Verifying Ordinary Differential Equations Using Neural Networks

Amatullah Sethjiwala
Department of Computer Science
University of Colorado Boulder
amatullah.sethjiwala@colorado.edu

Arth Beladiya
Department of Mechanical Engineering
University of Colorado Boulder
arth.beladiya@colorado.edu

Biljith Thadichi
Department of Computer Science
University of Colorado Boulder
biljith.thadichi@colorado.edu

Dheeraj Ravindranath
Department of Computer Science
University of Colorado Boulder
dhmu3474@colorado.edu

Abstract

In this project, we try to replicate the results of a conference paper published in ICLR 2018 (1), and it's updated version (2) (under review). We extend the paper from solving trigonometric and algebraic equations, to solving ordinary differential equations (ODEs). We also introduced a variant of Tree-RNN called Tree-GRUs and check its performance on the papers' datasets, and its efficacy on ODE equation verification task.

1 Introduction

The paper(s) introduces a novel Neural Programming technique for equation verification and completion. Neural Programming involves training Neural Networks to learn logic, mathematics, or computer programs. Until now, most neural programming research either relies solely on black-box function evaluations that do not capture the structure of the program, or on the availability of comprehensive program execution traces which are expensive to obtain. Both the methods have thus failed to generalize for equations of larger depth. The original paper aims to solve this problem by introducing a flexible and scalable framework for neural programming using Tree-LSTMs. It combines knowledge of symbolic representations of the relationships between the variables and the functions along with black-box function evaluations. The updated paper extends this original framework by augmenting Tree-LSTMs with an external memory called neural stacks. On equation verification tasks, this performs better than existing state of the art Tree-LSTM models by 10%.

1.1 Summary of Results

Our contribution is 4 fold. We (try to) replicate the results of the two papers and report our findings. We extend the original paper's equation generator to generate Ordinary Differential Equations (ODEs). We introduce 6 different Tree-GRU models and measure its performance on the original datasets consisting of algebraic and trigonometric identities as well as the ODE dataset generated by us. We show that not all of the original paper's results were reproducible due to errors [4]. We are able to only reproduce accuracies for 3 out of the 7 models within a reasonable margin ($< 5\%$) in Table:3. We explain the procedure to generate ODE equations [3], we show that Tree-LSTM and Tree-GRU have similar performance w.r.t accuracies on the original datasets in Table:4, and finally show that surprisingly, TreeLSTM performs well on ODE equation verification task Table:5 compared to its memory augmented counterpart.

1.2 Motivation

Machine learning algorithms like neural networks (NN), are excellent function approximators. They are the popular choice to model input-output relationships. Although Simple NN based algorithms like Fully-Connected/Recursive NNs perform well, they lack the ability to "reason". Advanced algorithms like Tree Long Short-Term Memory (TreeLSTM) networks have been proven to be powerful models to capture the semantics in data, which in our case is the compositionality of the equation and the relationships between the tokens. The ML models, on successful implementation in production, can replace the systems that power mathematical question answering systems (QA) like Mathematica.
Our motivation to choose this project is of two parts: firstly, to check if we could reproduce results of any recent ML research paper, as well as test our knowledge learnt in graduate ML class. Secondly, to check if we can make reasonable progress in extending the paper.

2 Dataset Generation

Since the data used to train this model is a set of correct and incorrect algebraic and trigonometric equations, we generate them using a set of axioms and identities scraped from Wikipedia. The scraped axioms and equations are then used to generate random equations. As random equations have more incorrect equations (identities) than correct ones, we use an approach from the paper called sub-tree matching algorithm to generate new identities.

2.1 Grammar

We first define the grammar of the mathematical identities using context-free grammar notation. In equation (2), E stands for the mathematical expression and can be composed of a terminal (T), a unary function $(F1)$ applied to any expression or a binary function applied to two expression arguments $(F2)$. This grammar covers the entire space of trigonometric and elementary algebraic identities. The trigonometry grammar rules are thus as follows:

$$I \rightarrow = (E, E), \neq (E, E) \qquad ...(1)$$

$$E \rightarrow T, F1(E), F2(E, E) \qquad ...(2)$$

Using the above mentioned grammar, we can create a parse tree for the evaluation of any expression. The parse tree helps with representing the equation to the ML model.
The depth of the tree is an indicator of the complexity of the function. Another indicator is the type of terminals used in the expression. For example: $1 + 1 + 1 + 1 = 4$ may be much easier to verify than $tan^2\theta + 1 = acos^2\theta$. Symbolic expressions have terminals of type constant or variable, whereas function evaluation expressions have constants and numbers as terminals.
As shown in Table 1, our domain includes 29 functions.

