VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Embedding Logic in Neural Networks

Colin Ian Jack Douch, 300229223

Supervisors: Dr. Marcus Frean and Dr. David Balduzzi

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

With the recent success of neural networks in perceptual tasks, one of the major open areas of research is to develop neural networks which have the ability to reason about the features that the perceptual neural networks output.

We present a *switching module* - a new internal module which can be embedded in any neural network. We demonstrate that this switching module allows neural networks to be able to learn a number of algorithmic operations simultaneously. We demonstrate the efficacy of this switching module by applying it to a series of increasingly complex neural networks, building up the set of operations which the system can learn, culminating in the ability to learn binary multiplication. Finally, we demonstrate that the internal primitives from the learned operations hold an additive property which maintains the semantic meaning of the operations.

Contents

1 Introduction **3**

2 Related Works **5**
- 2.1 Recurrent Neural Networks 5
- 2.2 Long Short Term Memories 5
- 2.3 Word2Vec 6
- 2.4 Neural Turing Machines 6
- 2.5 Learning Mathematical Operations 7
- 2.6 Neural GPUs 8

3 Switching Networks **11**
- 3.1 Binary Operators 11
- 3.2 Sequence Learning 14
- 3.3 Memory 14
- 3.4 Mixing 18

4 Switching Neural GPUs **21**
- 4.1 Multiplication 21
- 4.2 Tree Parsing 22

5 Analysis **27**
- 5.1 How Mixing Works 27
- 5.2 Logical Operators and their negations 27
- 5.3 Similarity Measures 28

6 Conclusions **31**

Chapter 1

Introduction

Recent works in the literature of Machine Learning have focused on Recurrent Neural Networks (RNNs), with Long Short Term Memory (LSTM)[1] units, proving their effectiveness at solving many different tasks, including Time Series Forecasting [2, 3], Image Classification and Captioning [4, 5, 6], and Speech Recognition [7, 8] to name a few. Companies such as Google and Facebook are investing large amounts of money to develop these technologies in Voice Recognition [9], Sequence Summarization [10] and Natural Language Processing [11] [12]. In fact for several of these areas, RNNs currently serve as the state of the art.

RNNs have shown particular success in their ability to solve *perceptual* tasks. These involve taking raw inputs from the environment, such as pictures, or audio input streams, and producing usable features. Comlementary to such perceptual processing a new area of research is to develop algorithms for *reasoning* tasks. These are algorithms which can learn from the features produced by a perceptual neural network and use them to solve reasoning tasks. These algorithms can then be coupled with the perceptual Neural Networks to produce systems which can not only interpret the raw inputs from an environment, but also use these to make decisions about, and perform tasks in, the environment. Prior work by Zaremba et al (Facebook) [13, 14] and Kaiser et al (Google) [15] develop Neural Networks that can learn to reason from examples of inputs and outputs to algorithms.

In this report, we take a significant step towards creating neural networks which have the ability to reason more deeply about information presented to it. To do so, we introduce switching modules as new internal modules which can be plugged into any Neural Network. These switching modules allow the network to learn a number of algorithm processes that can be swapped in and out arbitrarily. In this way, we develop the first Neural Networks to be able to simultaneously learn a number of algorithms, constructed efficiently from internal, learned primitives. Additionally we analyse the semantics of these internal primitives and liken them to the continuous vectors generated by the Word2Vec system proposed by Mikolov et al [16], designed to generate meaningful vector representations of words. We show that some of our learned internal primitives have the same additive property in that they can be added and subtracted with meaningful semantic results.

We define an algorithm as a task of compositional nature [14] - a task which requires a multi-step interaction with the world. Recent works have focused on algorithms such as copying a sequence [14, 17], reversing a sequence [15, 14, 17], and basic mathematical operations such as addition and subtraction [13, 14, 17]. It is key to the generalisation ability of these neural constructs that they learn the underlying small tasks of the large problem. If the construct is not able to determine this then generalising to unseen data becomes a problem: if the system is unable to learn the underlying small tasks, it will be unable to discover the general steps to map an input to its output (i.e. the algorithm). If this is the case, then any success which can be seen by the algorithm must be merely the result of learning discrete mappings from the seen inputs to their respective outputs, analogous to "solving" a classification task by simply storing and retrieving the training data. Without interpolation of insight, this will fail in the general case[13]. If we take the example of addition of two binary strings we can see that the underlying operations are a Binary XOR to get the output bit, and a Binary AND to get the carry bit. This process is repeated for every pair of bits in the sequence. Without learning some version of these two actions, the learning system must be, at some level, memorising discrete outputs i.e. individual mappings from discrete inputs to outputs and thus will generalise poorly to unseen sequences.

