ECE 6254 Project Report: **Symbolic Regression via Deep Neural Networks and Policy Gradients**

Blair Johnson, Van Cuong Nguyen, Aaron Wasserman

May 6, 2022

Blair-Johnson/symbRegress[1]

1 Project Summary

Deep Learning and Neural Networks have been thriving for the last decade, playing an important role in various AI applications. However, these Deep Neural Networks (DNNs) have also become commonly regarded as black boxes. Ultimately, the complexity of DNNs limits their interpretability. In researching how to mitigate this limitation, we discovered *symbolic regression*, which aims to identify the mathematical relationship among variables by the combination of tractable operators. Not only does this approach produce a model for the underlying data, but also a human-understandable syntax expression.

In this project, we attempt to replicate the Recurrent Neural Network (RNN) with policy gradient method described in Petersen et al[1] for the symbolic regression problem. We started by representing math expressions as sequences and generating testing/training datasets. Then, we built our model using a long-short term memory (LSTM) model instead of a vanilla RNN to reduce the problem of vanishing gradients. Finally, we applied a simplified version of the REINFORCE algorithm policy gradient.

Our system successfully generates expressions and fits those expressions to the target data to obtain reward scores. The system trains successfully, resulting in increasing average reward scores over time; however, it cannot yet produce expressions that correctly fit the target functions. Section 5 details notable observations during training and Section 6 covers the reasons that correct expressions are not produced and proposes future work.

2 Motivation

Deep Learning and Neural Networks have been thriving for the last decade, playing an important role in various AI applications. Despite the ability to fit increasingly large datasets, Deep Neural Networks (DNNs), such as Fully-Connected Neural Networks, Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs), are often considered black boxes, meaning that they give us the approximate underlying representation of any given datasets, but do not justify why and how such representation has been generated. In other words, the complexity of DNNs limits their interpretability. Surveying how to mitigate this limitation led us to the topic of *symbolic regression*, which aims to identify the mathematical relationship among variables by the combination of tractable operators. While training DNNs requires fixing model structure and only fits model parameters, training a symbolic regressor involves finding both the model structure (the appropriate combination of math expressions) and parameters (proper constants and variables taken as input arguments of the operators). In this way, the resulting expression can be more explainable and provide meaningful insights about the dataset.

3 Background

Solving a regression problem requires finding a function that best fits given datasets. The main difference between Deep Learning techniques and Symbolic methods is the use of the core model (often a neural network) The former uses the DNN to directly compute the output, while the latter employs the DNN to search over the space of concise and closed-form mathematical expression to find the best combination.

[1]https://github.com/Blair-Johnson/symbRegress

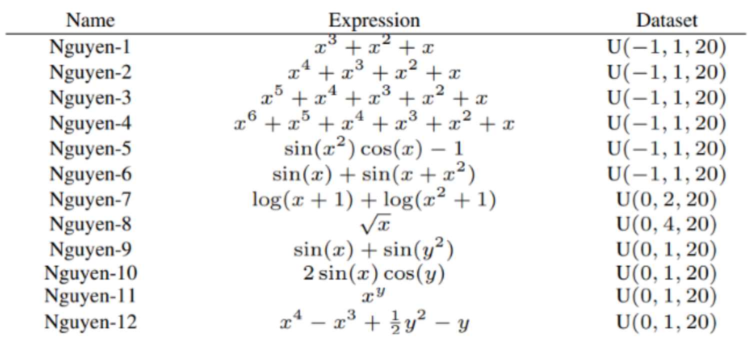
The search space often grows exponentially with the length of the expression. Some approaches utilized genetic programming (GP) algorithms [2, 3] to deal with this problem, such as [4]. However, GP tends to scale poorly to large datasets and be sensitive to hyperparameters. Recently, approaches leveraging deep learning for symbolic regression have been adopted. Sahoo et al. [5] developed a neural network whose activation functions are symbolic operators. [6, 7] leveraged autoregressive models (Recurrent Neural Networks – RNNs) to generate distribution over expressions, and [1] integrated policy gradients to mitigate the so-called "expectation problem" of RNNs.

4 Approach

In this project, we used the RNN-based with policy gradient method described by Petersen et al. [1] to the symbolic regression problem. Overall, the process involves three main steps: represent math expressions as sequences, build an RNN to generate math expressions from pre-set operators, and train the model using policy gradients.

