \tan^{-1}, \sinh, \cosh, \tanh, \sinh^{-1}, \cosh h^{-1}, \tanh^{-1}$

The original paper used three approaches to data generation, respectively Forward Generation (FWD), Backward Generation (BWD), and Backward Generation with Integration by Parts (IBP).

Forward Generation (FWD):

$$\overset{\text{Generate}}{\longrightarrow} f \overset{\text{Integrate}}{\longrightarrow} F$$

This approach generates a random function f, computes its antiderivative F using an external symbolic framework (Sympy, Mathematica, etc.), and adds (f, F) to the training set.

Backward Generation (BWD):

$$f \overset{\text{Differentiate}}{\longleftarrow} F \overset{\text{Generate}}{\longleftarrow}$$

This approach generates a function F, computes its derivative, and adds (f, F) to the training set. BWD leads to long problems with short solutions whereas FWD leads to short problems with longer solutions.

Integration by Parts (IBP):
This approach generates random functions F and G, computes their derivatives f and g and if $f * G$ is in the training set, computes the integral of $F * g$ with $\int Fg = FG - \int fG$.
In our task, we used the first two approaches (FWD and BWD) to generate data.

2.3 MODEL

- seq2seq model

The seq2seq model is converting a sequence signal as an input into an output sequence signal. Using a deep neural network, the process consists of two processes: encoding and decoding.

- Transformer model

The transformer model is a seq2seq model Vaswani et al. (2017).The architecture of transformer model contains two parts, the Encoder, and the Decoder. Unlike the original seq2seq model, there is no RNN in the transformer model, which is entirely based on Attention and fully connected layers.

3 IMPLEMENTATION DETAILS

3.1 DATASET

Basing the original paper, the implementation could be divided into two parts, the first one using the polynomial dataset and generated with the forward method, the second method using the BWD dataset [1] prepared from author that the task of integration and generated with the backward method.

3.2 BASELINE OF THE MODELS

The baseline of the models is same as the original paper, implementing the seq2seq model, using the default PyTorch's Transformer Encoder, Transformer Decoder module. Setting the learning rate with 0.0001, number of attentions of 8, number of encoder layer of 6, number of decoder layer of 6, model dimension of 512, and the Adam optimizer. Evaluating the performance of the model after each epoch on test and validation set. For evaluation, it simply compared the generated and target sequence. At inference, however, using sympy to compute the difference between the derivative of the generated primitive and the input. If there is no difference, it considers it as a valid answer.

3.3 FORWARD METHOD WITH POLYNOMIAL DATASET

By using the virtual machine prepared from ECS CAD Server, running the code in the GPU which Perf is P2 and the Pwr is P2. The environment is set with python 3.7, generating the data with the package sympy, building the model with PyTorch, and then training with PyTorch. Firstly, using the `data_gen.py` to generate the polynomial data setting with different numbers, such as 1000, 10000 and so on. As the limit of the GPU machine, we only run the dataset with the number about 100000. It is very slow with a dataset of 100000 numbers. Then, running the `train.py` file to check the result of the loss, this step becomes faster with the forward method.

[1] `https://dl.fbaipublicfiles.com/SymbolicMathematics/data/prim_bwd.tar.gz`

3.4 BACKWARD METHOD WITH INTEGRATION DATASET

By using the Google Colab virtual machine, running the code with the GPU that Tesla 48C. The environment is building with the file author prepared 'environment.yml', setting the environment with python 3.8, and the channels with PyTorch, anaconda and so on.
After activating the environment with the file, running the `main.py` with the different epoch sizes, batch sizes, epochs, training set and test set. Also, with the limit with the GPU, Tesla 48C could not run with the original dataset about 5000000 numbers, the maximum training set of the implementation is 1000000 numbers. Totally, running the five experiments with the different settings, the average time of each experiment is about 3-4 hours. Finally, the output results could be found with the generated log files in detail.

