Memory Augmented Recursive Neural Networks

Forough Arabshahi[1] [†]   **Zhichu Lu**[1] [†]   **Sameer Singh**[‡]   **Animashree Anandkumar**[♮]

farabsha@cs.cmu.edu   zhicul@cs.cmu.edu   sameer@uci.edu   anima@caltech.edu

[†]Carnegie Mellon University, [‡]University of California Irvine, [♮]California Institute of Technology

Abstract

Recursive neural networks have shown an impressive performance for modeling compositional data compared to their recurrent counterparts. Although recursive neural networks are better at capturing long range dependencies, their generalization performance starts to decay as the test data becomes more compositional and potentially deeper than the training data. In this paper, we present memory-augmented recursive neural networks to address this generalization performance loss on deeper data points. We augment Tree-LSTMs with an external memory, namely neural *stacks*. We define soft push and pop operations for filling and emptying the memory to ensure that the networks remain end-to-end differentiable. In order to assess the effectiveness of the external memory, we evaluate our model on a neural programming task introduced in the literature called equation verification. Our results indicate that augmenting recursive neural networks with external memory consistently improves the generalization performance on deeper data points compared to the state-of-the-art Tree-LSTM by up to 10%.

1 Introduction

Compositional data are ubiquitous and comopositional models are naturally the right candidates to fit such data. For example, recursive neural networks such as Tree-RNNs and Tree-LSTMs have been successfully applied to tasks in natural language processing, vision and neural programming (Tai et al., 2015; Socher et al.,

2011; Allamanis et al., 2017; Arabshahi et al., 2018; Evans et al., 2018).

Although recursive networks significantly outperform their flat counterparts, they still fail to generalize to unseen composition complexity. In other words, if we fix the composition complexity of the training data and test the models on unseen data points, the performance of the models start to decay as the complexity of the unseen data grows beyond that of the training data. This is an ongoing challenge for neural networks and it is important to address it because it is unreasonable to assume that the model sees all possible composition complexities during training. Therefore, it is important to build architectures that generalize to unseen composition complexity.

One of the reasons of this lack of generalization is error propagation as the composition complexity and depth grows. The more representative the components of the recursive neural network, the less the error propagation and the better the generalization performance to higher complexity. We show in this paper that external memory improves the generalization performance of recursive neural networks for neural programming.

Neural programming refers to the task of using neural network to learn programs, mathematics and logic from data. Often in neural programming, a core neural network cell is used to learn and represent a certain program such as addition or sort and mathematical or logical function such as multiplication and OR. Most of these programs and functions can be implemented more compactly and efficiently using recursion, potentially making it easier for neural networks to model them. For example, the pseudo code for a recursive and non-recursive implementation of the multiplication function is given in Figure 2. It is known that stacks are used to implement and execute a recursive function.

In this paper, we build upon this observation and intuition and introduce stack augmented recursive neural networks. We hypothesize that augmenting neural networks with additional stacks allows them to better represent recursive functions. We show that on the
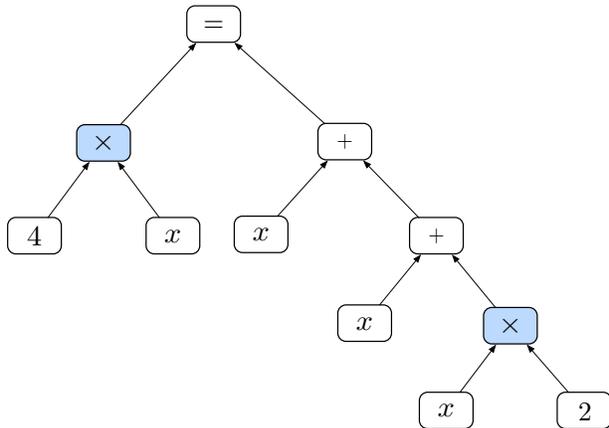
---

[1] Equal Contribution

Figure 1: recursive symbolic multiplication of $4 \times x = x + \big(x + (2 \times x)\big)$ given in Equation 1.

task of mathematical equation verification (Arabshahi et al., 2018) this structure consistently improves the generalization performance of Tree-LSTM on unseen (higher) depths. An example of a symbolic representation of the recursive multiplication algorithm in Figure 2 is given below:

$$4 \times x = x + 3 \times x$$
$$= x + \big(x + (2 \times x)\big) \tag{1}$$
$$= x + \big(x + (x + x)\big) \tag{2}$$

The compositional structure of Equation 1 is depicted in Figure 1. In this paper, we use recursive neural networks for representing symbolic mathematical expressions where each function is a neural network whose weights are shared with the other appearances of that function.

We show that if we augment the cells of a recursive neural network with additional memory with the structure of a stack the cells are able to learn a better representation of the functions resulting in less error propagation and an improved generalization performance.