4.1 Dataset Generation

In order to train and evaluate the fit of the generated model, we needed to create datasets. After researching previous benchmark suites for symbolic regression including those described in Petersen et al[1], we settled on the Nguyen Symbolic Regression Benchmark Suite[8] as shown in Figure 1. The Nguyen test cases use a reasonably constrained library of operators (+, -, ×, ÷, sin, cos, exp, log, x) and are common in current research. We used Nguyen-1 through Nguyen-8 to consider only cases with a single input variable, x. genDatasets.py[2] includes our code for generating random datasets for a given test case and seeds.



Name	Expression	Dataset
Nguyen-1	$x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-2	$x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-3	$x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-4	$x^6 + x^5 + x^4 + x^3 + x^2 + x$	$U(-1, 1, 20)$
Nguyen-5	$\sin(x^2)\cos(x) - 1$	$U(-1, 1, 20)$
Nguyen-6	$\sin(x) + \sin(x + x^2)$	$U(-1, 1, 20)$
Nguyen-7	$\log(x + 1) + \log(x^2 + 1)$	$U(0, 2, 20)$
Nguyen-8	$\sqrt{x}$	$U(0, 4, 20)$
Nguyen-9	$\sin(x) + \sin(y^2)$	$U(0, 1, 20)$
Nguyen-10	$2\sin(x)\cos(y)$	$U(0, 1, 20)$
Nguyen-11	x^y	$U(0, 1, 20)$
Nguyen-12	$x^4 - x^3 + \frac{1}{2}y^2 - y$	$U(0, 1, 20)$

Figure 1: The Nguyen Symbolic Regression Benchmark Suite [1].

4.2 Representing Mathematical Expressions as Sequences

Tree search, or tree traversal, is a central family of algorithms in the field of computer science. Data structures such as the binary tree enable highly-efficient searching and sorting, and important real-world tasks such as decision modeling, programming, and language parsing can be modeled as a tree search. Deep learning research across many task domains has created a new family of tree search algorithms that use artificial neural networks as a form of "intuition" to guide tree traversal when the search space becomes intractable for classical search algorithms.

In this project, we limited the set of operators (the search space) to eight: four *unary* operators: $\sin(), \cos(), \exp(), \log()$, which take one input argument; and four *binary* operators: $+, -, \times, \div$, which take two input arguments. We leveraged the fact that a mathematical expression can be represented as a tree, with leaf nodes being input arguments, which were either constants or variables, and internal nodes being operators. Since the output of a recurrent neural network is sequences, there is a need to sequence the expression tree, which can be done simply via a tree traversal. Therefore, we trained a recurrent neural network to generate the pre-order traversal of nodes, before recovering the math expression by reconstructing the corresponding tree of the traversal. For example, Figure 6 illustrates how the RNN generates the expression $\log(\cos(x)^2) + \sin(5x)$.

4.3 Deep Symbolic Regression

Our approach was primarily based on the work of Peterson et al. [1]. The authors of this paper posed the task of generating symbolic expressions as a reinforcement learning problem

[2]https://github.com/Blair-Johnson/symbRegress/blob/main/genDatasets.py

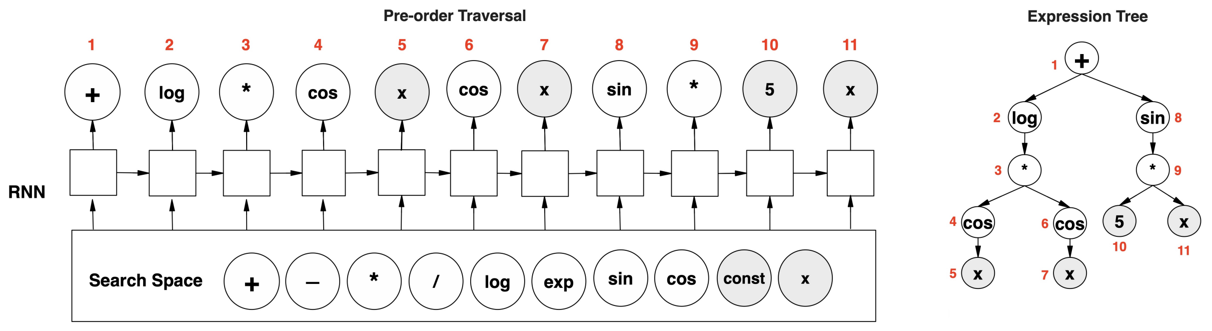


Figure 2: Mathematical expression generated from RNN: the model searches the token library, generating the pre-order traversal of a tree representing the expression.

