REPRODUCIBILITY REPORT DEEP SYMBOLIC REGRESSION: RECOVERING MATHEMATICAL EXPRESSIONS FROM DATA VIA RISK-SEEKING POLICY GRADIENTS

Samuel Lennard, Reivynn Chiang, Daniel Zelinka & Nikol Stoeva
Team: Symbolic Sleuths
Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
{sl2g19,rc1u19,dz1u19,ns1u19}@soton.ac.uk
https://github.com/slenn08/DSR

ABSTRACT

In this report we have reproduced the work from the paper ”Deep Symbolic Regression: recovering mathematical expressions from data via risk-seeking policy gradients”Petersen et al. (2021). We attempted to reproduce the results outlined in the paper for Deep Symbolic Regression (DSR) and two of its associated baselines, Priority Queue Training (PQT) and Vanilla Policy Gradient (VPG) on the Nguyen symbolic regression benchmarkNguyen et al. (2011). We achieved similar recovery scores of DSR, but found differences between the two baselines implemented in the paper with our implementation of PQT under-preforming significantly and VPG overperforming slightly.

1 INTRODUCTION

The main task of symbolic regression is to produce a concise, closed-form function[1]that fits a dataset[2]. Unlike previous attempts to solve this task, this paper opts to use a gradient-based approach with reinforcement learning. A recurrent neural network (RNN) is used to construct expression trees to attempt to fit the dataset. At each step of the construction, the RNN outputs a probability distribution over a short list of mathematical expressions. Samples are taken from this distribution to construct a finished expression, which are evaluated based on how close they match the dataset. This evaluation gives a reward signal for training the RNN which utilises risk-seeking policy gradient, a new contribution of this paper. This feedback cycle leads to better expressions having a higher probability of being sampled, which leads to the optimal expression being found.

2 EXPERIMENTAL METHODOLOGY

As we had a limited amount of time to implement and perform the experiments in the chosen paper, we had to establish a defined scope. Originally, the recovery rates for the benchmark Nguyen symbolic regression problems (Table 2) were computed against five other symbolic regression methods - VPG, PQT, Genetic Programming (GP), Eureqa, and Wolfram. For the purposes of this reimplementation and keeping in mind the time constraints, we decided to implement VPG and PQT. These baselines showcase the benefit of the risk-seeking policy gradient compared to vanilla policy gradient and priority queue training.

The experiments were ran on four computers with different CPUs and GPUs:

- AMD Ryzen 5 3600X 6-Core Processor, 16 GB RAM

[1]Function is defined as $f : \mathbb{R}^n - > \mathbb{R}$
[2]Dataset is defined as (X, y), where $X_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

- Intel(R) Core(TM) i7-11800H, RTX 3070 laptop GPU, 16 GB RAM
- Intel(R) Core(TM) i7-8750H CPU, NVIDIA GeForce GTX 1060 laptop GPU, 16 GB RAM
- Intel(R) Core(TM) i5-8300H CPU, 8 GB RAM

As the model can be ran on processors with a variety of specifications, we can conclude that the DSR algorithm is robust and compatible with most modern hardware.

Each Nguyen benchmark expression was attempted ten times by all algorithms. Table 1 outlines the recovery rates of each algorithm. During each model's run over 2000 epochs early stopping was used when exact symbolic equivalence is achieved. Although early stopping can significantly shorten the runtime for some expressions, it sometimes took up to 3 hours to run the algorithms fully. This had a substantial impact on the scope, and due to these constraints it was challenging to allocate sufficient time and resources for a thorough hyper-parameter search, and we therefore used the default hyper-parameters defined in the paper for each algorithm.

3 Implementation Details

All of the code was written by our group. The only use of the authors' code was to determine the model type and dimensions as they were not explicitly stated in the paper.

3.1 Model

The paper does not state the type of RNN used nor its dimensions, so we therefore had to make assumptions based on the author's implementation Petersen et al. (2021). In the original implementation, both LSTM and GRU models are implemented, and a default number of inputs, hidden, and cell size was set to 32. In our re-implementation, we used the LSTM model with the default dimension size, although we could not determine precisely what the original paper used to produce its results. We also made the assumption that each node of the expression tree can be represented with a 32-dimensional embedding, with empty nodes being represented with the 0 vector.