1.1 Summary of Results

Our contributions in this paper are twofold. First, we propose to augment recursive neural networks with an external memory that has the data structure of a stack. This memory will allow the network to generalize to data of higher complexity and depths.

We augment Tree-RNNs and Tree-LSTMs with an external data structures, namely differentiable stacks. A stack is a Last In First Out (LIFO) data structure often used for recursive computation and implementation. We develop soft push and pop operations for the stack and allow the model to learn to control these

```python
# Iterative multiplication
def mul(x, y):
  # x and y are lists of digits with the
    least significant bit in index 0
  prod = zeros(1,len(x)+len(y)-1)
  for x_i in range(0,len(x)):
    carry = 0
    for y_i in range(0,len(y)):
      prod[x_i+y_i] += carry + x[x_i] *
                       y[y_i]
      carry = prod[x_i+y_i] / 10.0
      prod[x_i+y_i] = prod[x_i+y_i] mod
                      10.0
    prod[x_i+len(y)-1] = carry
  return prod

# Recursive multiplication
def recursive_mul(x,y):
  if x<y:
    return recursive_mul(y,x)
  elif y != 0:
    return (x + recursive_mul(x, y-1))
  else:
    return 0
```

Figure 2: non-recursive vs. recursive multiplication of numbers x and y in base 10. As can be seen, the recursive implementation is more compact. Note that if x and y are floating point numbers they should first be pre-processed and converted into integers through multiplication and the result should be converted back to floating point through division.

operations.

Second, we test the proposed model on a neural programming task called mathematical equation verification where given a symbolic math equality, the goal is to verify if the equation is correct or incorrect (Allamanis et al., 2017; Arabshahi et al., 2018). We show that stack-augmented recursive neural networks consistently improve the generalization performance of Tree-NNs and Tree-LSTMs on unseen composition complexity. Moreover, we provide a model ablation study that shows the effect of different model components on the output results. We show that stack-augmented recursive neural networks consistently improve the performance of Tree-LSTMs and Tree-RNNs on the equation verification task on data of higher composition complexity.

1.2 Related Work

Recursive neural networks have been used to model compositional data in many applications including natural scene classification (Socher et al., 2011), sentiment classification, Semantic Relatedness and syntactic parsing (Tai et al., 2015; Socher et al., 2011), neural programming and logic (Allamanis et al., 2017; Zaremba

et al., 2014; Evans et al., 2018). In all these problems there is an inherent hierarchy nested in the data and capturing it allows the models to improve significantly.

Recursive neural networks have shown to be good at capturing long term dependencies (Tai et al., 2015) compared to flat recurrent neural networks (Graves et al., 2013; Hochreiter and Schmidhuber, 1997). However, their performance significantly drops when generalizing to dependency ranges not seen in the training data.

Recently there have been attempts to provide a global memory to recurrent neural models that will play the role of a working memory and can be used to store information to and read information from. (Graves et al., 2014; Jason Weston, 2015; Grefenstette et al., 2015; Joulin and Mikolov, 2015).

Memory networks and their differentiable counterpart Jason Weston (2015); Sukhbaatar et al. (2015) store instances of the input data into an external memory that can later be read through their recurrent neural network architecture. Neural Programmer Interpreters augment their underlying recurrent LSTM core with a key-value pair style memory and they additionally enable read and write operations for accessing it Reed and De Freitas (2015); Cai et al. (2017). Neural Turing Machines Graves et al. (2014) define soft read and write operations so that a recurrent controller unit can access this memory for read and write operations. Another line of research proposes to augment recurrent neural networks with specific data structures such as stacks and queues Das et al. (1992); Sun et al. (2017); Joulin and Mikolov (2015); Grefenstette et al. (2015). There has also been an attempt to improve the performance of recurrent neural networks (Trinh et al., 2018)

Despite the amount of effort spent on augmenting recurrent neural networks, to the best of our knowledge, there has been no attempt to enable an external memory to recursive networks which will allow them to generalize to higher recursion depths. In each step of the traversal the nodes get to read from or write to this additional memory. Therefore, in this work, inspired by the recent attempts to augment recurrent neural networks with stacks, we propose to augment recursive neural networks with an external memory that they can access to fill or read from.

In a parallel research direction Kumar et al. (2016) present episodic memory for question answering applications. This is different from the symbolic way of defining memory for models that handle neural programming tasks. Another different line of work are graph memory networks and tree memory networks Pham et al. (2018); Fernando et al. (2018) which construct a memory with a specific structure and are different from augmenting a recursive neural network with

an additional global memory

2 Memory augmented recursive neural networks

In this section, we introduce memory augmented recursive neural networks. We augment Tree-RNNs and Tree-LSTMs with external differentiable memory. This memory has the data structure of a neural stack. The network learns to read from and write to the memory by popping from and pushing to the memory, respectively. We develop soft push and pop operations making the network end-to-end differentiable, therefore, we can train these networks in an end-to-end manner. A sketch of the stack-augmented Tree-LSTM is shown in Figure 3b. Stacks are the $\mathbf{S}_j$, $\mathbf{S}_{c_{j1}}$ and $\mathbf{S}_{c_{j2}}$ in that figure, and the model accesses the memory through a_j. We describe more details about the model in the next subsections.

