Recognizing and Verifying Mathematical Equations using Multiplicative Differential Neural Units

Ankur Mali[1], **Alexander G. Ororbia**[2], **Dan Kifer** [1] **and C. Lee Giles**[1]
[1] The Pennsylvania State University, University Park, PA, 16802, USA
[2] Rochester Institute of Technology, Rochester, NY, 14623, USA
{aam35, duk17, clg20}@psu.edu
{ago}@cs.rit.edu

Abstract

Automated mathematical reasoning is a challenging problem that requires an agent to learn algebraic patterns that contain long-range dependencies. Two particular tasks that test this type of reasoning are (1) *mathematical equation verification*, which requires determining whether trigonometric and linear algebraic statements are valid identities or not, and (2) *equation completion*, which entails filling in a blank within an expression to make it true. Solving these tasks with deep learning requires that the neural model learn how to manipulate and compose various algebraic symbols, carrying this ability over to previously unseen expressions. Artificial neural networks, including recurrent networks and transformers, struggle to generalize on these kinds of difficult compositional problems, often exhibiting poor extrapolation performance. In contrast, recursive neural networks (recursive-NNs) are, theoretically, capable of achieving better extrapolation due to their tree-like design but are difficult to optimize as the depth of their underlying tree structure increases. To overcome this issue, we extend recursive-NNs to utilize multiplicative, higher-order synaptic connections and, furthermore, to learn to dynamically control and manipulate an external memory. We argue that this key modification gives the neural system the ability to capture powerful transition functions for each possible input. We demonstrate the effectiveness of our proposed higher-order, memory-augmented recursive-NN models on two challenging mathematical equation tasks, showing improved extrapolation, stable performance, and faster convergence. Our models achieve a 1.53% average improvement over current state-of-the-art methods in equation verification and achieve a 2.22% Top-1 average accuracy and 2.96% Top-5 average accuracy for equation completion.

Introduction

Mathematical reasoning is one problem domain that crucially requires understanding the composition between symbols and arithmetic operators. In demonstrating that it has an "understanding" of basic mathematical and logical concepts, an agent must solve new equations or resolve expressions that might become increasingly more complex with time. This entails understanding the structure of equations, as well as their underlying grammar, in order to properly and effectively extrapolate to unseen examples. With respect to this kind of reasoning, artificial neural networks (ANNs) have been shown to experience great difficulty achieving the robustness, adaptability, and flexibility exhibited by human agents (Fodor, Pylyshyn et al. 1988). Specifically, for tasks requiring the ability to compose knowledge, where complex expressions or structures are created by learning and manipulating rules that permit combination and usage of atomic elements of knowledge (such as the operations of addition or multiplication), ANNs struggle to work correctly and reliably. Indeed, it is often argued that ANNs are incapable of learning to perform the compositional actions needed to mathematically reason (Fodor, Pylyshyn et al. 1988).

Nonetheless, in this paper, we argue that the limitations in an ANN's ability to learn compositionality is due (at least in part) to several limitations in their current structural design. First, ANNs do not possess the right inductive bias (or prior, in the form of structural constraints) that would allow them to more readily and naturally extract the compostionality in various symbolic languages. Second, standard ANNs, even those that are stateful, e.g., recurrent neural networks (RNNs), lack a proper (interpretable) memory structure that would be allow them to properly handle the arrangements of symbols that compose mathematical expressions of increasing depth (complexity) and length. Since mathematical equations are derived from context-free languages related to mathematical identities (Arabshahi, Singh, and Anandkumar 2018), it would make sense to manipulate an external memory when processing equations. For example, a model trained on $(\sqrt{1}\times 1\times y)+x = (1\times y)+x$ should be capable of generally understanding structure that includes equality and inequality. Furthermore, a memory structure would allow a network to better generalize to unseen equations of different depths, since memory can offload some of the memorization that a network typically does using its short-term synapses for (Arabshahi et al. 2019). For instance, a model augmented with external memory should be capable of understanding the following equation (without any need to train it directly on it): $y\times\left(1^1\times(3+(-1\times 4^{0\times 1}))+x^1\right) = y\times 2^0\times(2+x)$.

Ultimately, we aim to better understand the components required to enhance an ANN's ability to mathematically reason and, therefore, we explore two important reasoning tasks: mathematical equation verification (is a stated identity true?) and mathematical equation completion (fill in the

blank to make the equation true). Both of these tasks are designed such that an agent must learn how to combine and arrange symbols and operators in order to properly parse complex mathematical equations like the two provided in the prior paragraph. However, most, if not all, modern-day neural architectures are ill-suited to properly tackle these problems, as evidenced by several recent studies in related tasks that require memory acquisition and knowledge of compositionality. For example, in the domain of semantic parsing, research indicates that as task complexity increases, i.e., sequence length increases and relationships between tokens/items becomes complicated, RNNs fail to generalize to unseen examples (Lake and Baroni 2018). Similarly, poor generalization was observed when RNNs were trained to grammatically infer complex Dyck languages (Mali, Ororbia, and Giles 2019; Mali et al. 2020; Suzgun et al. 2019a) and when neural transformers (Vaswani et al. 2017) were trained to do symbolic integration and solve differential equations (Lample and Charton 2019; Saxton et al. 2019).

This failure to generalize stems from the fact that models like the transformer are theoretically not capable of recognizing complex grammars (Hahn 2020) and, empirically, only perform well when the test set largely comes from the same distribution as the training set (Lample and Charton 2019). In essence, once a distributional shift occurs (test samples begin to vary significantly from training samples), models like the neural transformer break down and fail to extract the functionality of symbols presented to it (let alone learn how to actually compose them). In the realm of mathematical reasoning, such a shift could occur by simply changing the location of an operator (changing its role), or connecting it to different symbols/arguments, or even switching which side of equality it belongs. Modern-day ANNs fail to operate in these kinds of out-of-distribution cases.

Therefore, having the ability to extract and learn relationships between various operators and complex structures/entities/objects (that make up symbolic expressions) would facilitate such out-of-sample generalization when processing mathematical equations. One type of (data) structure that could serve as a useful inductive bias for the ANN is the binary tree, which is a useful way of representing equations. Such trees, upon construction, are interpretable and often inspected when attempting to understand relationships between symbols and operators. When data is available in tree form, recursive neural networks, e.g., tree RNNs, have been shown to outperform standard RNNs (Tai, Socher, and Manning 2015; Socher et al. 2011; Allamanis et al. 2017; Evans et al. 2018; Arabshahi, Singh, and Anandkumar 2018), generalizing well by exploiting their own natural tree-like design (making them suitable for processing equations). However, as the depth of their underlying tree increases, recursive networks struggle to generalize to unseen, longer strings, despite their suitable design. One reason for this is that, as tree depth increases, the credit assignment problem becomes more challenging, hampering the system's learning ability. On the other hand, the lack of a proper error-correction mechanism in the model itself (in the event there is no learning) limits its ability to adapt to novel pattern sequences. In the hopes of endowing recursive networks with some form of error-correction, recent work has attempted to combine differentiable memory with tree-based Long Short Term Memory (LSTM) models (Arabshahi et al. 2019). However, this work is limited since it focuses on adding external memory to a standard neural controller (hoping this alone will improve extrapolation ability), assuming that the original network is already sufficiently good at modeling compositionality instead of addressing the network's potential weaknesses. Our work will challenge this assumption by making the controller more powerful.