We draw a parallel here to the atomic primitives that are core to Symbolic AI. Symbolic systems attempt to reason about the world through the use of human defined "atoms" - predicates on which the logic of the system is built such as "At" or "In". One of the reasons this approach has not succeeded, despite decades of effort, however, is that despite their effectiveness on some theoretical tasks (i.e. tasks in which data is not directly from the real world), their applicability to real world tasks (particularly in the area of robotics) has been limited. This was partially due to the "Symbol Grounding Problem" [18]: these atoms, being defined essentially via human introspection, are not "grounded in reality". In effect this means that the logic being built did not directly map on to the inputs received from the world resulting in poor performance. Neural Networks solve this problem by not relying on human defined predicates [18]. Instead Neural Networks learn to solve perceptual tasks by learning from the ground up with the data - with no reliance on human provided prior knowledge (the "atoms" as above).

The rest of this report is organised as follows. Chapter 2 gives an overview of the related works. Chapter 3 introduces the new methods outlined in this report and provides some results as to their performance. In particular, we first introduce a simple switching network for solving multiple different binary operations, comparing this to a similar feed-forward neural network. After proving the switching networks efficacy on these problems, we then look towards operations which require memory, introducing a recurrent version of the switching network which is able to solve addition and subtraction. Crucially, we show that the feed-forward analogue of this network is *unable* to solve these problems in the same manner, and that neither network is able to solve the binary multiplication task. In Chapter 4 we modify the existing Neural GPU architecture[15], including our switching module, and show that our network is now able to learn multiplication. We then use this new construct, deemed the Switching Neural GPU, to evaluate mathematical parse trees and show its effectiveness even on reasonably deep and complex trees. Chapter 5 gives an analysis of some of the aspects of the neural models developed in this report. Finally, Chapter 6 presents conclusions.

Chapter 2

Related Works

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs), are Neural Networks in which one or more layers have an input which is calculated based on the previous output of that layer. To handle this dependency RNNS generally contain memory units - hidden units in the network whose outputs are fed back into the network as inputs, rather than being outputted from the system. RNNs come in several forms, and typically operate over sequential data. Sequential data has no assumption of independance between datapoints - one must have knowledge of previous datapoints to predict future ones. RNN models generally process sequential data on a datapoint-by-datapoint basis with each datapoint being processed independently from the last. Because of this method of processing RNNs are generally independently of the length of the sequences which are presented to them and thus can generalise to arbitrary sequence lengths. This makes them well suited to time series problems.

The original RNNs introduced had very naive memory units. The values stored in these units are replaced after every datapoint, resulting in networks that had difficulty in remembering values for longer than a single iteration. These systems proved to be very effective in areas such as language modelling [19] but are deficient in problems which require values to be remembered for extended lengths of time such as speech processing [20], due to the lack of logic in how, and when, values are stored in the memory units.

2.2 Long Short Term Memories

In a standard RNN there is no explicit control over when a value stored in the memory units is remembered, forgotten or modified. Long Short Term Memories (LSTMs), proposed by Hochreiter and Schmidhuber [21] are a form of memory unit which allows explicit control over how the memories are stored. Recent work has demonstrated their effectiveness in widespread applications in areas such as Time Series Forecasting [2, 3], Image Classification and Captioning [4, 5, 6], and Speech Recognition [7, 8]. LSTM units consist of several learnable *gates* which supply this control. A *gate* in LSTM literature is defined as a construct which takes inputs and produces a value g, where $0 \leq g \leq 1$. This value is used to switch between two values, x and y, outputting a value $gx + (1 - g)y$. Generally x is the previous value stored, and y is the new value we may want to store. The input gate of the LSTM determines how much of the current input is fed into the layer; the forget gate determines when to forget the current value in the layers memory; and the output gate determines whether to output the current output. Each of these gates is tasked with learning a particular mapping from the input and memory to a g, allowing the network to learn when to remember and