In this section we present matrices with bold uppercase letters, vectors with bold lowercase letters and scalars with non-bold letters. For simplicity, we will present the equations for a binary recursive neural network in this paper. However, everything can be trivially extended to n-ary recursive neural networks. A recursive neural network is a tree-structured network where each node of the tree is a neural network block. The structure of the tree is often indicated by the data. For example, for language models, the structure of the tree can be the structure of the dependency parse of the input sentence, or for mathematical equation verification, the structure of the tree is the structure of the input equation (Figure 1).

All the nodes or blocks of the recursive neural networks have a state denoted by $\mathbf{h}_j \in \mathbb{R}^n$ and an input denoted by $\mathbf{i}_j \in \mathbb{R}^{2n}$ where n is the hidden dimension, $j \in [0, N-1]$ and N is the number of nodes in the tree. Let us label the children of node j with c_{j1} and c_{j2}. We have

$$\mathbf{i}_j = [\mathbf{h}_{c_{j1}}; \mathbf{h}_{c_{j2}}] \tag{3}$$

where $[\cdot \, ; \, \cdot]$ indicates concatenation. If the block is a leaf node, $\mathbf{i}_j$ is the input of the network. For example, in the equation tree shown in Figure 1, all the terminal nodes are the inputs to the neural network. Note that for simplicity we assume that the internal blocks do not have an external input and only take inputs from their children. However, the extension to the case where we do have an external input is trivial and can be done by additionally concatenating the input with the children's states in Equation 3.

the way $\mathbf{h}_j$ is computed using $\mathbf{i}_j$ depends on the neural network block's architecture and in the subsections below we will explain how $\mathbf{h}_j$ is computed given the
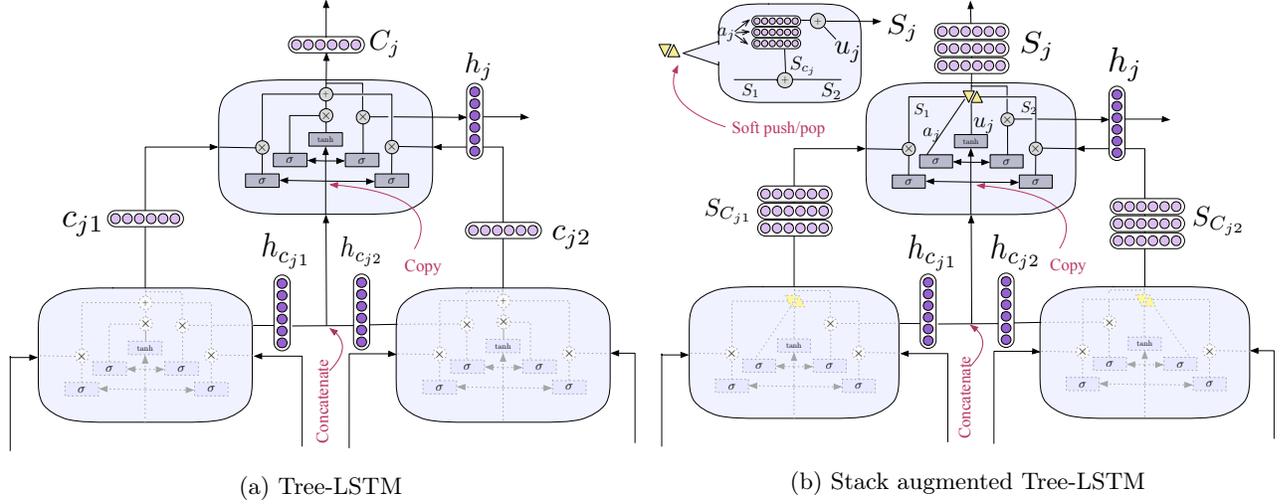
(a) Tree-LSTM

(b) Stack augmented Tree-LSTM

Figure 3: Model architecture of Tree-LSTM vs. stack-augmented Tree-LSTM. The yellow symbol in Figure 3b represents the soft push/pop operation and it's details are shown in the square labeled soft push/pop. Both figures show a sub-tree with a parent block and its two children.

input state for each model. In the following sub-sections we will re-use notation to make model comparison easier and refrain from introducing too many variables to make the flow easier. We have tried to make the names consistent for corresponding variables across models. Therefore, note that the variables defined in each sub-section should only be used within that sub-section unless noted otherwise.