To fundamentally resolve the limitations described above, in this work we look to creating more powerful recursive networks by generalizing them to utilize higher-order synaptic weight parameters. Higher-order (or tensor) synaptic connections have been shown, both classically and in recent efforts, to be provably capable of encoding complex grammars, especially in the context of temporal neural models that tackle the challenging task of processing and inferring complex context-free grammars (Omlin and Giles 1996; Mali, Ororbia, and Giles 2019; Stogin et al. 2020). With tensor parameters, there is a straight forward mapping of a state machine into a set of transition rules that can be programmed into an ANN's representations, something which is currently not possible for first-order models (which is what most modern networks actually are). This, we argue, makes higher-order synapses a potentially useful inductive bias for recursive networks, especially since mathematical equations are a kind of context-free grammar. Furthermore, we account for the higher computational cost imposed by using higher-order parameters in an ANN (which has prevented their widespread use in modern-day deep learning) by developing several approximations based on multiplicative interactions. These approximations are used in tandem with a differentiable stack memory, which prior work has shown is important in allowing RNNs to recognize certain classes of grammars (Suzgun et al. 2019b,a; Mali et al. 2020).

The primary contributions of this paper are as follows:

- We introduce the first higher-order recursive recurrent neural network, known as the second-order Tree-RNN.
- We introduce new variant models that approximate second-order Tree-RNNs.
- We introduce new stack-augmented variants of these higher-order recursive recurrent models, beating out state-of-the-art results for two mathematical reasoning tasks.

Background and Notation

In this section, we define the notation to be used in this paper. We denote scalars with non-bold letters (e.g., c), vectors with bold lowercase letters (e.g., $\mathbf{x}$), and matrices with bold uppercase letters (e.g., $\mathbf{W}$). A recursive neural network (recursive-NN) is a kind of tree-structured neural architecture in which each node is represented by an ANN (see Figure 1). Such a design has been empirically shown to capture semantic relationships in symbolic data, allowing the model to better generalize to harder problems in natural language processing (Tai, Socher, and Manning 2015; Arabshahi, Singh, and Anandkumar 2018; Socher et al. 2013b;

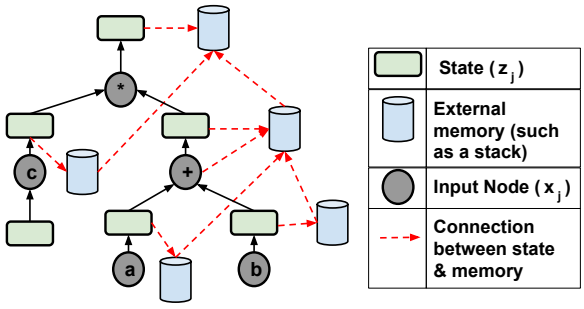


Figure 1: Architecture of the recursive neural network with nodes augmented with external memory. The network shown is operating on the equation $c * (a + b)$ and has arranged its nodes to process the underlying symbol sequence.

Bowman, Potts, and Manning 2015). All the nodes of a rec-RNN have a state denoted by $\mathbf{z}_j \in \mathbb{R}^n$ and an input denoted by $\mathbf{x}_j \in \mathbb{R}^{kn}$ where n is the hidden dimension, k is number of children nodes, N is the number of nodes in the tree, and $j \in [0, N-1]$. The children of node j are denoted as c_{jk}, where k is number of children for node$j$. The hidden representation $\mathbf{z}_j$ depends on the network architecture, acting as memory for the model. For instance, in a simple Tree RNN, state $\mathbf{z}_j$ is computed by passing an input $\mathbf{x}_j$ through a feedforward network. However, since recurrence is absent in this memory node, we believe that the model is limited in its ability to transfer knowledge across time steps. Furthermore, other factors we argue hinder model performance are: 1) the absence of a better input encoding, and 2) the fact that neurons in the Tree RNN are trying to both model compositionality and extrapolate across large sequences, creating a harder learning problem than needed. To address these issues, we will develop several novel variations of the Tree RNN that integrate recurrent second order synapses (or approximations thereof) and external (stack) memory.

Second-Order Tree Recurrent Networks

In this section, we describe our second (2^{nd})-order Tree RNN architecture. This is a tree structure where every node j contains a 2^{nd}-order RNN. In a 1^{st}-order RNN, including popular variants such as the Long Short-Term Memory LSTM (Hochreiter and Schmidhuber 1997), the hidden state $\mathbf{z}_j^{t-1}$ of node j at time $t-1$ is combined with its input $\vec{x}_j$ to produce the hidden state at time t: $\mathbf{z}_j^t = f(W_1 \mathbf{z}_j^{t-1} + W_2 \mathbf{x}_j)$, where f is an activation function. In contrast, a 2^{nd}-order RNN uses a third-order weight tensor $\mathbf{W}$ to update the state (Giles et al. 1990; Omlin and Giles 1996) as follows:

$$\mathbf{z}_j^t[\ell] = f\left(\sum_{i_1, i_2} \mathbf{W}[\ell, i_1, i_2] \mathbf{z}_j^{t-1}[i_1] \mathbf{x}_j[i_2] + \mathbf{b}_j[\ell]\right). \quad (1)$$

This tensor operation can also be written as $\mathbf{z}_j^t = f\left(\mathbf{W}(\mathbf{z}_j^{t-1}, \mathbf{x}_j) + \mathbf{b}_j\right)$. The benefit of this update style is that it concisely captures the state transitions of a deterministic finite automata (DFA). For example, if $\mathbf{x}_j$ is a one-hot encoding of an input symbol (say i_1, with a vocabulary size V) and $\mathbf{z}_j$ is a one-hot encoding of an automaton state (say i_2), then $\mathbf{W}[:, i_1, i_2]$ can be set to be the one-hot encoding of the state that the DFA should transition to. Thus, the 2^{nd}-order RNN architecture represents a differentiable transition function over a distributed state and input representation (Giles et al. 1990; Mali, Ororbia, and Giles 2019). Tensor operations have also been experimented with ANNs containing feedforward tensor weights (Socher et al. 2013a), yielding promising results on logical reasoning data, outperforming first-order networks (Bowman 2013; Socher et al. 2013a). In contrast to these related efforts, our work explores tensor recurrent weights combined with recursive connections.