in which a policy RNN, $\pi(\theta)$, would be trained to approximate the conditional distribution over future mathematical operations in a syntax tree, given the previous operations in the tree and data drawn from the distribution of the target function. This can be written as $\pi(\mathcal{T}_{0:k};\theta) \approx p(\tau_{k+1}|\theta, X, F_{\text{target}}(X))$, where $\mathcal{T}_{0:k}$ is the set of previous operations placed in a syntax tree, τ_{k+1} is the set of possible next operations to be placed in the syntax tree, θ are the parameters of the policy model, X is a dataset of target function inputs, and $F_{\text{target}}(X)$ is the dataset of corresponding target function outputs [1].

By training the policy model to approximate this distribution, we can generate syntax trees that maximize the probability of exactly fitting the target function by autoregressively sampling the most probable nodes from the policy distribution at each step of the preorder traversal, appending these nodes to our tree, and generating a new policy distribution from this tree. In this way, we can greedily generate a set of candidate trees, fit their constants to the training data using gradient descent, and then compare their performance against a threshold to determine the likelihood that a syntax tree is equivalent to F_{target}.

4.4 Reinforcement Learning Approach

The reinforcement learning algorithm used in the original paper involved two stages. In the first stage, the policy model would be used to sample a batch of syntax trees (the authors found that batch sizes of 1000 performed the best in their hyperparameter search). The constants in these syntax trees would be fit to the target function, and each tree would be assigned a reward according to the quality of its approximation of F_{target}. In the second stage, risk-seeking policy gradients (more on that later) would be calculated with respect to the policy model weights, and the policy model would receive parameter updates via these gradients [1].

Interestingly, the authors chose not to condition the policy model on data from the target functions at run-time, rather the entire system would need to be retrained in-order to fit a new target function. Under this paradigm, the policy model's only observations of the target function come from noisy reward values, and it never receives input data from the target function. The authors never explicitly address this decision, so we decided to precondition a policy model on the target data, hoping to create a more-general symbolic regression model that could identify different types of functions at run-time rather than requiring lengthy RL training for each inference.

4.5 Long Short Term Memory Model

The first difference between our approach and that of Peterson et al. is the use of a long-short term memory (LSTM) model, which was invented to reduce the problem of vanishing gradients of RNNs, rather than a vanilla RNN model. Another advantage of the LSTM cell for our use case is that it allows us to condition the policy model on a-priori information about the target function. We do this by using a 1D CNN as a feature extractor, taking vectors of input-output pairs from the target function and encoding them into a learned feature vector through layers of convolution kernels. We input this feature vector as the hidden state of the first LSTM execution in the syntax tree generation traversal. By doing this, every subsequent node selection can be informed by this prior information about the target function, theoretically allowing the model to generalize to new functions[3].

[3]There is an argument that experience with a sufficient variety of target functions could allow a model to interpolate a space of common features to expression fragment mappings and recover previously unseen combinations of these when presented. The practical viability of this is unknown to us.

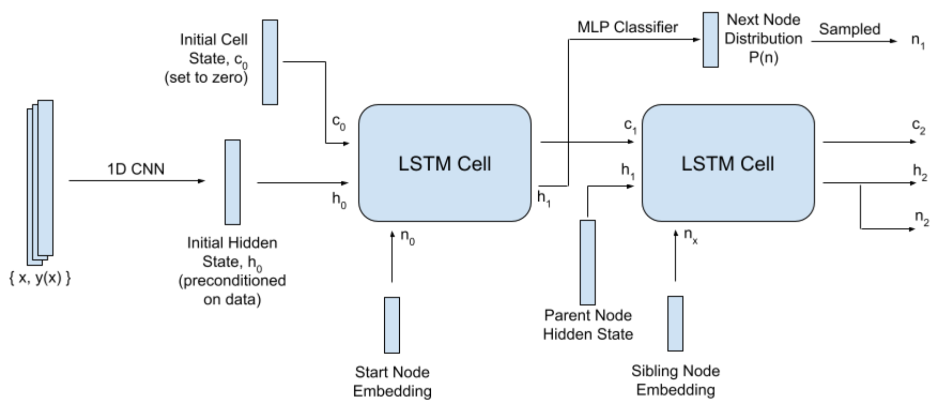


Figure 3: LSTM model employed in our project. Note that each instance of the LSTM cell in the diagram is a recursive call of the same LSTM cell. We use a 1D CNN as a feature extractor to precondition the LSTM at step 0 via its hidden state.