2.1 Tree-RNNs

before we present the stack-augmented recursive neural networks, let us start with vanilla Tree-RNNs.

the node state $\mathbf{h}_j$ is computed by passing $\mathbf{i}_j$ through a feed-forward neural network. For example, Assuming that the network blocks are single-layer networks we have,

$$\mathbf{h}_j = f(\mathbf{W}_j \mathbf{i}_j + \mathbf{b}_j), \qquad (4)$$

where f is a nonlinear function such as Sigmoid and $\mathbf{W}_j \in \mathbb{R}^{n \times 2n}$ is the matrix of network weights and $\mathbf{b}_j$ is the bias. Note that the weights and biases are indexed with j. These weights can either be shared among all the blocks or can be specific to the node type. To elaborate, in the case of mathematical equation verification the parameters can be specific to the function such that we have different weights for addition and multiplication. In the simplest case, the weights are shared among all the neural network block in the tree.

Stack augmented Tree-RNNs

In this model, each node j is augmented with a stack $\mathbf{S}_j \in \mathbb{R}^{p \times n}$ where p is the stack size. The input of each

node will be stored in the stack if the model decides to push, and the output representation is computed using the stack if the model decides to pop from it.

A stack is a LIFO data structure and the network can only interact with it through its top. This is the desired behavior when dealing with recursive function execution. We indicate the top of the stack with $\mathbf{S}_j[0] \in \mathbb{R}^n$. The stack has two operations pop and push. Inspired by Joulin and Mikolov (2015) and the push-down automation we use a 2-dimensional action vector $\mathbf{a}_j \in \mathbb{R}^2$ whose elements represent the soft push and pop operations for interacting with the stack. These two actions are controlled by the network's input state at each node.

$$\mathbf{a}_j = \sigma(\mathbf{A}_j \mathbf{i}_j + \mathbf{b}_j^{(a)}) \qquad (5)$$

where $\mathbf{A}_j \in \mathbb{R}^{2 \times 2n}$ and σ is the softmax function. We denote the probability of the action push with $\mathbf{a}_j^{\text{push}} = \mathbf{a_j}[0] \in [0, 1]$ and pop with $\mathbf{a}_j^{\text{pop}} = \mathbf{a_j}[1] \in [0, 1]$ and these two probabilities sum to 1.

We assume that the top of the stack is located at index 0. Let $\mathbf{S}_{c_j} \in \mathbb{R}^{p \times 2n}$ denote the concatenation of the children's stacks given below

$$\mathbf{S}_{c_j} = [\mathbf{S}_{c_{j1}}; \mathbf{S}_{c_{j2}}] \qquad (6)$$

We have,

$$\mathbf{S}_{c_j}^{\text{push}} = \sigma(\mathbf{U}_j^{\text{push}} \mathbf{S}_{c_j}^\top + \mathbf{b}_j^{\text{push}})^\top \qquad (7)$$

$$\mathbf{S}_{c_j}^{\text{pop}} = \sigma(\mathbf{U}_j^{\text{pop}} \mathbf{S}_{c_j}^\top + \mathbf{b}_j^{\text{pop}})^\top \qquad (8)$$

$$\mathbf{u}_j = \sigma(\mathbf{U}_j^{(u)} \mathbf{i}_j + \mathbf{b}_j^{(u)}) \qquad (9)$$

Where $\mathbf{U}_j^{\text{push}}, \mathbf{U}_j^{\text{push}}, \mathbf{U}_j^{(u)} \in \mathbb{R}^{n \times 2n}$ and $(\cdot)^\top$ indicates transposition. The stack update equations are then as

follows

$$\mathbf{S}_j[0] = \mathbf{a}_j^{\text{push}} \ \mathbf{u}_j + \mathbf{a}_j^{\text{pop}} \ \mathbf{S}_{c_j}^{\text{pop}}[1] \tag{10}$$

$$\mathbf{S}_j[i] = \mathbf{a}_j^{\text{push}} \ \mathbf{S}_{c_j}^{\text{push}}[i-1]) + \mathbf{a}_j^{\text{pop}} \ \mathbf{S}_{c_j}^{\text{pop}}[i+1] \tag{11}$$

where $\mathbf{U}_{\text{pop}}, \mathbf{U}_{\text{push}}, \mathbf{U}_j \in \mathbb{R}^{n \times 2n}$ and $i = 1, 2, \ldots, p-1$ is the stack row index. The stack is initialized with the 0 matrix. Note that for $i = p-1$ the stack will be out of index in Equation 11 and we assume that in that case we pop an all-zero vector.

The output state of the node of this model, is computed by looking at the top k elements of the stack where $k \in [1, p]$ and the input state as given below:

$$\mathbf{p}_j = \sigma(\mathbf{U}_j^{(p)} \mathbf{i}_j + \mathbf{b}_{j1}^{(p)}) \tag{12}$$

$$\mathbf{h}_j = f(\mathbf{W}_j \mathbf{i}_j + \mathbf{p}_j \mathbf{S}_j[0 : k-1] + \mathbf{b}_j) \tag{13}$$

where $\mathbf{U}_j^{(p)} \in \mathbb{R}^{k \times n}$, $\mathbf{p}_j \in \mathbb{R}^k$ and $\mathbf{S}_j[0 : k-1]$ indicates the top-k rows of the stack. k is a tuning parameter of the model and its choice is problem dependent.