Table 1: Symbols in our grammar, i.e. the functions, variables, and constants

Unary Function $F1$				
sin	cos	csc	sec	tan
cot	$arcsin$	$arccos$	$arccsc$	$arcsec$
$arctan$	$arccot$	$sinh$	$cosh$	$csch$
$sech$	$tanh$	$coth$	$arsinh$	$arcosh$
$arcsch$	$arsech$	$artanh$	$arcoth$	exp
log				

Binary Function $F2$
$+$
$-$
pow
$diff$

Terminals T		
1	2	3
4	5	6
7	8	9
10	0	0.5
0.7	π	

2.1.1 Axioms:

The paper uses a small set of basic trigonometric and algebraic axioms to generate a large dataset of mathematical identities. Some examples of our axioms are (in ascending order of depth)
$x = x$, $x + y = y + x$, $x(yz) = (xy)z$, $sin^2\theta + cos^2\theta = 1$ and $sin(3\theta) = -4sin^3\theta + 3sin\theta$

2.2 Dataset of Mathematical Equations

The paper generates new mathematical identities by performing local random changes to the axioms. These changes result in identities of similar or higher complexity, which may be correct or incorrect, that are valid expressions within the grammar.

2.2.1 Generating Possible Identities:

First randomly select a node in the expression tree, then randomly select one of the following actions to make the local change to the equation at the selected node:

- *ShrinkNode*: Replace the node, if it's not a leaf, with one of its children, chosen randomly.
- *ReplaceNode*: Replace the symbol at the node (i.e. the terminal or the function) with another compatible one, chosen randomly.
- *GrowNode*: Provide the node as input to another randomly drawn function f , which then replaces the node.
- *GrowSides*: If the selected node is an equality, either add or multiply both sides with a randomly drawn number, or take both sides to the power of a randomly drawn number.

After we generate an identity, we use sympy to classify it as a correct or an incorrect equation. This method will result in an imbalanced dataset with a large number of incorrect identities.

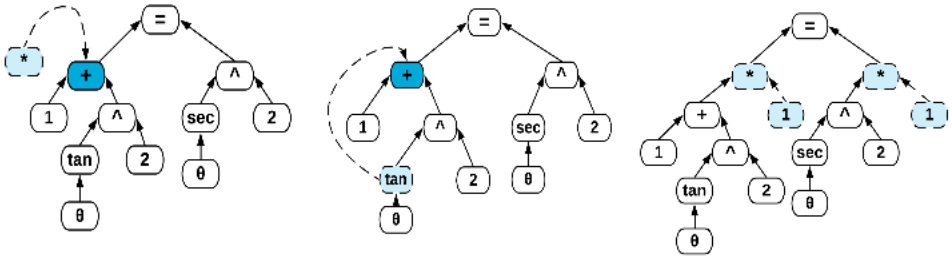

Figure 1: (a) Replace Node (b) Shrink Node (c) Grow Sides

2.2.2 Generating Additional Correct Identities:

To generate only correct identities, we maintain a dictionary of valid statements (mathDictionary) that maps a mathematical statement to another. For example, the dictionary key $x + y$ has value $y + x$. We look for keys that match a sub-tree of the equation then replace that subtree with the value of the key. E.g. given input equation $1 + tan^2\theta = sec^2\theta$, this sub-tree matching might produce equality $tan^2\theta + 1 = sec^2\theta$ by finding key-value pair $x + y : y + x$. We limit the depth of the final equation and only increase this limit if no new equations are added to the correct equations for a number of repeats.