4.6 Policy Gradient Training

In reinforcement learning, it is predominantly the case that rewards are obtained through a non-differentiable, or unobserved environment, making traditional gradient descent methods impossible to employ. Policy gradients are a method for updating RL model parameters when direct backpropagation of gradients from a reward function to the model parameters is impossible. At their core, policy gradient algorithms seek to maximize the likelihood of actions that eventually produce favorable outcomes, and minimize the likelihood of actions that eventually produce non-favorable outcomes. Policy gradients are calculated using records of previously observed states, actions taken, and the eventual rewards obtained (propagated backward through time to support attribution).

In [1], the authors employ a risk-seeking policy gradient algorithm, selectively calculating gradients using the top ϵ-quantile of replays (by reward score). The authors claim that this formulation allows them to bias the policy model toward expressions that closely match the target by maximizing the top-end performance of the policy network rather than its expected performance [1]. In symbolic regression, finding the correct expression is more important than outputting expressions that approximate the correct expression in the limit. Thus, it makes sense to introduce additional variance in the policy network predictions if it increases the likelihood that the true expression will be discovered in a large batch of candidates drawn from the policy distribution. Due to computational and time constraints, we were unable to generate the huge number of performant replays necessary for the risk-seeking policy gradient method.

In our implementation of the symbolic regression system, we use a simplified version of the REINFORCE algorithm policy gradient: $\nabla J(\theta) \approx \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \pi(\tau_i;\theta) R(\tau_i)$. $J(\theta)$, here, is the value function representing the discounted sum of future rewards. Due to time constraints, we have not finished implementing the PyTorch hook functions necessary to perform the *mean* operation over accumulated gradients within the policy model. As a result, our model currently performs weight updates using the sum of policy gradients accumulated throughout an episode: $\nabla J(\theta) \approx \sum_{i=1}^{n} \nabla_\theta \pi(\tau_i;\theta) R(\tau_i)$. This undoubtedly increases gradient variance and hinders training stability (as we will discuss in Section 5).

5 Results

Our system successfully generates expressions and fits those expressions to the target data to obtain reward scores. The system trains successfully, resulting in increasing average reward scores over time; however, it cannot yet produce expressions that correctly fit the target functions posed to it. The reasons for this will be discussed in Section 6. The following are example expressions generated by our system. These range from short and simple to long and unreasonable:

$$xe^x$$
$$e^{x/(x+0.142)}$$
$$\sin((-0.403 + (-0.748 + x)))$$
$$0.522 + (x - \sin((-0.008 * ((1.119 - (-0.001/x))/x))))$$

5.1 Catastrophic Forgetting

A common problem in reinforcement learning is known as catastrophic forgetting. When this occurs, a policy model that was improving in performance suddenly loses its abilities, and the average reward drops. Figure 5 demonstrates this phenomenon occurring during the training of our system. To combat this, researchers store the weights from 'hall of fame' models that demonstrate the best performance, and they re-initialize their policy model weights to one of these stored models when catastrophic forgetting occurs. Similar techniques store periodic checkpoints and reinitialize those when performance drops. We have not had time to implement either of these techniques.

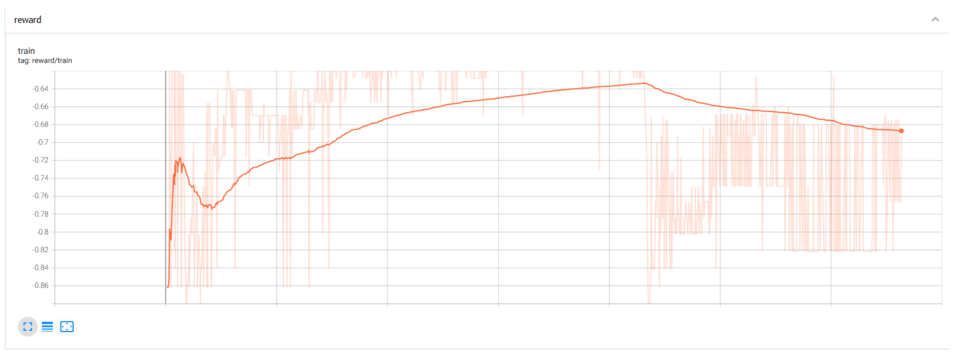


Figure 4: Average reward curve with clear drop in performance as model experiences catastrophic forgetting.