Additional stack operation: No-Op We can additionally add another stack operation called no-op where $\mathbf{a}_j \in \mathbb{R}^3$ and the elements correspond to push, pop and no-op. No-op is the state where the network neither pushes to the stack nor pops from it and keeps the stack as is. We have

$$\mathbf{S}_{c_j}^{\text{no-op}} = \sigma(\mathbf{U}_j^{\text{no-op}} \mathbf{S}_{c_j}^\top + \mathbf{b}_j^{\text{no-op}})^\top \tag{14}$$

where $\mathbf{U}_j^{\text{no-op}} \in \mathbb{R}^{n \times 2n}$. Therefore, the stack operations change as shown below.

$$\mathbf{S}_j[0] = \mathbf{a}_j^{\text{push}} \ \mathbf{u}_j + \mathbf{a}_j^{\text{pop}} \ \mathbf{S}_{c_j}^{\text{pop}}[1] + \\ \mathbf{a}_j^{\text{no-op}} \ \mathbf{S}_{c_j}^{\text{no-op}}[0] \tag{15}$$

$$\mathbf{S}_j[i] = \mathbf{a}_j^{\text{push}} \ \mathbf{S}_{c_j}^{\text{push}}[i-1]) + \mathbf{a}_j^{\text{pop}} \ \mathbf{S}_{c_j}^{\text{pop}}[i+1] + \\ \mathbf{a}_j^{\text{no-op}} \ \mathbf{S}_{c_j}^{\text{no-op}}[i] \tag{16}$$

The state can still be computed using Equation 13 but with the new stack updated using Equations 15 and 16.

2.2 Tree-LSTMs

In this section we present stack-augmented Tree-LSTMs. Similar to Tree-RNNs and in order to be self-contained, we start from Tree-LSTM and then ex-

tend it to stack-augmented tree-LSTMs.

$$\mathbf{inp}_j = \sigma(\mathbf{U}_j^{(inp)} \mathbf{i}_j + \mathbf{b}_j^{(i)}) \tag{17}$$

$$\mathbf{f}_{j1} = \sigma(\mathbf{U}_{j1}^{(f)} \mathbf{i}_j + \mathbf{b}_{j1}^{(f)}) \tag{18}$$

$$\mathbf{f}_{j2} = \sigma(\mathbf{U}_{j2}^{(f)} \mathbf{i}_j + \mathbf{b}_{j2}^{(f)}) \tag{19}$$

$$\mathbf{o}_j = \sigma(\mathbf{U}_j^{(o)} \mathbf{i}_j + \mathbf{b}_j^{(o)}) \tag{20}$$

$$\mathbf{u}_j = \tanh(\mathbf{U}_j^{(u)} \mathbf{i}_j + \mathbf{b}_j^{(u)}) \tag{21}$$

$$\mathbf{c}_j = \mathbf{inp}_j \odot \mathbf{u}_j + \mathbf{f}_{j1} \odot \mathbf{c}_{j1} + \mathbf{f}_{j2} \odot \mathbf{c}_{j2} \tag{22}$$

$$\mathbf{h}_j = \mathbf{o}_j \odot \tanh(\mathbf{c}_j) \tag{23}$$

Where $\odot$ indicates element-wise multiplication, all the vectors in the left-hand-side of equations 17-23 are in $\mathbb{R}^n$ and all the weight vectors $\mathbf{U}$ in the right-hand sides are weight matrices in $\mathbb{R}^{n \times 2n}$. This structure is shown in Figure 3a. As it is shown in the figure, Tree-LSTM's memory, $\mathbf{c}_j$, is a 1-dimensional vector. The stack-augmented Tree-LSTM will have a 2-dimensional vector for the memory ($\mathbf{S}_j$ in Figure 3b) where each row corresponds to a stack entry. We propose a push and pop mechanism for reading from and writing to this memory. In the next subsection we introduce stack-augmented Tree-LSTMs.

Stack-augmented Tree-LSTMs

Tree-LSTM equations were presented in Equations 17 through 23. There are two main differences between tree-LSTM and its stack-augmented counterpart. First, the stack-augmented Tree-LSTM does not have an input gate (**inp** in Equation 17) and instead uses a push gate to combine the input with the contents of the memory. Moreover, it fills up the stack using the push and pop gates that are presented below.

The children's stacks are combined using $\mathbf{f}_{j1}$ and $\mathbf{f}_{j2}$ gates in Equations 18 and 19

$$\mathbf{S}_{c_j}[i] = \mathbf{f}_{j1} \odot \mathbf{S}_{c_{j1}}[i] + \mathbf{f}_{j2} \odot \mathbf{S}_{c_{j2}}[i] \tag{24}$$

for $i = 0, \ldots, p-1$.