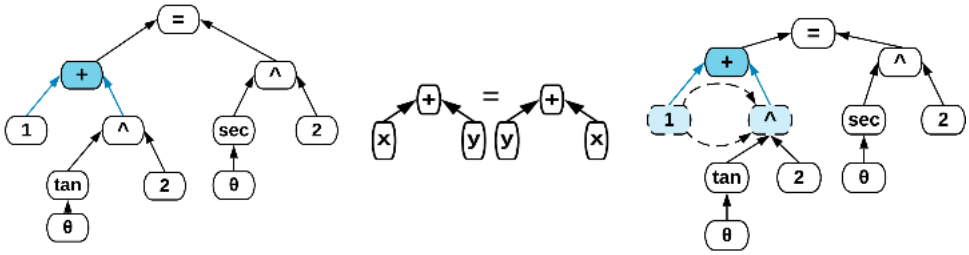

Figure 2: (a) Choose a node (b)mathDictionary (c) Match Value

2.2.3 Generating Numerical Expression Trees:

We represent the floating point numbers with their decimal expansion which is representable in our grammar. Eg : $2.5 = 2 * 10^0 + 5 * 10^{-1}$.

3 Generating Dataset for Ordinary Differential Equations

As an extension to the original paper, we proposed building an ODE solver (4). We can divide the solving of a differential Equation into two main steps:

1. Generating a set of candidate solutions for the differential equation.

2. Accepting the correct solution from the given set of candidate solutions.

In order to verify if a given candidate solution actually solves the ODE, we replace the node that represents the unknown function $f(x)$ with the candidate solution's EquationTree object. Figure 3 illustrates our approach. Then using our Machine Learning model, we can verify if the candidate solution actually solves the ODE. In this extension, we focus on finding a way for achieving Step 2. The algorithm to create this datset of ODE equations is discussed below.

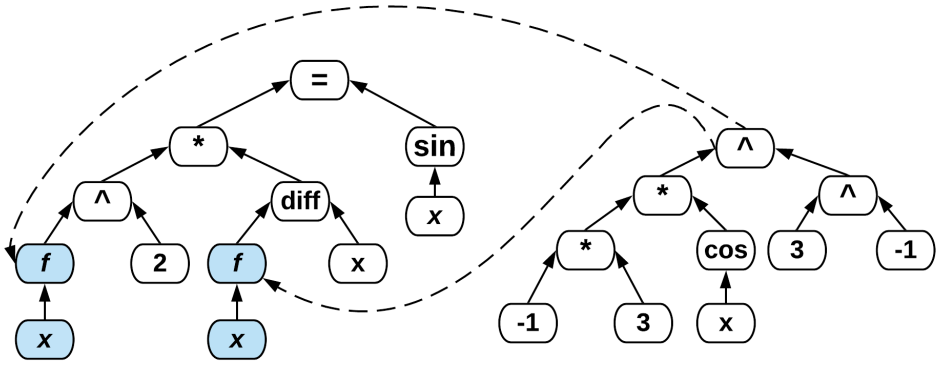

Figure 3: ODE : $f(x)^2 \frac{d(f(x))}{dx} = sin(x)$ and Solution : $f(x) = (-3cos(x))^{\frac{-1}{3}}$

3.1 ODE Dataset Generation Algorithm

Algorithm 1: Generate Equations Containing Derivatives

input : A dataset of trigonometric and algebraic equations created using the approach outlined in Section 2: $\mathcal{D}_{in}$
Number of equations to be generated: n

output : A dataset of equations that contain Derivatives: $\mathcal{D}_{out}$

while *n > 0* **do**

Choose random integers a_0, a_1
$lhs \leftarrow a_0 f(x) + a_1 \frac{d}{dx} f(x)$
Pick a random equation $e_{in} \in \mathcal{D}_{in}$
$rhs \leftarrow$ randomly pick lhs or rhs of e_{in}
$ode \leftarrow lhs = rhs$
$odeSolution \leftarrow sympy.dsolve(ode)$
Generate e_{out} by replacing function f in ode with $odeSolution$
Create EquationTree object for e_{out} and serialize it as $eTreeCorrect_{out}$
Do local change to a node in the EquationTree object to create $eTreeInCorrect_{out}$
$\mathcal{D}_{out} \leftarrow \mathcal{D}_{out} \oplus eTreeCorrect_{out} \oplus eTreeIncorrect_{out}$
$n = n - 1$

end

4 Challenges

As key parts of the code were missing, we contacted the author Dr. Forough Arabshahi. She was gracious enough to point us to an updated paper. The code was better documented and written in PyTorch. We then shifted our focus on the new paper.