5.2 Learning Second-Order Polynomials

One interesting behavior that we noticed was the discovery of polynomial form when limiting our training set to polynomials of various degrees. Figure 6 demonstrates the average reward curve for this experiment. Prior to episode 400, the policy model generated deep and complex expressions with long chains of addition and multiplication, leading to noisy reward scores. After episode 400, the policy model converged to producing simple polynomial forms, explaining the jump in average reward. These forms rarely matched the target, often failing to ascribe constants to one or more terms of the polynomial, but the discovery of polynomial expressions alone was enough to produce more consistent rewards.

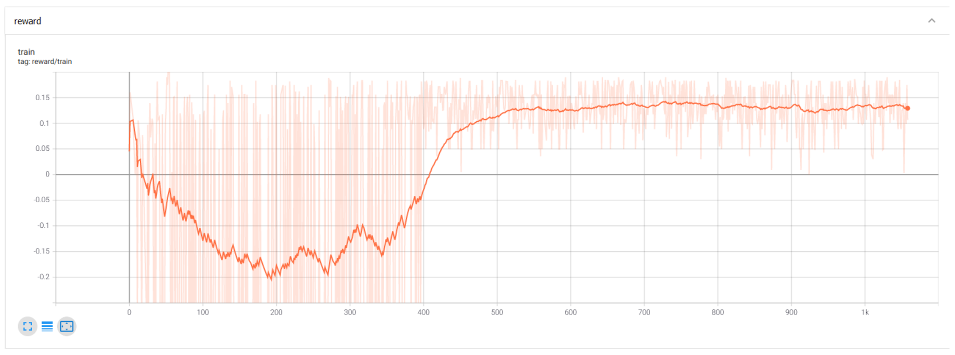


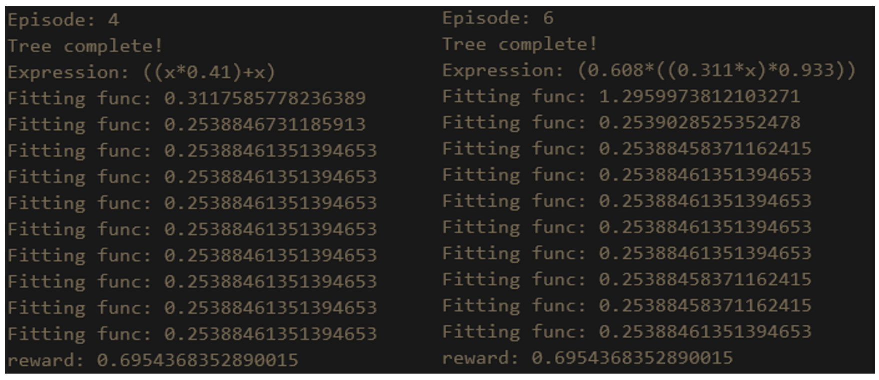
Figure 5: Average reward curve with sharp bump after model discovers second-order polynomial form.

5.3 Observations of Function Equivalence

One feature of our system is the practical equivalence of expressions that simplify one another. Figure 7 is a training log demonstrating two expressions discovered by the policy model. These expressions both reduce to the simple form $y = ax$ and after fitting their constants, both are ascribed the same reward because of this.