To get a 2^{nd}-order Tree RNN, we set each node j to be a 2^{nd}-order RNN which takes in as input $\mathbf{x}_j$ a concatenation of its k children. Hence, the tensor weight matrix $\mathbf{W}$ has dimensions $n \times n \times kn$. Note that we also set activation f to be the hyperbolic tangent. To the best of our knowledge, this work is the first to propose a recursive-NN (tree) model built with higher order weights, which, as we will show, is more powerful than a simple Tree-RNN (particularly given the fact that a Tree-RNN uses feedforward ANNs to parameterize its nodes and hence does not exploit recurrence well).

To effectively handle the processing of equations, we further incorporate hidden-to-output weight transition matrices. One can either share weights or have node-specific weight parameters in the architectures. In the case of equations, we assign a separate set of parameters to each unique operator that could be used to define an equation, allowing the rec-RNN to learn relationships across these operators. When an operator or variable is duplicated, i.e., it occurs more than once in an equation symbol stream, we share the parameters across the occurrences.

Multiplicative Tree LSTM

Despite having theoretical justification and better expressivity, 2^{nd} order networks are computationally expensive, scaling poorly to large datasets due to their higher-order synapses, which create a challenging optimization problem over a large computation graph. To overcome this limitation , we take inspiration from recent success in approximating higher order synaptic connections. Specifically, we look to RNNs that use multiplicative weights, which shown promise in tasks such as grammatical inference (Mali et al. 2020) and sequence modeling (Wu et al. 2016; Krause et al. 2016). Multiplicative synapses are inspired by 2^{nd} order connections (Giles et al. 1990; Giles and Omlin 1993; Sun, Giles, and Chen 1998; Rabusseau, Li, and Precup 2019) and aim to approximate their computation, having also been shown to provably encode complex grammatical structure (Omlin and Giles 1996; Mali, Ororbia, and Giles 2019) (yielding more interpretable RNNs that generalize well). A multiplicative network introduces an intermediate state ($\mathbf{m}$) that creates a flexible input-output transition that also endows the RNN with implicit error handling and improved generative ability (Wu et al. 2016).Based on this, we design a novel recursive-NN that employs multiplicative synapses instead of second-order tensor weights as a form of memory. Specifically, we propose the multiplicative Tree Long Short-Term Memory (MTree-LSTM) (generalizing the tree-LSTM (Tai,

Socher, and Manning 2015)), a type of recursive-NN with nodes made up of multiplicative weights that allow for different transition matrices for each possible input (much like second-order synapses do).

The above multiplicative memory structure is coupled with LSTM gates to actively process sequences containing long-range dependencies. For an N-ary MTree-LSTM , the branching factor of the model's underlying structure is at most N, assuming that children nodes are ordered, *i.e.*, we can index them from 1 to N. For any node j, the hidden state and memory cell of the nth child would then be represented as $\mathbf{z}_{jn}$ and $\mathbf{c}_{jn}$ respectively.

The resulting N-ary MTree-LSTM transition equations, following the description above, would then be:

$$\hat{\mathbf{z}}_n = W^{(m)}\mathbf{x}_n + R_z\mathbf{z}_n^{t-1}) \tag{2}$$

$$\mathbf{m}_n = (W^{(m)}\mathbf{x}_n) + (R_m\hat{\mathbf{z}}_n) \tag{3}$$

$$\mathbf{i}_j = \sigma\left(W^{(i)}\mathbf{x}_j + \sum_{n=1}^{N} U_n^{(i)}\mathbf{m}_n + b^{(i)}\right), \tag{4}$$

$$\mathbf{f}_{jn} = \sigma\left(W^{(f)}\mathbf{x}_j + \sum_{n=1}^{N} U_n^{(f)}\mathbf{m}_n + b^{(f)}\right), \tag{5}$$

$$\mathbf{o}_j = \sigma\left(W^{(o)}\mathbf{x}_j + \sum_{n=1}^{N} U_n^{(o)}\mathbf{m}_n + b^{(o)}\right), \tag{6}$$

$$\mathbf{u}_j = \tanh\left(W^{(u)}\mathbf{x}_j + \sum_{n=1}^{N} U_n^{(u)}\mathbf{m}_n + b^{(u)}\right), \tag{7}$$

$$\mathbf{c}_j = \mathbf{i}_j \odot \mathbf{u}_j + \sum_{n=1}^{N} \mathbf{f}_{jn} \odot \mathbf{c}_{jn}, \tag{8}$$

$$\mathbf{z}_j = \mathbf{o}_j \odot \tanh(\mathbf{c}_j), \tag{9}$$

where all input-to-hidden synapses $\mathbf{W}$ are matrices of shape $\mathbb{R}^{n\times 2n}$ and hidden-to-hidden synapses $\mathbf{U}$ are matrices of shape $\mathbb{R}^{2n\times 2n}$. In our experiments, note that $m = z$. Intuitively, we can interpret each parameter matrix as higher-order synapses that encode the correlation between component vectors of the multiplicative unit, the input x_j, and the hidden state z_k (obtained from children nodes).

Multiplicative-Integration Tree LSTMs

Much in the same spirit as the MTree-LSTM, in the effort to overcome limitations of 2^{nd} order RNNs, we design a multiplicative-integration Tree-LSTM (MI Tree-LSTM), extending the Tree-LSTM (Tai, Socher, and Manning 2015) with nodes that are a first-order approximation of 2^{nd} order connections (integrated with LSTM cells). This generally amounts to replacing the addition operation ($+$) with a Hadamard product ($\odot$) in the standard Elman-RNN equation. This element-wise multiplication has been argued to perform a rank-1 approximation of the operation carried by a 2^{nd} order connection. These have been shown to be useful in several sequence modeling settings (Krause et al. 2016).

For an N-ary MI Tree-LSTM, the branching factor is at most N (again, children nodes are ordered). For any node j, the hidden state and memory cell of the nth child is represented as z_{jn} and c_{jn} respectively. Formally, the N-ary MI Tree-LSTM transition equations are:

$$\mathbf{i}_j = \sigma\left(W^{(i)}\mathbf{x}_j \odot \sum_{n=1}^{N} U_n^{(i)}\mathbf{z}_{jn} + b^{(i)}\right), \tag{10}$$

$$\mathbf{f}_{jn} = \sigma\left(W^{(f)}\mathbf{x}_j \odot \sum_{n=1}^{N} U_n^{(f)}\mathbf{z}_{jn} + b^{(f)}\right), \tag{11}$$

$$\mathbf{o}_j = \sigma\left(W^{(o)}\mathbf{x}_j \odot \sum_{n=1}^{N} U_n^{(o)}\mathbf{z}_{jn} + b^{(o)}\right), \tag{12}$$

$$\mathbf{u}_j = \tanh\left(W^{(u)}\mathbf{x}_j \odot \sum_{n=1}^{N} U_n^{(u)}\mathbf{z}_{jn} + b^{(u)}\right), \tag{13}$$

$$\mathbf{c}_j = \mathbf{i}_j \odot \mathbf{u}_j + \sum_{n=1}^{N} \mathbf{f}_{jn} \odot \mathbf{c}_{jn}, \tag{14}$$

$$\mathbf{z}_j = \mathbf{o}_j \odot \tanh(\mathbf{c}_j), \tag{15}$$

where all the input-to-hidden weights $\mathbf{W}$ and hidden-to-hidden weights $\mathbf{U}$ are matrices in $\mathbb{R}^{n\times 2n}$.