Here the push and pop operations are element-wise gates given below

$$\mathbf{a}_j^{\text{push}} = \sigma(\mathbf{A}_j^{(\text{push})} \mathbf{i}_j + \mathbf{b}_j^{(\text{push})}) \tag{25}$$

$$\mathbf{a}_j^{\text{pop}} = \sigma(\mathbf{A}_j^{(\text{pop})} \mathbf{i}_j + \mathbf{b}_j^{(\text{pop})}) \tag{26}$$

where $\mathbf{a}_{j,\text{push}}, \mathbf{a}_{j,\text{pop}} \in \mathbb{R}^n$

The stack and state update equations are therefore:

$$\mathbf{S}_j[0] = \mathbf{a}_j^{\text{push}} \odot \mathbf{u}_j + \mathbf{a}_j^{\text{pop}} \odot \mathbf{S}_{c_j}[1] \tag{27}$$

$$\mathbf{S}_j[i] = \mathbf{a}_j^{\text{push}} \odot \mathbf{S}_{c_j}[i-1] + \mathbf{a}_j^{\text{pop}} \odot \mathbf{S}_{c_j}[i+1] \tag{28}$$

where $\mathbf{u}_j$ is given in Equation 21. The output state is computes by looking at the top-k stack elements as shown below if $k > 1$

$$\mathbf{p}_j = \sigma(\mathbf{U}_j^{(p)}\mathbf{i}_j + \mathbf{b}_{j1}^{(p)}) \tag{29}$$

$$\mathbf{h}_j = \mathbf{o}_j \odot \tanh\left(\mathbf{p}_j \mathbf{S}_j[0:k-1]\right) \tag{30}$$

where $\mathbf{U}_j^{(p)} \in \mathbb{R}^{k \times n}$ $\mathbf{p}_j^{(i)} \in \mathbb{R}^{1 \times k}$ and $\mathbf{S}_j[0:k-1]$ indicates the top-k rows of the stack. If $k = 1$ we have:

$$\mathbf{h}_j = \mathbf{o}_j \odot \tanh(\mathbf{S}_j[0]) \tag{31}$$

where $\mathbf{o}_j$ is given in Equation 20. As noted in the stack-augmented Tree-LSTM section, k is a problem dependent tuning parameter.

Additional stack operation: No-Op Similar to stack-augmented Tree-RNNs we can use the no-op operator to keep the stack in its previous state if need be. In this case the no-op gate and the stack update equations are

$$\mathbf{a}_j^{\text{no-op}} = \sigma(\mathbf{A}_j^{(\text{no-op})}\mathbf{i}_j + \mathbf{b}_j^{(\text{no-op})}) \tag{32}$$

where $\mathbf{a}_j^{\text{no-op}} \in \mathbb{R}^n$. The stack update equations change as shown below

$$\mathbf{S}_j[0] = \mathbf{a}_j^{\text{push}} \odot \mathbf{u}_j + \mathbf{a}_j^{\text{pop}} \odot \mathbf{S}_{c_j}[1] +$$
$$\mathbf{a}_j^{\text{no-op}} \odot \mathbf{S}_{c_j}[0] \tag{33}$$
$$\mathbf{S}_j[i] = \mathbf{a}_j^{\text{push}} \odot \mathbf{S}_{c_j}[i-1] + \mathbf{a}_j^{\text{pop}} \odot \mathbf{S}_{c_j}[i+1] +$$
$$\mathbf{a}_j^{\text{no-op}} \odot \mathbf{S}_{c_j}[i] \tag{34}$$

Similar to stack-augmented Tree-RNNs, the output can be computed using the stack-augmented Tree-LSTM $\mathbf{h}_j$ in Equations 30 and 31 depending on the k.

3 Experimental setup

In this section we discuss the problem we evaluate our model on and state our implementation details. We explore the applicability of our model in a neural programming task called mathematical equation verification defined by Arabshahi et al. (2018). We briefly define this task in the next section and then provide implementation details about our model.

3.1 Mathematical Equation Verification

In this task, the inputs are symbolic and numeric mathematical equations from trigonometry and linear algebra and the goal is to verify their correctness. For example, the following symbolic equation is correct:

$$1 + \tan^2(\theta) = \frac{1}{\cos^2(\theta)}, \tag{35}$$

whereas the numeric equation $\sin(\frac{\pi}{2}) = 0.5$ is incorrect. These symbolic and numeric equations are a composition of mathematical functions in trigonometry and algebra, and the recursive neural networks in the experiments mirror the composition structure of each input equation. The equations and therefore the recursive neural networks are rooted at equality as shown in Figure 1. An indication of composition complexity in this scenario is the depth of the equations. We observe that as the depth of the equations grow beyond that of training data, the accuracy drops significantly and we show that augmenting recursive neural networks improve the generalization performance on equations of higher depth.