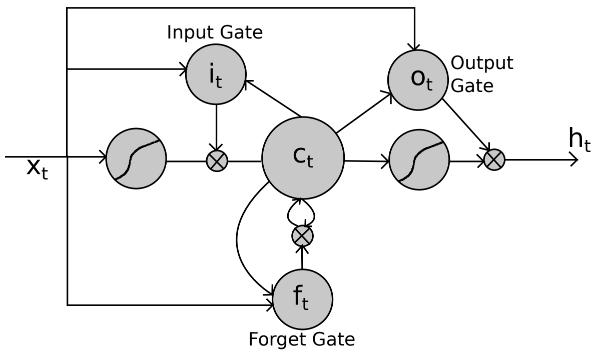


Figure 2.1: The architecture of a Long Short Term Memory Unit, replicated from [21]

when to forget input sequences. A diagram of this architecture is given in Fig. 2.1.

This allows LSTM networks to feasibly remember some values indefinitely, while retaining the flexibility to also quickly pick up new values.

2.3 Word2Vec

Word2Vec, proposed in 2013 by Mikolov et al [16] was a series of systems designed to compute high quality vector representations from sets of words. To do this the authors take shallow neural networks and train them in various ways to reconstruct sentences, given a large corpus of text. This was motivated by a desire to move towards representations of words which capture their semantic similaritites, deviating from previous models in which words are stored as atomic units, devoid of contextual information. Of particular interest to us is the additive property of derived representations. It was demonstrated that the generated vectors can be added and subtracted from each other in ways that produce semantically relevant results. The authors give the example of countries and their capitals. Given the learned function f which takes a word and produces a continuous vector representation of it, the authors show that $f(\text{Berlin}) - f(\text{Germany}) + f(\text{France})$ gives a vector very similar to $f(\text{Paris})$. In this way, the authors conclude that the vector representations they generate are able to capture semantic relationships between words. We demonstrate in Chapter. 5 that the internal primitives we use to represent operations to some extent hold the same property.

2.4 Neural Turing Machines

Neural Turing Machines, proposed by Graves et al [17] in 2014, were one of the first models to escape the architectural bounds of standard feed forward and recurrent neural networks. Thse two models have two such limitations:

1. The network is only presented with every entry in the input sequence once. This means that any algorithm that requires multiple passes of the input cannot be learned, such as reversing a sequence.
2. The number of memory units is linear in the number of hidden units in the network. This means that any algorithm which is not linear ($O(n)$) in memory cannot be learned.

In a Neural Turing Machine architecture, the memory is split out from the computational architecture or “neural controller”. This controller learns to produce outputs which move a number of read and write heads around this memory "tape", performing computations based on the values read from this tape and storing intermediate values onto it. This architecture solves the above two problems by 1. Allowing the system to be able to move around the tape arbitarily to allow multiple passes over the input, and 2. Allow for an arbitarily large armount of memory, not bounded by the size of the network. The authors assess their architecture on a number of tasks, including the "copy" task, in which the network is trained to move an input sequence from one location on its memory tape to another. They show that, despite performing well, the Neural Turing Machine fails to generalise well to arbitrary sequence lengths. They also examine a "sorting" task in which the Turing Machine is tasked with reporting a sorted version of the input sequence. The results in this task are less successful. The authors point out that in some places in the sequence, the Neural Turing Machine indeed follows an optimal process, however, in many places it deviates from this, speculating that the Turing Machine is "forgetting" values at the start of the sequence as it scans (as the only memory in the system is on the tape). Most importantly, each of these tasks represents problems which were previously impossible with standard architectures and most are not used as benchmarking techniques for further development on these architectures.

2.5 Learning Mathematical Operations

Neural networks perform exceptionally well on *perceptual* tasks (interpreting raw data from the world). In such tasks they consistently represent the state of the art by large margins. An important open problem is to develop neural networks which are able to solve more abstract *reasoning* based tasks, based on the perceptual information produced by other algorithms. Ultimately we wish to develop neural networks which can reason about the world in sophisticated ways.

Learning how to execute mathematical operations (and algorithms in general) has been explored in several contexts, particularly as a means of evaluating the expressiveness of existing machine learning constructs. Typically a construct is given two binary sequences, and it must learn to output the result of applying some operation, e.g. addition to those inputs.