Note that for both the MTree-LSTM and MITree-LSTM models, the memory cell $\mathbf{c}_j$ is a one-dimensional vector. This cell vector could alternatively be enhanced by instead substituting it with an entire differentiable data/memory structure such as a stack. It is this special situation that we will explore in the next section, where we explicate how to augment our recursive networks with external memory to further increase capacity and improve generalization.

Stack-Augmented Tree-RNNs

All of the previously proposed rec-RNN models can be extended to make use of an external, differentiable stack as a means to increase memory capacity. This extension follows in the same spirit as prior research in integrating data structures that improve the generalization ability of RNN models for various sequence processing tasks (Joulin and Mikolov 2015; Mali et al. 2020; Suzgun et al. 2019b) (though this focused on traditional, first-order RNNs). In our rec-RNN models, each node j is augmented with an external stack $\mathbf{S}_j \in \mathbb{R}^{p\times n}$, where p is the stack size/length.

A stack is a last-in-first-out (LIFO) data structure that an ANN can only interact with by manipulating the structure's top data storage slot. Stacks are often claimed to be more interpretable in nature but, more importantly, they crucially align with formal language theory. Specifically, one key result from theory is that models that make use of two stacks are Turing complete (Hopcroft, Motwani, and Ullman 2006), meaning that a stack is indispensable when learning context-free languages. When working with mathematical equations, we wish to exploit the computational learning capability that comes with allowing an ANN to control a stack memory, especially given the fact that long-range dependencies are created when extracting structure from equations.

The top of the stack is denoted by $\mathbf{S}_j[0] \in \mathbb{R}^n$. The stack has two primary operations – pop and push. Integrating a stack means we are integrating a push-down automation into

our network. Specifically, the network will use a 2D action vector $\mathbf{a}_j \in \mathbb{R}^2$ whose elements represent the push and pop operations for interacting with the stack. These two actions are controlled by the network's state at each node:

$$\mathbf{a}_j = \phi(\mathbf{A}_j \mathbf{z}_j + \mathbf{b}_j^{(a)}) \tag{16}$$

where $\mathbf{A}_j \in \mathbb{R}^{2 \times 2n}$ and ϕ is the softmax function. We denote the probability of the action "push" with the label *actionPush* $= \mathbf{a}_j[0] \in [0, 1]$ and "pop" with *actionPop* $= \mathbf{a}_j[1] \in [0, 1]$. Note that since the softmax nonlinearity has been used – these two probabilities must sum to 1 to create a valid distribution over stack actions.

In order to calculate the next hidden state of our model, we employ the following set of equations:

$$\hat{\mathbf{z}}_\mathbf{j} = \mathbf{z}_j^{t-1} + \sum_{n=1}^{N} P\mathbf{S}_n^{0,t-1}, \quad \mathbf{z}_j = f_1(\mathbf{W}_j \mathbf{x}_j \odot R\hat{\mathbf{z}}_j^{t-1}) \tag{17}$$

where we see that a state update for node j is a summation of the top slots of each child node's corresponding stack. We assume that the top of a stack is located at index 0. with value $\mathbf{S}_n[0]$, via the following:

$$\mathbf{S}_n[0] = \mathbf{a}_j[PUSH]\sigma(D\mathbf{z}_j) + \mathbf{a}_j[POP]\mathbf{S}_n^{t-1}[1] + \mathbf{a}_j[NoOP]\mathbf{S}_n^{t-1}[0] \tag{18}$$

where the symbols PUSH, POP, and NoOP correspond to the unique integer indices 0, 1, and 2 that access the specific action value in their respective slot. D is a $1 \times m$ matrix and $\sigma(v) = 1/(1 + exp(-v))$ is the logistic sigmoid. If $\mathbf{a}_j[PUSH] = 1$, we add the element to the top of the stack and if $\mathbf{a}_j[POP] = 1$, we remove the element at top of the stack (and shift/move the stack upwards). Similarly, for the elements stored at depth $i > 0$ in the stack, the following rule must be followed:

$$\mathbf{S}_{c_j}[i] = \mathbf{a}_t[PUSH]\mathbf{S}_{c_j}^{t-1}[i-1] + \mathbf{a}_t[POP]\mathbf{S}_{c_j}^{t-1}[i+1]. \tag{19}$$

Note that for more complex hidden state functions, the state calculation and stack integration is identical to the description in this section. When the above stack is integrated into our previous two models, we obtain the novel variants we call the M-Tree-LSTM+stack and the MITree-LSTM+stack.

Mathematical Reasoning over Equations

In this section, we discuss the tasks investigated, presenting the datasets used for evaluation. Model performance was measured on two challenging benchmark tasks, i.e., mathematical equation verification and equation completion, introduced in (Arabshahi, Singh, and Anandkumar 2018).

For both tasks investigated, we generated $41,894$ equations of various depths. To create the training/validation/testing splits for this problem, we generate new mathematical identities by performing local random changes to known identities, starting with the 140 axioms provided by (Arabshahi, Singh, and Anandkumar 2018). These changes resulted in identities of similar or higher complexity (equal or larger depth), which may be correct or incorrect and are valid expressions within the grammar (CFGs). Models were trained on equations of depths 1 through 7 and then tested on equations of depths 8 through 13. The data creation process was identical to the one proposed in (Arabshahi et al. 2019). Table **??** provides the statistics of the generated samples, showing the number of equations available at each parse tree depth.

Data Creation: For the equation completion task, we evaluate each model's ability to predict the missing pieces of an equation such that the overall mathematical expression condition holds true. For this experiment, we utilize the same model(s) and baselines used for the mathematical completion task. To create evaluation data, we take all of the generated test equations and randomly choose a node at depth k (k is between 1 to 13) in each and every equation. Next, we replace this with all possible configurations for (problem) depth 1 through 13 generated using context-free grammars (CFGs) or related generative grammars as suggested by (Arabshahi, Singh, and Anandkumar 2018).