A data generation strategy for this task was presented in Arabshahi et al. (2018). We use this data generation strategy to generate symbolic mathematical equations of up to depth 13. We generate $41,894$ equations of different depths. The complete data statistics is given in Table 2. This will allow us to evaluate the generalizability of our model on more complex compositions. The full statistics of the generated data is presented in Table 2. As it can be seen in the Table, the dataset is approximately balanced with 56% correct and 44% incorrect equations. We generate equations of depth 1 through 13. We train the models on equations of depth 1 through 7 and evaluate the model on equations of depth 8 through 13. More details about this task are given in Arabshahi et al. (2018). Table 1 lists some examples that were generated sampled randomly.

3.2 Implementation Details

The models are implemented in PyTorch (Paszke et al., 2017). Our recursive neural networks perform a two-class classification by optimizing the softmax loss on the output of the root. The root of the model represents equality and performs a dot product of the output embeddings of the right and left sub-tree. The input of the neural networks are the terminal nodes in the equations that consist of symbols representing variables in the equation and numbers. The leaves of the neural networks are two-layer feed-forward networks that embed the symbols and numbers in the equation and the other tree nodes are single-layer neural networks that represent different functions in the equation. We share the parameters of the nodes that have the same functionality. For example, all the addition functions use the same set of parameters.

All the models use the Adam optimizer (Kingma and Ba (2014)) with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and learning rate 0.001. We regularize the models with a 0.00001 weight decay. The hidden dimension of the models are 50. All the models are ran using three different seeds and the reported results are the average of the three seeds as

Table 1: **Examples of generated equations in the dataset**

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

Table 2: **Data set statistics**

Statistic	Data depth													
	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 portion	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

well as their standard deviation. We choose the models based on the best accuracy on the validation data. The train and validation datasets contain equations of depth 1-7 and the test dataset contains equations of depth 8 through 13. The code and the data for re-producing the experiments is available here[2].

3.3 Baselines and Evaluation Metrics

We use several baselines to validate our experiments as described below. We also provide a model ablation study and investigate the behavior of our model under different settings in Section 4. In order to assess the generalizability of the model, we train our models on Equations of depths 1 through 7 and test our model on Equations of depths 8 through 13. Let us first discuss the baselines that we used in the experiments.

Majority class baseline is a classification that always predicts the majority class. This is an indication of how hard the classification task is and shows how balanced the training data is.

Tree-RNN is the vanilla recursive neural network presented in Section 2. The implementation details of this network is similar to the experimental setup given in Section 3.2.

Tree-LSTM is the Tree-LSTM network proposed by Tai et al. (2015) and presented in Section 2. the leaves

of this Tree-LSTM are two-layer neural networks that embed the symbols and numbers and the other nodes are LSTM networks whose weights are shared between the same function. The hidden dimension and the optimizer parameters are the same as what's described in Section 3.2 .

We did not choose recurrent neural networks as baselines since the above baselines have already outperformed these models on the same task (Arabshahi et al., 2018).

Evaluation metric Our evaluation metric is the accuracy, precision and recall of predicting correct and incorrect equations. These metrics are reported as a percentage in Table 3 and abbreviated as Acc, Prec and Rcl for accuracy, precision and recall, respectively.

4 Results

In this section, we evaluate the performance of the stack-augmented tree-RNNs and Tree-LSTMs. As mentioned, we evaluate our model on the task of equation verification described in Section 3.

Table 3 shows the performance of all models on our equation verification task. We report the average and standard deviation of accuracy, precision and recall for the models initialized with different random seeds. The stack size in all the models in Table 3 is set to 5 and the models are choosing the top-1 stack element to compute the output, therefore $k = 1$.