The model takes in as input the concatenation of the embeddings of the sibling and parent nodes to the current node, each of which are empty if they have not yet been filled in. During the forward pass, these embeddings with the previous hidden state are used to produce a categorical probability distribution over the nodes, which are then sampled to produce the next node. The process is repeated until a full expression tree is produced, which can then be evaluated on the dataset.

3.2 Environment

The Gymnasium library was used to define the reinforcement learning environment for the model. The environment was used to get an observation during each step based on the action and to calculate the reward of a full expression. The expression is implemented as a tree of nodes with a stack holding nodes with space for children, which is used to properly assign children nodes as they are added to the tree. At each step, the environment also produced a mask to disallow certain nodes from being sampled. Masked values were used to limit the size of the expression to be between 4 and 30, disallow all children of an operator to be constants, disallow the descendants of trigonometric operators to be trigonometric operators are implemented generally, and disallow children of unary operators being its inverse.

The reward function used a NRMSE loss of samples from a uniform distribution specified by each problem. We also implement constant optimization using a sub-optimization task with Limited memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS)Liu & Nocedal (1989). This gets ran whenever the reward function is called and constants are in use, so that constants are optimised with respect to the target dataset. However, we did not use it for any of the benchmarks but have just implemented it to fully recreate the work in the paper. These were manually tested on a few test cases to verify that it works correctly.

Table 1: Nguyen symbolic regression problem recovery rates and specifications. Inputs are x and/or y. U(a,b,c) depicts a uniform distribution of c random points between a and b. L_0 is a baseline library composed where L_0 = $\{+, -, \cdot, \div, \sin, \cos, \exp, \log, x\}$

Benchmark	Expression	DSR	PQT	VPG	Dataset	Library
Nguyen-1	$x^3 + x^2 + x$	100%	100%	100%	U(-1,1,20)	L_0
Nguyen-2	$x^4 + x^3 + x^2 + x$	100%	100%	100%	U(-1,1,20)	L_0
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	90%	0%	90%	U(-1,1,20)	L_0
Nguyen-4	$x^5 + x^4 + x^3 + x^2 + x$	90%	0%	90%	U(-1,1,20)	L_0
Nguyen-5	$\sin(x^2)\cos(x) - 1$	90%	20%	0%	U(-1,1,20)	L_0
Nguyen-6	$\sin(x) + \sin(x + x^2)$	100%	80%	100%	U(-1,1,20)	L_0
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	40%	20%	0%	U(0,2,20)	L_0
Nguyen-8	$\sqrt{x}$	100%	20%	0%	U(0,4,20)	L_0
Nguyen-9	$\sin(x) + \sin(y^2)$	100%	100%	100%	U(0,1,20)	$L_0 \bigcup \{y\}$
Nguyen-10	$2\sin(x)\cos(y)$	100%	90%	50%	U(0,1,20)	$L_0 \bigcup \{y\}$
Nguyen-11	x^y	100%	100%	100%	U(0,1,20)	$L_0 \bigcup \{y\}$
Nguyen-12	$x^4 - x^3 + \frac{1}{2}y^2 - y$	0%	0%	0%	U(0,1,20)	$L_0 \bigcup \{y\}$
	Average	85.0%	52.5%	60.8%		

Table 2: The original table of results for the different algorithms that we re-implemented.