4.1 DataSet Generation

The code available on GitHub for generation of trigonometric and algebraic identities was filled with errors. We encountered frequent crashes while trying to run the code. While making the *EquationTree* object for an identity, the code would not make a deep-copy of the node hence resulting in incorrect equation.
The author gave us the code for generation of ODE data. This code had several issues in terms of missing components and errors which we fixed. We added code for encoding of integers and floats as *EquationTree* objects. This would help create a more diverse dataset. Subsequently, we added code for checking whether the generated *EquationTree* object comes from the restricted grammar mentioned in Section 2.1. We fixed an error in the code which ran in an infinite loop when generating a negative ODE. We also made other minor changes in the ODE generation algorithm. The final version of our ODE generation algorithm is captured in Algorithm 1.

4.2 Model Building

As the new code was modular, we had to implement 3 classes: *EquationTree* object, *unary* Nodes, and *binary* Nodes. 2-3 days were spent in defining possible architectures of Tree-GRUs. It was challenging to come up different ways to generate reset and update gates - whether to combine the downstream left and right subtree hidden state into a single reset/update gate, and then generate the hidden state to be sent upstream like 5.1, 5.3, or 5.4; generate separate reset/update gates and then generate a single intermediary/final hidden state like 5.2, 5.5, or 5.6; or generate two intermediary hidden states, and then combine them to generate a final upstream hidden state like 5.6.
Quick iterations were impossible to make as the model took an average of 30 hours to train, and about 7.5 GB of RAM. We could only run 2-3 threads on 2 of our local machines.
We then modified our code to incorporate ODE Equations. We had to add additional constants like 'ComplexInfinity', 'NegativeInfinity', 'NaN' and functions like 'Derivative' and 'Tuple'. Tuple is a function which was generated by symPy as a way of denoting derivatives. It is a two-input function which takes in the variable with respect to what the derivative was calculated and second input as the degree of derivative. A setback we faced while trying to train the models was the change of log from a two-input function which took the base as input to a one-input function with base e. Although this change was easy to make, identifying the cause of the error took considerable time and debugging.
Another challenge was not being able to train several models due to hardware and time constraints. Memory augmented networks have several architectures for the neural stack implementation, all of which we were not able to test due to the aforementioned constraints. Moreover, the models also greatly depended on the activation function used on the different gates, most predominantly $sigmoid$ and $tanh$. We could not test out different models with varying activation functions.

5 TreeGRUs

Gated Recurrent Units are simpler versions of LSTMs, as, in comparision, they have less weights to train. We extended this similar intuition for TreeLSTMs. In this report, we introduce 6 different TreeGRU architectures, and submit to the reader, their performance on baselines of the paper, and the ODEs. To refresh the memory of the reader, a GRU has two gates

$$r_t = \sigma(U^r h_{t-1} + W^r X_t) \rightarrow (1)$$

$$z_t = \sigma(U^z h_{t-1} + W^z X_t) \rightarrow (2)$$

$$\tilde{h_t} = tanh(U(r_t \odot h_{t-1}) + W X_t) \rightarrow (3)$$

$$h_t = (1 - z_t) h_{t-1} + z_t \tilde{h_t} \rightarrow (4)$$

where r_t is the reset gate, and z_t is the update gate, and h_t is the hidden state that is sent upstream. The first two versions were a quick experiments, the latter 4 were thought through. We've included all 6 for readers' curiosity.