---

[2]https://github.com/ForoughA/recursiveMemNet

Table 3: **Overall accuracy of the models on train and test datasets**

| Approach	Train (Depths 1-7)			validation (Depths 1-7)			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±0.92	82.63±0.07	79.29 ± 2.26
Tree-NN+Stack	95.92	95.74	97.32	88.88	87.37	93.95	78.02 ± 0.12	81.28 ± 1.75	74.74 ± 2.29
Tree-NN+Stack +no-op	95.86	96.40	96.49	88.44	87.21	93.29	78.87 ± 1.10	83.23 ± 0.95	74.10 ± 3.91
Tree-LSTM	99.34	99.40	99.47	93.86	92.67	96.82	83.64 ± 1.23	**86.96 ± 0.89**	80.45 ± 3.83
Tree-LSTM+Stack	99.23	99.19	99.49	93.31	92.36	96.15	85.24 ± 0.25	85.13 ± 0.59	86.59 ± 0.86
Tree-LSTM+Stack+normalize	98.76	98.59	99.29	93.32	92.23	96.33	84.44 ± 1.77	85.49 ± 0.55	84.16 ± 3.38
Tree-LSTM+Stack+normalize+no-op	98.34	98.13	99.04	93.84	92.60	96.87	**86.01 ± 0.74**	86.27 ± 1.14	**86.78 ± 2.70**
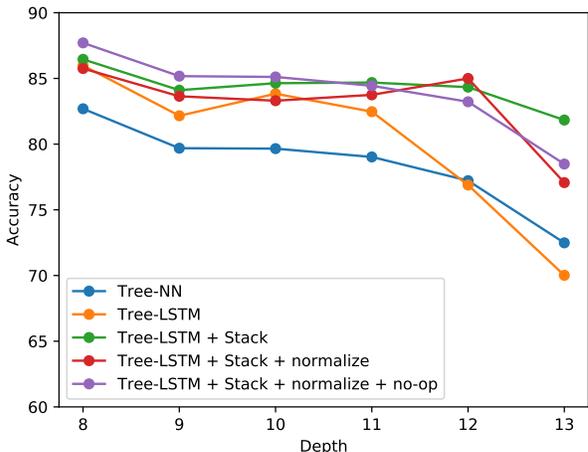
Figure 4: breakdown of model accuracy across different depths for the stack-LSTM models and baselines Tree-RNN and Tree-LSTM

4.1 Model Ablation

Tree-NN+stack refers to the memory-augmented Tree-RNN introduced in Section 2 and Tree-RNN+stack+no-op is the same model with the no-op operation introduced in Section 2. Tree-LSTM+stack refers to the model presented in Section 2. Tree-LSTM+stack+normalize refers to the stack-augmented Tree-LSTM model where the push and pop action vectors $\mathbf{a}_j^{\mathrm{push}}$ and $\mathbf{a}_j^{\mathrm{pop}}$ are element-wise normalized so that each $\mathbf{a}_j^{\mathrm{push}}[i] + \mathbf{a}_j^{\mathrm{pop}}[i] = 1$ for $i = 1, 2, \ldots, n$. Tree-LSTM+stack+normalize+no-op further adds the no-op operator explained in Section 2 to the model. As can be seen the normalized action with no-op is the best model in terms of overall accuracy.

4.2 Generalization to higher depth

In order to see the performance of the models on different depths, we have plotted the accuracy breakdown in terms of depth in Figure 4. As it can be seen, Tree-

LSTM+stack consistently improves the performance of Tree-NN and Tree-LSTM on higher depths and as depth grows beyond the training data, the improvement gap widens. Therefore, stack-augmented recursive neural networks improve model's generalizability on data of higher depth.

This is an important result since generalizing to data of higher complexity than training is an ongoing challenge for neural networks in neural programming and we show that augmenting recursive neural networks with an external memory with the structure of a stack can be a potential solution to this problem.

Specifically, since the models have access to an external memory that has the structure of a stack they are able to fit the functions better and therefore, result in better output representations that will reduce error propagation.

5 Conclusions

Recursive neural networks have shown a good performance for modeling compositional data. However, their performance degrades significantly when generalizing to data of higher complexity. In this paper we present memory augmented recursive neural networks to address this challenge. We augment recursive neural networks, namely Tree-NNs and Tree-LSTMs with an external differentiable memory that has the structure of a stack. Stacks are Last In First Our data structures with two operations push and pop. We present differentiable push and pop operations and augment recursive neural networks with this data structure.

Our experiments indicate that augmenting recursive neural networks with external memory allows the model to generalize to data of higher complexity. We evaluate our model against baselines such as Tree-RNNs and Tree-LSTMs and achieve up to 10% better generalizability compared to the baselines. We also provide a model ablation study to analyze the preformance of the different components of the model on the final result

Acknowledgements

References

M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. Sutton. Learning continuous semantic representations of symbolic expressions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 80–88. JMLR. org, 2017.

F. Arabshahi, S. Singh, and A. Anandkumar. Combining symbolic expressions and black-box function evaluations in neural programs. *International Conference on Learning Representations (ICLR)*, 2018.

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

S. Das, C. L. Giles, and G.-Z. Sun. 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*, page 14, 1992.

R. Evans, D. Saxton, D. Amos, P. Kohli, and E. Grefenstette. Can neural networks understand logical entailment? *International Conference on Learning Representations (ICLR)*, 2018.

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

A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

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

E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

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

A. B. Jason Weston, Sumit Chopra. Memory networks. 2015.

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

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

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

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

T. Pham, T. Tran, and S. Venkatesh. Graph memory networks for molecular activity prediction. *arXiv preprint arXiv:1801.02622*, 2018.

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

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

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

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

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

T. H. Trinh, A. M. Dai, M.-T. Luong, and Q. V. Le. Learning longer-term dependencies in rnns with auxiliary losses. *arXiv preprint arXiv:1803.00144*, 2018.

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