Once created, we present this new set of equations to each model/baseline and measure its Top-1 and Top-5 accuracy (these are reported in the plots found in the main paper). Top-1 and Top-5 rankings serve as a proxy for each model's confidence when predicting the blank in the mathematical expression (helping us to observe further if the model ensures the correctness of a target expression/equation). Sample equations/expressions from the generated datasets are shown in Table 6 along with the truth label (the gold standard) and accompanying problem (recursion) depth (which, in turn, serves as a proxy measure of problem difficulty).

Mathematical Equation Verification: In this task, the agent is to process symbolic and numerical mathematical equations from trigonometry and linear algebra. The goal of an agent should be to successfully learn the relationships between these equations and, ultimately, verify their correctness. Verification can be likened to a binary classification problem. Note that we take the equation's parse tree depth as a metric to evaluate hardness of the math problem.

Mathematical Equation Completion: The goal of this task is to predict the missing pieces in a given mathematical equation such that the final outcome holds true (and is mathematically valid). Models trained for equation verification are used to conduct equation completion.

Experiments

Baseline Models: Below we list all of the baselines that we compared to our proposed rec-RNN models, for both of our experimental tasks. The baseline models we compared against were: **Majority Class**: this baseline is a classification approach that always predicts the majority class. **LSTM**: this baseline is a Long Short-Term Memory network (Hochreiter and Schmidhuber 1997). **Tree-LSTM**: this baseline is the original Tree-LSTM network (Tai, Socher, and Manning 2015), where each node is a standard LSTM cell. **Tree-SMU**: this baseline is the standard recursive neural network coupled with a stack (Arabshahi et al. 2019). All of the recursive models (both baselines and

proposed models) share parameters whenever the functionality of a given node is similar (in the context of an equation's parse tree). For equation verification, all models optimize the Categorical log likelihood at the output (which is the root node). The root of the model represents equality and performs a dot product of the output embedding of the right and left sub-trees/nodes. For equation completion, all models are optimized to minimize the cross-entropy loss at the output (the root or equality node). The input to the recursive networks includes the terminal nodes (leaves) of the equations – these terminals consist of symbols (representing variables in the equation) and numbers. The leaves of the recursive networks are embedding layers which encode the symbols and numbers of the equation.

Evaluation Metrics: For the task of equation verification, we report model performance after measuring accuracy, precision, and recall. We report each metric measurement as a percentage in Table 2 and 3 and they are abbreviated as "Acc", "Prec", and "Rcl" for accuracy, precision, and recall, respectively. On the other hand, for equation completion, we report Top-K accuracy (Arabshahi, Singh, and Anandkumar 2018). Measuring Top-K accuracy is often a useful performance metric since it accounts for the percentage of samples for which any given model predicts correctly on at least one correct sample for any given blank.

Implementation Details

All of the models experimented with in this paper were implemented using the PyTorch Python framework (Paszke et al. 2019). Models were optimized using back-propagation of errors to calculate parameter gradients and were updated using the Adam (Kingma and Ba 2014) adaptive learning rate, with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and by starting its global learning rate at $\lambda = 0.1$ and then employing a patience scheduling that divided this rate by half whenever there was no improvement observed on the validation set. We regularized the models with a weight decay of 0.00002.

The number of neurons in each model's hidden layer as well as the drop-out rate were tuned using a coarse grid search, i.e., hidden layer size was searched over the array $[8, 15, 25, 30, 40, 45, 50, 55, 60, 80, 100]$ and the drop-out rate was searched over the array $[0.1, 0.2, 0.3]$. Parameter gradients were estimated over mini-batches of size 50 for all experiments. We ran all models using 10 different seeds and report the 10-trial average and standard deviation of the results. Models were trained for a maximum of 500 epochs or until convergence was reached, i.e., early stopping was used. The best accuracy of the model is reported when it reaches its best measured performance on the validation set.

Ablation Study

To complement our results, we provide a small ablation study to demonstrate that the removal of the higher-order synaptic connections leads to a degradation in model performance of the model. This means that the complexity of the synaptic parameters matters in order to achieve consistent performance. As shown in Table 5, we compare recursive neural models with and without higher order synapses. For all the models trained without higher-order weights, the standard LSTM cell was used as memory. Again, we observe (in tandem with the results presented above) that utilizing second-order weights help yield stable results across problem recursion depths.

Second, it is important to test whether or not the higher-order parameters offer any computational advantage. To test this, we conducted an experimental analysis of each model's convergence performance (including baselines and proposed

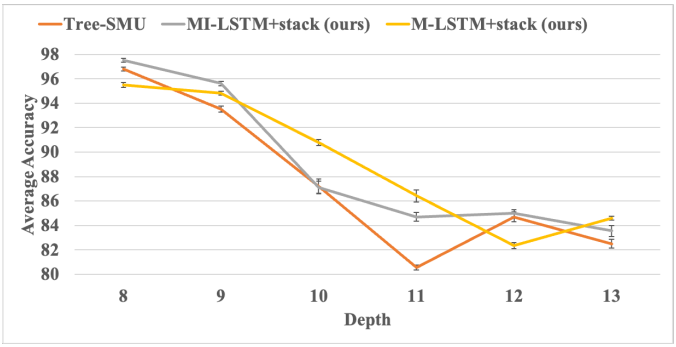
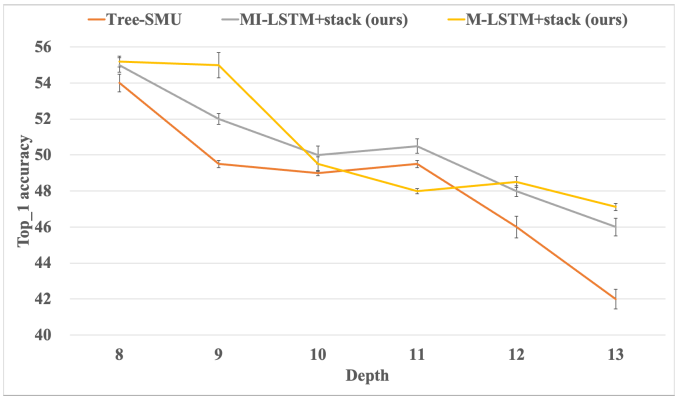
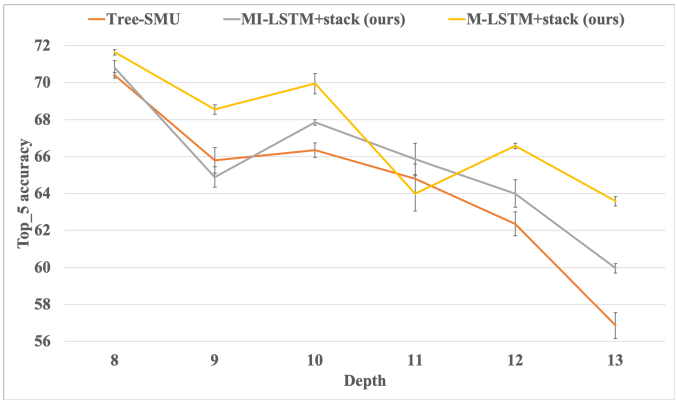


Figure 2: Average accuracy metrics breakdown in terms of performance on the test data depth for equation verification.