5.1 TreeGRU: V1

$$r_t = \sigma(U^{lr}h_{lt-1} + U^{rr}h_{rt-1} + W^r X_t) \rightarrow (1)$$
$$z_t = \sigma(U^{lz}h_{lt-1} + U^{rz}h_{rt-1} + W^z X_t) \rightarrow (2)$$
$$\tilde{h_t} = tanh(U(r_t \odot h_{lt-1}) + r_t \odot h_{rt-1}) + WX_t) \rightarrow (3)$$
$$h_t = z_t(h_{lt-1} + h_{rt-1}) + (1 - z_t)\tilde{h_t} \rightarrow (4)$$

5.2 TreeGRU: V2

$$r_l t = \sigma(U^{lr}h_{lt-1} + W^{lr}X_t) \rightarrow (1)$$
$$r_r t = \sigma(U^{rr}h_{rt-1} + W^{rr}X_t) \rightarrow (2)$$
$$z_t = \sigma(U^{lz}h_{lt-1} + U^{rz}h_{rt-1} + W^z X_t) \rightarrow (3)$$
$$\tilde{h_t} = tanh(U(r_l t \odot h_{lt-1} + r_r t \odot h_{rt-1}) + WX_t) \rightarrow (4)$$
$$h_t = z_t(h_{lt-1} + h_{rt-1}) + (1 - z_t)\tilde{h_t} \rightarrow (5)$$

5.3 TreeGRU: V3

$$r_t = \sigma(U^{lr}h_{lt-1} + U^{rr}h_{rt-1} + W_r X_t) \rightarrow (1)$$
$$z_t = \sigma(U^{lz}h_{lt-1} + U^{rz}h_{rt-1} + W_z X_t) \rightarrow (2)$$
$$\tilde{h_t} = tanh(U(r_t \odot h_{lt-1}) + r_t \odot h_{rt-1}) + WX_t) \rightarrow (3)$$
$$h_t = (1 - z_t)\frac{(h_{lt-1} + h_{rt-1})}{2} + z_t\tilde{h_t} \rightarrow (4)$$

5.4 TreeGRU: V4

$$r_l t = \sigma(U^{lr}h_{lt-1} + W_{lr}X_t) \rightarrow (1)$$
$$r_r t = \sigma(U^{rr}h_{rt-1} + W_{rr}X_t) \rightarrow (2)$$
$$z_t = \sigma(U^{lz}h_{lt-1} + U^{rz}h_{rt-1} + W_z X_t) \rightarrow (3)$$
$$\tilde{h_t} = tanh(U(r_{lt} \odot h_{lt-1} + r_{rt} \odot h_{rt-1}) + WX_t) \rightarrow (4)$$
$$h_t = (1 - z_t)\frac{(h_{lt-1} + h_{rt-1})}{2} + z_t\tilde{h_t} \rightarrow (5)$$

5.5 TreeGRU: V5

$$r_{lt} = \sigma(U^{lr}h_{lt-1} + W^{lr}X_t) \rightarrow (1)$$
$$r_{rt} = \sigma(U^{rr}h_{rt-1} + W^{rr}X_t) \rightarrow (2)$$
$$z_{lt} = \sigma(U^{lz}h_{lt-1} + W^{lz}X_t) \rightarrow (3)$$
$$z_{rt} = \sigma(U^{rz}h_{rt-1} + W^{rz}X_t) \rightarrow (4)$$
$$\tilde{h_t} = tanh(U(r_{lt} \odot h_{lt-1} + r_{rt} \odot h_{rt-1}) + WX_t) \rightarrow (5)$$
$$h_t = \frac{(1 - z_{lt})h_{lt-1} + (1 - z_{rt})h_{rt-1} + (z_{lt} + z_{rt})\tilde{h_t}}{2} \rightarrow (6)$$

5.6 TreeGRU: V6

$$r_{lt} = \sigma(U^{lr} h_{lt-1} + W^{lr} X_t) \rightarrow (1)$$

$$r_{rt} = \sigma(U^{rr} h_{rt-1} + W^{rr} X_t) \rightarrow (2)$$

$$z_{lt} = \sigma(U^{lz} h_{lt-1} + W^{lz} X_t) \rightarrow (3)$$

$$z_{rt} = \sigma(U^{rz} h_{rt-1} + W^{rz} X_t) \rightarrow (4)$$

$$\tilde{h_{rt}} = tanh(U(r_{lt} \odot h_{lt-1} + r_{rt} \odot h_{rt-1}) + W X_t) \rightarrow (5)$$

$$\tilde{h_{lt}} = tanh(U(r_{lt} \odot h_{lt-1} + r_{rt} \odot h_{rt-1}) + W X_t) \rightarrow (5)$$

$$h_t = \frac{(1 - z_{lt}) h_{lt-1} + (1 - z_{rt}) h_{rt-1} + z_{lt} \tilde{h_{lt}} + z_{rt} \tilde{h_{rt}}}{2} \rightarrow (6)$$