6 Conclusions and Future Work

This project was our group's first foray into reinforcement learning, and one of our primary observations was that RL, at least in the space of symbolic regression, relies heavily on engineering. The most complex component of this system was the tree data structure that we made. It needed to fulfill a number of requirements including syntax checking, embedding retrieval, parent and sibling queries, function compilation, and PyTorch parameter tracking and optimization. It was also apparent to us that the performance of our system was



```
Episode: 4                                Episode: 6
Tree complete!                            Tree complete!
Expression: ((x*0.41)+x)                  Expression: (0.608*((0.311*x)*0.933))
Fitting func: 0.3117585778236389          Fitting func: 1.2959973812103271
Fitting func: 0.2538846731185913          Fitting func: 0.2539028525352478
Fitting func: 0.25388461351394653         Fitting func: 0.25388458371162415
Fitting func: 0.25388461351394653         Fitting func: 0.25388461351394653
Fitting func: 0.25388461351394653         Fitting func: 0.25388461351394653
Fitting func: 0.25388461351394653         Fitting func: 0.25388461351394653
Fitting func: 0.25388461351394653         Fitting func: 0.25388461351394653
Fitting func: 0.25388461351394653         Fitting func: 0.25388458371162415
Fitting func: 0.25388461351394653         Fitting func: 0.25388458371162415
Fitting func: 0.25388461351394653         Fitting func: 0.25388461351394653
reward: 0.6954368352890015                reward: 0.6954368352890015
```

Figure 6: Expression 1 (left) reduces to be equivalent with expression 2 (right). Both expressions obtain the same reward after fitting their constants (bottom).

directly tied to the quality of the tree algorithms that we implemented. Our system began producing significantly more plausible expressions when we biased it towards them with a more complex syntax checking system. Originally we used a simple mapping denoting node types that may not be the direct child of each node type. By implementing a system for checking up the inheritance chain of the syntax tree, we were able to eliminate nested trigonometric and inverse functions such as $sin(cos(x))$ or $log(e^x)$. Further work on more complex syntax checking will help reduce the search space for our policy network and bias it toward scientifically plausible expressions.

Another area where our project needs further work is in its exploration-exploitation trade-off. Currently, the model operates in a purely exploitative manner, sampling operations from the discrete distributions generated by the policy model. Implementing techniques such as multi-armed bandits, as discussed in Dr. Muthukumar's guest lecture, could help our system avoid becoming siloed into producing only certain families of expressions.

Our 1D CNN encoder head is currently trained end-to-end with the rest of the system. The system would likely benefit from pretraining this branch as an autoencoder and using the resulting learned features as inputs to the LSTM. In doing so, we will give the encoder stronger supervision than it would receive via backpropagation through an LSTM, and the LSTM would receive more useful learned representations of the input data.

Finally, the most important step that we can take to improve our system going forward is the implementation of more sophisticated policy gradient estimation techniques. This is an engineering problem, as we need to store and manipulate gradient vectors for many episodes of training in-order to reduce variance in the stochastic gradient estimates.

7 Member Contributions

7.1 Coding

Coding contributions are visualized in our main branch's contribution page[4]. Van Cuong did additional work simplifying the original paper's code in this branch[5], those contributions are not reflected in the contribution page.

7.2 Final Report

- Blair Johnson: Sections 4.3-6
- Van Cuong Nguyen: Sections 2, 3, 4.2
- Aaron Wasserman: Sections 1, 4.1

References

[1] Brenden K Petersen et al. "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients". In: *arXiv preprint arXiv:1912.04871* (2019).

[2] John R Koza. "Genetic programming as a means for programming computers by natural selection". In: *Statistics and computing* 4.2 (1994), pp. 87–112.

[3] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. *Evolutionary computation 1: Basic algorithms and operators*. CRC press, 2018.

[4]https://github.com/Blair-Johnson/symbRegress/graphs/contributors

[5]https://github.com/Blair-Johnson/symbRegress/tree/simplify-paper-code

[4] T Nathan Mundhenk et al. “Symbolic Regression via Neural-Guided Genetic Programming Population Seeding”. In: *arXiv preprint arXiv:2111.00053* (2021).

[5] Subham Sahoo, Christoph Lampert, and Georg Martius. “Learning equations for extrapolation and control”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4442–4450.

[6] Daniel A Abolafia et al. “Neural program synthesis with priority queue training”. In: *arXiv preprint arXiv:1801.03526* (2018).

[7] Chen Liang et al. “Memory augmented policy optimization for program synthesis and semantic parsing”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[8] Nguyen Quang Uy et al. “Semantically-based crossover in genetic programming: application to real-valued symbolic regression”. In: *Genetic Programming and Evolvable Machines* 12.2 (2011), pp. 91–119.