(a) Top 1 Accuracy



(b) Top 5 Accuracy

Figure 3: Top-K accuracy metrics breakdown in terms of the test data depth for the equation completion task.

	all	1	2	3	4	5	6	7	8	9	10	11	12	13
Number of equations	41,894	21	355	2,542	7,508	9,442	7,957	6,146	3,634	1,999	1,124	677	300	189
Correct classes	0.56	0.52	0.57	0.62	0.61	0.58	0.56	0.54	0.52	0.52	0.49	0.50	0.50	0.50

Table 1: Dataset statistics for the equation sequence dataset (sorted by data depth).

Approach	**Train (Depths 1-7)**			**validation (Depths 1-7)**		
	Acc	**Prec**	**Rcl**	**Acc**	**Prec**	**Rcl**
Majority Class	58.12	-	-	56.67	-	-
RNN (Arabshahi et al. 2019)	68.50	69.61	81.27	65.77±0.44	65.69±0.72	83.06±1.24
LSTM (Arabshahi et al. 2019)	90.03	87.02	97.37	85.47±0.27	81.97±0.38	95.32±0.17
Stack-RNN	92.03	86.42	98.43	83.47±0.35	84.58±0.32	95.00±0.27
Tree-RNN (Arabshahi et al. 2019)	94.98	94.25	97.29	89.27 ± 0.04	87.8 ± 0.39	94.16 ± 0.61
Tree-LSTM (Arabshahi et al. 2019)	98.51	97.67	99.83	93.77 ± 0.02	90.92 ± 0.08	98.88 ± 0.08
Tree-SMU	96.71	95.12	99.09	92.59 ± 0.03	90.55 ± 0.17	98.07 ± 0.18
Tree-SMU - no-op	98.02	97.07	98.99	93.58 ± 0.21	91.44 ± 0.23	98.11 ± 0.09
Tree-SMU - no-op - normalize	97.97	96.57	99.79	93.21 ± 0.20	90.29 ± 0.17	98.30 ± 0.12
2^{nd} order Tree-RNN (**ours**)	95.62	94.58	98.25	90.28 ± 0.07	90.25 ± 0.35	98.08 ± 0.20
MI-Tree-LSTM (**ours**)	98.80	98.01	99.80	94.20 ± 0.02	91.50 ± 0.07	98.99 ± 0.06
MTree-LSTM (**ours**)	98.25	97.00	98.81	94.09 ± 0.02	91.00 ± 0.05	99.00 ± 0.06
MI-Tree-LSTM + stack (**ours**)	96.28	96.00	99.25	93.99 ± 0.04	91.55 ± 0.15	98.57 ± 0.09
M-Tree-LSTM + stack (**ours**)	97.35	97.15	99.20	94.29 ± 0.02	90.99 ± 0.04	98.28 ± 0.09

Table 2: Overall performance of the models on train and validation datasets for the equation verification task.

Approach	**Train (Depths 1-7)**			**Test (Depths 8-13)**		
	Acc	**Prec**	**Rcl**	**Acc**	**Prec**	**Rcl**
Majority Class	58.12	-	-	51.71	-	-
RNN (Arabshahi et al. 2019)	68.50	69.61	81.27	55.5±0.25	55.85±0.61	67.32±3.62
LSTM (Arabshahi et al. 2019)	90.03	87.02	97.37	73.09±0.64	73.92±1.48	74.34±1.53
Stack-RNN	92.03	86.42	98.43	73.09±0.64	73.92±1.48	74.34±1.53
Tree-RNN (Arabshahi et al. 2019)	94.98	94.25	97.29	81.82 ± 0.12	82.66 ± 0.55	82.08 ± 0.55
Tree-LSTM (Arabshahi et al. 2019)	98.51	97.67	99.83	86.8 ± 0.6	83.68 ± 0.63	92.54 ± 0.76
Tree-SMU	96.71	95.12	99.09	87.51 ± 0.49	84.00 ± 0.31	94.21 ± 0.62
Tree-SMU - no-op	98.02	97.07	98.99	87.08 ± 0.15	84.32 ± 0.52	92.51 ± 0.51
Tree-SMU - no-op - normalize	97.97	96.57	99.79	87.01 ± 0.50	83.01 ± 0.62	93.77 ± 0.49
2^{nd} order Tree-RNN (**ours**)	95.62	94.58	98.25	86.05 ± 0.11	84.05 ± 0.30	88.99 ± 0.30
MI-Tree-LSTM (**ours**)	98.80	98.01	99.80	87.80 ± 0.7	84.08 ± 0.45	93.00 ± 0.40
MTree-LSTM (**ours**)	98.25	97.00	98.81	87.25 ± 0.5	84.29 ± 0.25	93.02 ± 0.35
MI-Tree-LSTM + stack (**ours**)	96.28	96.00	99.25	88.25 ± 0.81	84.41 ± 0.39	$\mathbf{94.47 \pm 0.61}$
M-Tree-LSTM + stack (**ours**)	97.35	97.15	99.20	$\mathbf{89.04 \pm 0.54}$	$\mathbf{84.59 \pm 0.47}$	94.39 ± 0.59

Table 3: Overall performance of the models on train and test datasets for the equation verification task.

architectures). Table 4 contains the results of this model convergence analysis. All models/baselines were trained 10 times – we report the number of epochs that each model required in order to reach best its measured validation accuracy. From the table, it is encouraging to see that the higher-order synapses do indeed result in faster convergence without affecting performance.

Finally, we present extended results in Table 2 and 3 where the performance of standard RNNs, with and without an external stack memory, are compared to the various re-

Models	2nd Order	1st Order
MTree-LSTM	87.25	84.59
MITree-LSTM	87.80	84.59
MI-Tree-LSTM+ stack	88.25	86.51
M-Tree-LSTM + stack	89.04	86.88

Table 4: Average number of epochs (& validation error reached) required to reach convergence when optimized.

Models	Average Epochs	Valid Error
TreeLSTM	135	93.77 ± 0.02
Tree-SMU	99	93.58 ± 0.21
MTree-LSTM	102	94.09 ± 0.02
MI-Tree-LSTM+ stack	95	93.99 ± 0.04
M-Tree-LSTM + stack	91	94.29 ± 0.02
2^{nd} order Tree-RNN	98	90.28 ± 0.07

Table 5: Model test performance (accuracy) with (2nd Order) and without higher order (1st Order) synapses.

cursive networks experimented with earlier. This highlights how difficult the mathematical reasoning task is in general. As discussed before, a recursive structure is quite necessary to solve such complex tasks/problems.