6 Experimental setup

We verify the paper's claim of using Recursive NNs to the task of mathematical equation verification as defined by (1). The task of equation verification is a binary classification task which expects the model to predict whether the equation is True or False. Baseline models such as Recursive NN, TreeLSTM, and TreeLSTM with augmented memory are tested for reproducibility.
The models are implemented in PyTorch. They are optimizing the softmax loss on the output of the root. The root represents equality computed by the dot product of output embeddings of left and right subtree. The input are through the terminal nodes of the tree which represents symbols consisting of variables and numbers. The leaves are two-layer feed-forward networks that embed symbols and numbers in the equation, and the other tree nodes are single-layer neural network representing different (sub) functions of the equation. Each type of operator uses the same set of parameters.
All models uses Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and learning rate of 0.001. Weight decay is 0.00001 and the hidden dimensions of the model is 50. Baselines models were run using 3 different seeds and the reported results are average of 3 seeds. The best model is the one with the best accuracy on the validation data, with the corresponding test results being reported. The training is on equations of depth 1-7, and the test of baseline models is from depth 8-13. For ODEs, similar rules apply as above, except, training and testing is done with depths 1-32. The evaluation metrics are accuracy, precision, and recall.

7 Results

We reproduce the baseline results of the new paper, as well as share the performance of Tree-NN, Tree-LSTM variants, and Tree-GRU variants on ODEs. The tables 2, 3 show the claimed results and the results reproduced by us. Models highlighted in bold are the ones which had more than 5% reduction in its F1 score relative to the claim made by authors.
Table 4 shows the performance of TreeGRUs on the original dataset. It can be infered that TreeGRUs performance is on part with its TreeLSTMs counterpart.
Table 5 shows the performance of Tree-GRUs on ODEs in addition to the models considered in the paper. The results are much higher on the test-set compared to the baseline results. Also, it is intriguing to see TreeLSTM perform better than its memory augmented counterpart. One of the reasons for this behavior might be, we trained the models on equations of every depth and tested it on equations of every depth whereas the original paper trained on smaller depths and tested on larger depths.

8 Conclusion

In this report we showcase our project work done over the last semester. We (unsuccessfully) try to replicate the original paper. We reproduce results of the updated paper with good outcomes. We extend the paper to work for ODEs. We introduce a simpler Recursive-NN called TreeGRU and test its performance on ODEs as well as the paper's datasets.

Table 2: **Accuracy of the models on Symbolic Dataset: Claimed Results**

Approach	Train (Depths 1-7)			Validation (Depths 8-13)			Test (Depths 8-13)		
	Acc	Prec	Rcl	Acc	Prec	Rcl	Acc	Prec	Rcl
Majority Class	58.12			56.67			51.71		
Tree-NN	96.03	95.36	97.94	89.11	87.79	93.84	80.67	82.63	79.29
Tree-NN + Stack	95.92	95.74	97.32	88.88	87.37	93.95	78.02	81.28	74.74
Tree-NN + Stack + no-op	95.86	96.40	96.49	88.44	87.21	93.29	78.87	83.23	74.10
Tree-LSTM	99.34	99.40	9.47	93.86	92.67	96.82	83.64	86.96	80.45
Tree-LSTM + Stack	99.23	99.19	99.49	93.31	92.36	96.15	85.24	85.13	86.59
Tree-LSTM + Stack + Normalize	98.76	98.59	99.29	93.32	92.23	96.33	84.44	85.49	84.16
Tree-LSTM + Stack + Normalize + no-op	98.34	98.13	99.04	93.84	92.60	96.87	86.01	86.27	86.78