Zaremba et al [13] attempt to learn how to execute Python programs with standard LSTMs. Although their approach is successful to some extent, it is still rather error prone. The authors hypothesise that the character-by-character approach to parsing the program may not be the best approach in that the network cannot achieve a semantic understanding of a program's intricacies in such a manner. Additionally, constructs like loops and conditionals, wherein the flow of the program has to be directed back to a previous point, are not well supported in standard feed forward networks for reasons we shall discuss later. Riedel et al[22] expand on this idea using another programming language, Forth. Forth is a stack based language, and the authors hypothesise that, by designing the LSTM network with this prior knowledge in mind, the network might be better able to effectively interpret arbitrary programs. The authors also define more formal methods for dealing with particular programming structures such as branches and loops, modeling branches as gate structures, calculating the result of each branch and using the sum of the two results, weighted by some gating value, as the output of the conditional. With the aid of these formalised methods, the authors show that their LSTM model is able to learn to execute simple Forth "sketches" within the limitations of the LSTM model discussed.

A major limitation encountered by these methods [13, 23], is the memory model of standard recurrent neural networks. In the standard RNN architectures the memory of the network is contained entirely inside the hidden neurons of the network - one value in each memory unit. This constrains the amount of memory in the network to be constant in the architecture in the network. Additionally, the network is presented with each input value in the sequence exactly once before it has to produce an output. If a problem requires multiple passes over the input to produce a correct result, for example reversing a sequence, then a standard architecture will necessarily fail to achieve a correct result in the general case. In order to solve this, new memory models and input methods which allow multiple passes of the input are required.

In another attempt at the learning of algorithms, Zaremba at al use a different model in [14], altering the memory model to include a read and a write "head" on separate, two dimensional, grids. The authors hope to escape the limitations of standard feed forward networks, bringing their model much closer to the ideas of Neural Turing Machines [17] discussed in Section 2.4, except in the model proposed by Zaremba et al the input and output structures are separate and can take on an arbitrary shape, rather than just a one dimensional tape. The authors similarly abstract the movement of the read and write heads to a "controller". The controller is responsible for turning the outputs of the neural network into actions over the input and output grids and is trained through the Q-Learning algorithm. In this way, the model is given free rein to pass over the input arbitrarily. However the authors do encounter an issue - this model finds it difficult to generalise on some tasks. Tasks such as the reversing task, in which the model has to learn to output a reversed version of the input sequence, involve the model moving its read head all the way to the end of the input. Because there is no way for the model to determine that it is at the end of the input sequence, it must move a fixed number of steps at the start of problem. If the model has been trained on sequences of a different length than the one presented, it will necessarily move an incorrect number of steps, resulting in incorrect results.

2.6 Neural GPUs

The Neural GPU model proposed by Kaiser and Sutskever [15] in 2015 is a new type of neural network architecture. Instead of operating as an RNN with memory in the hidden memory units of the model, Neural GPUs operate over a three dimensional ($S \times N \times H$) neural model which acts as the input, memory and output of the system. Here S is the length of the sequence to calculate, N is the number of features in one input and H is the depth parameter (3 in [15]). A key feature of Neural GPUs is that they escape the memory limitations of standard Recurrent Neural Networks. Additionally, because of the nature of the input to the model - the entire sequence being loaded into the model at the beginning, rather than being fed in one entry at a time, the Neural GPU model is able to perform multiple passes over the input sequence.

The Neural GPU itself is composed to two "Convolutional Gated Recurrent Units" (CGRUs), the definition of which is given in Equations. 2.1.

$$\begin{aligned}
\mathrm{CGRU}(s) &= u \odot s + (1-u) \odot \tanh(W * (r \odot s) + B) \\
u &= \sigma(U' * s + B') \\
r &= \sigma(U'' * s + B'') \\
U * s[x,y,i] &= \sum_{u=\lfloor -k_w/2 \rfloor}^{\lfloor k_w/2 \rfloor} \sum_{u=\lfloor -k_h/2 \rfloor}^{\lfloor k_h/2 \rfloor} \sum_{c=1}^{m} s[x+u, y+v, c] \cdot U[u,v,c,i]
\end{aligned} \tag{2.1}$$

Initially, the neural model is loaded with the first layer being an $S \times N$ representation of the entire input sequence. At each iteration, a learned convolution operation is applied