Results and Discussion

Equation Verification: In Figure 2, we compare our best model with the recently proposed state-of-the-art model in (Arabshahi et al. 2019). Of the five proposed models, as shown in Figure 2, the M-Tree-LSTM+stack and the MI-Tree-LSTM+stack reach the best performance and are stable across equation examples (as the depth of the test set is increased). In Table 2 and 3, we provide a more extensive performance comparison across all models, baselines, and the proposed rec-RNNs on the equation verification task.

Equation Completion: To further test the generalization ability of the various models studied in this paper, we next turned our attention to the equation completion task. In Figures 3a and 3b, we report the top-1 and top-5 accuracy measurements of various recursive models. Notably, observe that the performance of the proposed higher-order rec-RNNs is consistently better than that reported in prior work for both models with and without differentiable memory. The performance of standard RNN models on this task was quite poor as evident in table 2 and 3. This strongly demonstrates that just simple (first-order) recurrent architectures fail to capture much, if any, useful compositional knowledge underlying mathematical equations.

Related Work

The mathematical reasoning problems considered in this work are examples of neural programming, or a task family that requires an ANN to learn (complex) structures such as programs, mathematical equations, and logic from data (Allamanis et al. 2017; Evans et al. 2018; Graves, Wayne, and Danihelka 2014; Zaremba, Kurach, and Fergus 2014; Reed and De Freitas 2015; Cai, Shin, and Song 2017; Saxton et al. 2019). Neural programming tasks are a key application for testing an ANN's ability to extrapolate and compose elements of knowledge. Grammatical inference has also often served as another kind of neural programming problem that challenges an RNN's ability to extract useful hierarchical representations over symbolic sequences, especially as the grammar complexity increases (Suzgun et al. 2019b,a; Mali et al. 2020).

Recursive networks, which theoretically could prove invaluable for neural programming, have been used to model a wide of compositional data types across many applications (all of which contain an inherent hierarchy nested in the data) (Hupkes, Veldhoen, and Zuidema 2018; Hupkes et al. 2020), e.g., natural scene classification (Socher et al. 2011), sentiment classification, semantic relatedness/syntactic parsing (Tai, Socher, and Manning 2015; Socher et al. 2011, 2013a,b; Bowman 2013; Bowman, Potts, and Manning 2015), and neural programming and logic (Allamanis et al. 2017; Zaremba, Kurach, and Fergus 2014; Evans et al. 2018). While a great deal of recent research has strived to integrate differentiable memory into standard RNNs (Graves, Wayne, and Danihelka 2014; Weston, Chopra, and Bordes 2014; Grefenstette et al. 2015; Joulin and Mikolov 2015; Mali et al. 2020; Weston, Chopra, and Bordes 2014; Sukhbaatar et al. 2015; Graves, Wayne, and Danihelka 2014; Graves et al. 2016; Reed and De Freitas 2015; Cai, Shin, and Song 2017; Graves, Wayne, and Danihelka 2014; Das, Giles, and Sun 1992; Kumar et al. 2016; Sun et al. 2017; Mali, Ororbia, and Giles 2019; Mali et al. 2020), even based on the theoretical grounding of formal language (Hopcroft, Motwani, and Ullman 2006), far less work exists on integrating external memory into recursive networks. One notable effort was made in (Arabshahi et al. 2019), where a recursive network was combined with a stack and LSTM gates.

Despite the amount of effort that has been spent on augmenting RNNs with memory, to the best of our knowledge, there has been no attempt at designing and generalizing external memory as a means to increase the compositionality abilities of ANNs, especially with the goal of design recursive networks with recurrent weights that better extrapolate to harder problem instances that are often out-of-sample. Another different, yet related, line of work is that on graph memory networks and tree memory networks (Pham, Tran, and Venkatesh 2018; Fernando et al. 2018) – however, while powerful models, this differs from our work given that these studies do not investigate the value of higher order connections, approach memory construction differently, and ultimately examine different applications.

Conclusion

In this paper, we proposed five novel types of recursive recurrent neural networks, which we have shown are useful for modeling compositional data, specifically for processing mathematical equations and expressions. We demonstrated that the performance of most kinds of recurrent networks degrades significantly when the depth or complexity of an input equation increases. By generalizing recursive neural net-

Example	Label	Depth
$(\sqrt{1} \times 1 \times y) + x = (1 \times y) + x$	Correct	4
$\sec(x + \pi) = (-1 \times \sec(\sec(x)))$	Incorrect	4
$y \times \Big(1^1 \times (3 + (-1 \times 4^{0 \times 1})) + x^1\Big) = y \times 2^0 \times (2 + x)$	Correct	8
$\sqrt{1 + (-1 \times (\cos(y + x))^{\sqrt{\csc(2)}})} \times (\cos(y + x))^{-1} = \tan(y^1 + x)$	Incorrect	8
$2^{-1} + \Big(-\frac{1}{2} \times \big(-1 \times \sqrt{1 + (-1 \times \sin^2(\sqrt{4} \times (\pi + (x \times -1))))}\big)\Big) + \cos^{\sqrt{4}}(x) = 1$	Correct	13
$\big(\cos(y^1 + x) + z\big)^w = \big(\cos(x) \times \cos(0 + y) + \big(-1 \times \sqrt{1 + -1 \times \cos^2(y + 2\pi)}\ \big) \times \sin(x) + z\big)^w$	Correct	13
$\sin\Big(\sqrt{4}^{\ -1}\pi + (-1 \times \sec\big(\csc^2(x)^{-1} + \sin^2(1 + (-1 \times 1) + x + 2^{-1}\pi)\big) \times x)\Big) = \cos(0 + x)$	Incorrect	13

Table 6: Examples of generated equations used in the paper's experiments (Arabshahi et al. 2019).

works to use higher-order synaptic connections and to interactively manipulate a stack memory, we designed agents that are capable of acquiring rich, compositional representations of mathematical equations, allowing for out-of-sample generalization. More importantly, our work demonstrates that higher-order recursive models consistently achieve stable and overall better performance compared to state-of-the-art baselines for two important mathematical reasoning tasks.

References

Allamanis, M.; Chanthirasegaran, P.; Kohli, P.; and Sutton, C. 2017. Learning continuous semantic representations of symbolic expressions. In *International Conference on Machine Learning*, 80–88. PMLR.

Arabshahi, F.; Lu, Z.; Singh, S.; and Anandkumar, A. 2019. Memory Augmented Recursive Neural Networks. *arXiv preprint arXiv:1911.01545* .

Arabshahi, F.; Singh, S.; and Anandkumar, A. 2018. Combining symbolic expressions and black-box function evaluations in neural programs. *arXiv preprint arXiv:1801.04342* .

Bowman, S.; Potts, C.; and Manning, C. D. 2015. Recursive neural networks can learn logical semantics. In *Proceedings of the 3rd workshop on continuous vector space models and their compositionality*, 12–21.