Table 3: **Accuracy of the models on Symbolic Dataset: Reproduced Results**

Approach	Validation (Depths 8-13)			Test (Depths 8-13)		
	Acc	Prec	Rcl	Acc	Prec	Rcl
Majority Class	56.67			51.71		
Tree-NN	90.05	88.33	94.32	79.32	83.66	74.75
Tree-NN + Stack	81.33	81.78	86.28	69.45	69.55	72.78
Tree-NN + Stack + no-op	81.85	83.01	85.48	69.23	71.06	68.32
Tree-LSTM	94.69	93.79	97.06	83.71	86.76	80.92
Tree-LSTM + Stack	89.8	88.85	93.82	79.06	83.70	74.15
Tree-LSTM + Stack + Normalize	93.82	92.72	96.66	80.98	85.30	76.36
Tree-LSTM + Stack + Normalize + no-op	94.11	92.83	97.11	86.16	85.51	88.19

8.1 Future Work

In the future, we would like to generate enough equations for each depth (1-32). We would like to train the Tree-GRU models on smaller depths, and test for generalization on on larger depths. We would also like to see if the results of the updated paper holds true for equations involving inequalities.

Table 4: **Accuracy of the TreeGRU models on original dataset (100 epochs)**

Approach	Validation			Test		
	Acc	Prec	Rcl	Acc	Prec	Rcl
Majority Class	56.67			51.71		
Tree-GRU V1	94.49	93.62	96.88	80.46	84.59	76.08
Tree-GRU V2	94.18	93.46	96.50	79.97	83.40	76.52
Tree-GRU V3	95.07	94.21	97.29	82.97	85.75	80.41
Tree-GRU V4	95.05	94.17	97.28	83.53	85.21	82.48
Tree-GRU V5	94.21	93.46	96.54	82.21	84.67	80.10
Tree-GRU V6	88.07	86.47	93.62	77.56	82.49	71.84

Table 5: **Accuracy of the models on ODE Dataset (30 epochs)**

Approach	**Validation (Depths 1-32)**			**Test (Depths 1-32)**		
	Acc	**Prec**	**Rcl**	**Acc**	**Prec**	**Rcl**
Majority Class	50.01			50.00		
Tree-NN	89.30	83.08	98.75	89.65	83.76	98.40
Tree-LSTM	96.18	93.61	99.15	96.46	94.18	99.03
Tree-LSTM + Stack + Normalize + no-op	95.80	93.45	98.55	95.75	93.39	98.50
Tree-GRU V3	95.76	93.53	98.35	94.93	92.08	98.33
Tree-GRU V4	95.76	92.84	99.20	95.25	91.92	99.23
Tree-GRU V5	95.46	92.85	98.55	95.04	92.29	98.30
Tree-GRU V6	92.68	88.16	98.65	91.88	87.07	98.36

We would also like to explore the different stack architectures like full-stack, semi-stack etc., and analyze the differences it would make on the model.

Acknowledgments

We like to thank the original author of the paper, Dr. Forough Arabshahi (11) for helping us with the updated code and an updated paper to work with, our TAs for providing feedback, and Prof. Chenhao Tan for providing us the opportunity to test our ML skills through this project.

References

[1] Arabshahi, F., Singh, S. and Anandkumar, A., 2018. Combining symbolic expressions and black-box function evaluations for training neural programs. In International Conference on Learning Representations.

[2] Arabshahi, F., Lu, Z. Singh, S. and Anandkumar, A, 2019. Memory augmented recursive neural networks. Memory Augmented Recursive Neural Networks https://arxiv.org/abs/1911.01545

[3] https://en.wikipedia.org/wiki/List_of_trigonometric_identities

[4] https://forougha.github.io/paperPDF/neuralODE.pdf

[5] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondrej Cert, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy symbolic computing in python. PeerJ Computer Science, 3e103, 2017

[6] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), volume 1, pp. 1556–1566, 2015.

[7] Guillaume Lample, François Charton (2019) Deep Learning for Symbolic Mathematics: https://arxiv.org/abs/1912.01412

[8] Zaremba, Wojciech, Karol Kurach, and Rob Fergus. "Learning to discover efficient mathematical identities." In Advances in Neural Information Processing Systems, pp. 1278-1286. 2014.

[9] Kalyan, Ashwin, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. "Neural-guided deductive search for real-time program synthesis from examples." arXiv preprint arXiv:1804.01186 (2018).

[10] Polosukhin, Illia, and Alexander Skidanov. "Neural program search: Solving programming tasks from description and examples." arXiv preprint arXiv:1802.04335 (2018).

[11] https://forougha.github.io/