Benchmark	Expression	DSR	PQT	VPG
Nguyen-1	$x^3 + x^2 + x$	100%	100%	96%
Nguyen-2	$x^4 + x^3 + x^2 + x$	100%	99%	47%
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	100%	86%	4%
Nguyen-4	$x^5 + x^4 + x^3 + x^2 + x$	100%	93%	1%
Nguyen-5	$\sin(x^2)\cos(x) - 1$	72%	73%	5%
Nguyen-6	$\sin(x) + \sin(x + x^2)$	100%	98%	100%
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	35%	41%	3%
Nguyen-8	$\sqrt{x}$	96%	21%	5%
Nguyen-9	$\sin(x) + \sin(y^2)$	100%	100%	100%
Nguyen-10	$2\sin(x)\cos(y)$	100%	91%	99%
Nguyen-11	x^y	100%	100%	100%
Nguyen-12	$x^4 - x^3 + \frac{1}{2}y^2 - y$	0%	0%	0%
	Average	83.6%	75.2%	46.7%

4 Analysis

4.1 Discussion about results

From Table: 1 we can see that DSR performed significantly better than its contemporaries which was in line with paper's results. From Table: 2 we can see that we achieved roughly the same results using DSR and had failures on the same functions showing that we had implemented the algorithm properly. PQT performed considerably worse than the implementation on the paper with lower performance on almost all the benchmarks. This is most likely due to a lack of clarity on how it was implement in the original paper as there was very little description of how this algorithm is performed. VPG in our testing performed significantly better than in the paper with the standouts being Nguyen-4 and Nguyen-5 with ours instead having issues with Nguyen-10. These problems in PQT and VPG can be attributed to the lack of clarity of how these were meant to be implemented in the paper as they were only briefly mentioned. It is also possible that due to our low trials we may have gotten unlikely results. If we had more time, more trials could be ran to get a better idea of performance.

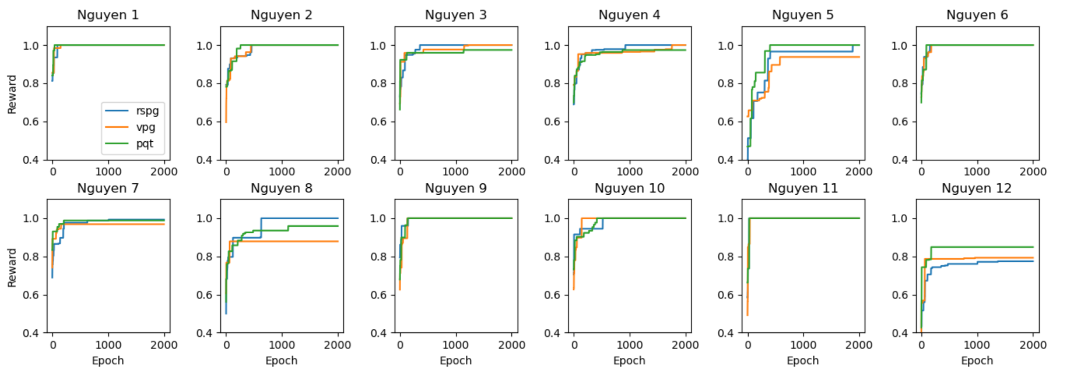

Figure 1: Reward curves of different Nguyen benchmarks.

Figure 1 shows plots of a single run for all 12 benchmarks. Each plot shows curves for DSR, PQT and VPG. The plots show that the speed of learning is pretty similar for all of the loss functions.

5 Conclusion on Reproducibility

Overall, we are able to reproduce DSR with it matching results from the paper and attempted to reproduce VPG and PQT to middling success. As expected DSR performed better than VPG and PQT which also shows that the novel introduction of the risk seeking policy gradient has significant merit in giving good results on this problem. However, our implementation of VPG and PQT did not match the performance given in the paper. This could be for a variety of reasons; for example, our number of trials were limited due to time constraints, one of our implementations could be incorrect, or one of our assumptions about the implementation could be different. Despite this, however, we were still able to recreate the central algorithm of the paper, and verify the claims made. Reinforcement learning has been shown to be a useful tool for symbolic regression, and risk-seeking policy gradient has been shown (in this case) to provide a tangible benefit to reinforcement learning.

References

Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

Quang Uy Nguyen, Nguyen Hoai, Michael O'Neill, Robert McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12:91–119, 06 2011. doi: 10.1007/s10710-010-9121-2.

Brenden K Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=m5Qsh0kBQG.