Bowman, S. R. 2013. Can recursive neural tensor networks learn logical reasoning? *arXiv preprint arXiv:1312.6192* .

Cai, J.; Shin, R.; and Song, D. 2017. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611* .

Das, S.; Giles, C. L.; and Sun, G.-Z. 1992. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*, 14.

Evans, R.; Saxton, D.; Amos, D.; Kohli, P.; and Grefenstette, E. 2018. Can neural networks understand logical entailment? *arXiv preprint arXiv:1802.08535* .

Fernando, T.; Denman, S.; McFadyen, A.; Sridharan, S.; and Fookes, C. 2018. Tree memory networks for modelling long-term temporal dependencies. *Neurocomputing* 304: 64–81.

Fodor, J. A.; Pylyshyn, Z. W.; et al. 1988. Connectionism and cognitive architecture: A critical analysis. *Cognition* 28(1-2): 3–71.

Giles, C. L.; and Omlin, C. W. 1993. Extraction, insertion and refinement of symbolic rules in dynamically driven recurrent neural networks. *Connection Science* 5(3-4): 307–337.

Giles, C. L.; Sun, G.-Z.; Chen, H.-H.; Lee, Y.-C.; and Chen, D. 1990. Higher order recurrent networks and grammatical inference. In *Advances in neural information processing systems*, 380–387.

Graves, A.; Wayne, G.; and Danihelka, I. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401* .

Graves, A.; Wayne, G.; Reynolds, M.; Harley, T.; Danihelka, I.; Grabska-Barwińska, A.; Colmenarejo, S. G.; Grefenstette, E.; Ramalho, T.; Agapiou, J.; et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538(7626): 471.

Grefenstette, E.; Hermann, K. M.; Suleyman, M.; and Blunsom, P. 2015. Learning to transduce with unbounded memory. In *Advances in neural information processing systems*, 1828–1836.

Hahn, M. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics* 8: 156–171.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8): 1735–1780.

Hopcroft, J. E.; Motwani, R.; and Ullman, J. D. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Pearson. ISBN 0321455363.

Hupkes, D.; Dankers, V.; Mul, M.; and Bruni, E. 2020. Compositionality Decomposed: How do Neural Networks Gen-

eralise? *Journal of Artificial Intelligence Research* 67: 757–795.

Hupkes, D.; Veldhoen, S.; and Zuidema, W. 2018. Visualisation and'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure. *Journal of Artificial Intelligence Research* 61: 907–926.

Joulin, A.; and Mikolov, T. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, 190–198.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

Krause, B.; Lu, L.; Murray, I.; and Renals, S. 2016. Multiplicative LSTM for sequence modelling. *arXiv preprint arXiv:1609.07959* .

Kumar, A.; Irsoy, O.; Ondruska, P.; Iyyer, M.; Bradbury, J.; Gulrajani, I.; Zhong, V.; Paulus, R.; and Socher, R. 2016. Ask me anything: Dynamic memory networks for natural language processing. In *International conference on machine learning*, 1378–1387.

Lake, B.; and Baroni, M. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning*, 2873–2882. PMLR.

Lample, G.; and Charton, F. 2019. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412* .

Mali, A.; Ororbia, A.; and Giles, C. L. 2019. The Neural State Pushdown Automata. *arXiv preprint arXiv:1909.05233* .

Mali, A.; Ororbia, A.; Kifer, D.; and Giles, C. L. 2020. Recognizing Long Grammatical Sequences Using Recurrent Networks Augmented With An External Differentiable Stack. *arXiv preprint arXiv:2004.07623* .

Omlin, C. W.; and Giles, C. L. 1996. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM (JACM)* 43(6): 937–972.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, 8026–8037.

Pham, T.; Tran, T.; and Venkatesh, S. 2018. Graph memory networks for molecular activity prediction. In *2018 24th International Conference on Pattern Recognition (ICPR)*, 639–644. IEEE.

Rabusseau, G.; Li, T.; and Precup, D. 2019. Connecting weighted automata and recurrent neural networks through spectral learning. In *The 22nd International Conference on Artificial Intelligence and Statistics*, 1630–1639. PMLR.

Reed, S.; and De Freitas, N. 2015. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279* .

Saxton, D.; Grefenstette, E.; Hill, F.; and Kohli, P. 2019. Analysing mathematical reasoning abilities of neural models. *arXiv preprint arXiv:1904.01557* .

Socher, R.; Chen, D.; Manning, C. D.; and Ng, A. 2013a. Reasoning with neural tensor networks for knowledge base completion. *Advances in neural information processing systems* 26: 926–934.

Socher, R.; Lin, C. C.; Manning, C.; and Ng, A. Y. 2011. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, 129–136.

Socher, R.; Perelygin, A.; Wu, J.; Chuang, J.; Manning, C. D.; Ng, A. Y.; and Potts, C. 2013b. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, 1631–1642.

Stogin, D.; Mali, A.; Giles, D.; et al. 2020. Provably Stable Interpretable Encodings of Context Free Grammars in RNNs with a Differentiable Stack. *arXiv preprint arXiv:2006.03651* .

Sukhbaatar, S.; Weston, J.; Fergus, R.; et al. 2015. End-to-end memory networks. In *Advances in neural information processing systems*, 2440–2448.

Sun, G. Z.; Giles, C. L.; and Chen, H. H. 1998. *The neural network pushdown automaton: Architecture, dynamics and training*, 296–345. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-69752-7. doi:10.1007/BFb0054003.

Sun, G.-Z.; Giles, C. L.; Chen, H.-H.; and Lee, Y.-C. 2017. The neural network pushdown automaton: Model, stack and learning simulations. *arXiv preprint arXiv:1711.05738* .

Suzgun, M.; Gehrmann, S.; Belinkov, Y.; and Shieber, S. M. 2019a. LSTM Networks Can Perform Dynamic Counting. *CoRR* abs/1906.03648. URL http://arxiv.org/abs/1906.03648.

Suzgun, M.; Gehrmann, S.; Belinkov, Y.; and Shieber, S. M. 2019b. Memory-Augmented Recurrent Neural Networks Can Learn Generalized Dyck Languages. *CoRR* abs/1911.03329.

Tai, K. S.; Socher, R.; and Manning, C. D. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* .

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Weston, J.; Chopra, S.; and Bordes, A. 2014. Memory networks. *arXiv preprint arXiv:1410.3916* .

Wu, Y.; Zhang, S.; Zhang, Y.; Bengio, Y.; and Salakhutdinov, R. R. 2016. On multiplicative integration with recurrent neural networks. In *Advances in Neural Information Processing Systems*, 2856–2864.

Zaremba, W.; Kurach, K.; and Fergus, R. 2014. Learning to discover efficient mathematical identities. In *Advances in Neural Information Processing Systems*, 1278–1286.