4 RESULT & ANALYSIS

4.1 RESULTS OF BWD DATASET

Table 1: Results of BWD Data

Experiment Number	Epoch Size	Batch Size	Epoch Number	Train Size	Test Size	Valid Size	Running Time (GPU)	Accuracy
#1	1024	32	30	1024	32	32	0:02:24	0.0%
#2	10000	32	30	10000	32	32	2:58:22	28.1%
#3	10000	32	30	10000	128	128	1:45:26	12.5%
#4	10000	32	40	10000	32	32	2:07:28	46.9%
#5	100000	32	30	1000000	500	500	4:28:37	95.4%

As shown in Table 1, Experiment 1 was an attempt to figure out the model complexity. The accuracy is 0.0% since the number of training set is too small. As shown in Experiment 2, the accuracy improves as the number of training sample increases.

Comparing the results of Experiment 2 and 3, it shows that the model will perform poorly if only increasing the number of test and validation set without increasing the number of training set with the same order of magnitude. From the previous experimental results, it is known that the accuracy is relatively high only when the ratio of training size and test size is large.

Further, by comparing Experiment 4 and 2, the accuracy is almost doubled by increasing the epoch number while maintaining the size of training and test sets. However, both the test and validation accuracy are unstable as shown in Figure 1. This issue will be addressed in the following experiment by increasing the number of training set.

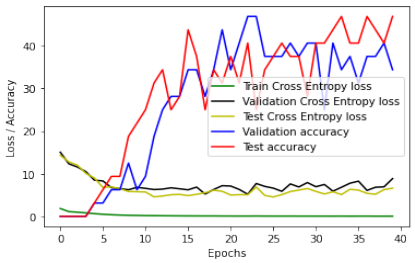

Figure 1: Fluctuations of Accuracy

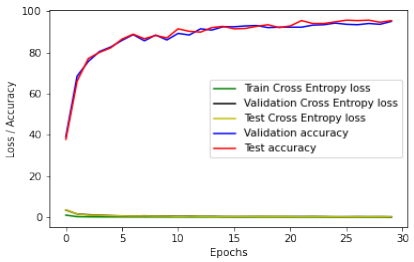

Figure 2: Final Result

Figure 2 presents the plot of Experiment 5. It can be observed that the accuracy is stable and steadily increasing with each subsequent epoch finally reaching 95.4%. It is promising to achieve the 98.4%

accuracy, which is implemented by the original paper, by increasing the sizes of training and test samples accordingly. Therefore, this is considered to be a successful reproduction of part of the original paper.

4.2 RESULTS OF FWD DATASET

Table 2: Results of FWD Data

Data	Batch Size	Epoch	Loss
100	64	20	0.52101
500	64	200	0.04062
1000	64	200	0.00804
10000	64	100	0.00983
10000	64	200	0.00347
100000	32	10	0.01900
100000	64	100	0.00958

Table 2 shows the results of FWD dataset. The trends of results are similar to those of BWD dataset, so it will not be described again. Notably, instead of using the given data, new data generated here by ourselves was used as the training set.

As mentioned in Section 3.3, the maximum 100,000 data were generated and trained with 100 epochs which took more than 27 hours. Due to the computing power of available GPU, this is the limit of the experiment that can be conducted.

5 CONCLUSION

In this task, we show that the machine translation model is suitable for symbolic mathematic tasks, and the seq2seq model can be used for difficult tasks like function integration. We use Forward Generation (FWD) to generate polynomials and express them as sum of powers. We use Backward Generation (BWD) to generate data and calculate the integration. After experiments, we show that the 6th experiment (100,000 epoch size, 32 batch size, 1,000,000 training size) has the best performance with the accuracy of 95.4%.
This approach takes advantage of neural networks to reconstruct mathematical problems based on translating some of math's complicated equations with the seq2seq model.

REFERENCES

Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. Towards solving differential equations through neural programming. In *ICML Workshop on Neural Abstract Machines and Program Induction (NAMPI)*, 2018.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. *Advances in Neural Information Processing Systems*, 27, 2014.