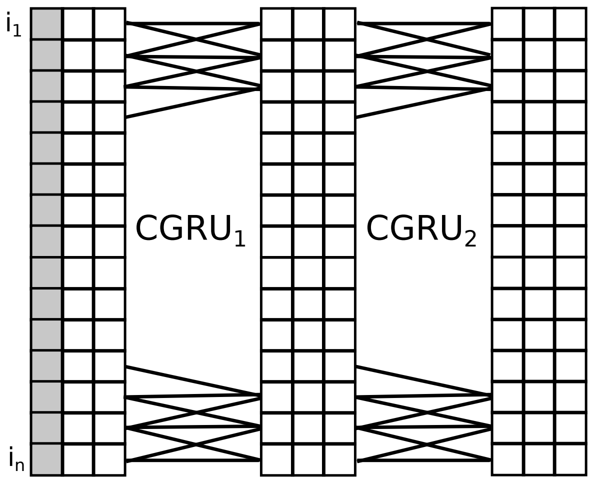


Figure 2.2: The architecture of a Neural GPU, replicated from [15]

over this neural model. After a given number of iterations the first layer is read back again as the output. This convolution operation forms what is learned by the Neural GPU - a 4 dimension bank of convolutions which are applied to move the neural model to the next step. A single iteration of this procedure is shown in Fig. 2.2.

Because the convolution operation is applied over the entire block of the neural model and is not dependent on the sequence length S, the neural GPU can be trained to generalise over arbitrarily long sequences. Kaiser and Sutskever demonstrate that their model learns to solve a variety of difficult problems including reversing a sequence and multiplication of two arbitrarily long bit strings. Additionally, they prove the generalisation ability of the model, demonstrating successful results on bit strings up to 2000 bits long, after having trained on sequences up to 200 bits.

The big win here is the binary multiplication problem. The work done in this paper by Kaiser and Sutskever presents the first Neural Network based system to be able to learn the binary multiplication in the general case and, in doing so, fundementally changes the set of problems previously thought possible by Neural Networks. While perhaps seemingly innocuous, Multiplication presents a very difficult problem for traditional learning models. Even with an optimal algorithm, Multiplication requires $O(n^2)$ steps over the input sequence, and $O(n \log n)$ memory - both properties not traditionally supported by neural networks. On top of these limitations, the Neural GPU is given no prior knowledge as to the actual steps behind performing multiplication. Simply from examples of the two input bit strings, and the resultant bit string of multiplying them together, the Neural GPU is able to infer the complex underlying sub tasks involved in order to be able to develop a general algorithm which can solve this task.

In Chapter 4 we apply our switching module to the Neural GPU architecture, demonstrating the new models ability to learn a number of operators, including multiplication simultaneously.

Chapter 3

Switching Networks

In this chapter we introduce the idea of a "Switching Module" and show its effectiveness in learning binary operators before expanding it to support arbitrarily long sequences and operations which require memory.

The idea of Switching networks is to represent a number of problems in a single network, utilising the similarities between problems to "compress" the internal representation compared to simply having a single network for each problem. Additionally, switching networks allows us to encapsulate the idea of an operator which is applied to operands. A switching network is a neural network structure which has the ability to *switch* its output based on a secondary input, designated the "operator". We present several such networks, each building on the previous to allow for larger classes of problems to be solved.

3.1 Binary Operators

Our first foray into the learning or operators comes in the form of learning logical operators on binary strings. We focus on the operators AND, OR, XOR, and their negations NAND, NOR and NXOR, as well as negation of the two inputs: NOT(X) and NOT(Y), and the identity functions, True(X) and False(X) which emit sequences the same length as X of 1s and 0s respectively. These were chosen as a first step as they operate over the bits of a binary sequence independently, requiring no memory to solve.

We compare two architectures here. The first we apply is a standard feed forward network. This network is constructed on an input layer, 2 sigmoid layers and a softmax output layer. This architecture is given graphically in Fig. 3.1. This network takes as an input a concatenation of the three input variables in this system - two binary strings and an operation to perform on them. In this problem the operator encoding o was fixed such that every operator is orthogonal to every other, i.e. operators are given representations of $(1, 0, 0, ...)$, $(0, 1, 0, 0...)$, $(0, 0, 1, 0, 0...)$ etc for simplicity.

In contrast to the above feed forward model, we now introduce the idea of a switching module. The switching module is an internal module in a neural network which encapsulates the idea of an operator as a seperate piece of data to the operands that it acts on. Equation. 3.1 gives the definition of this switching module.

$$\begin{aligned} a &= \sigma(X \cdot W) \\ b &= \sigma(o \cdot W') \\ out &= a \odot b \end{aligned} \tag{3.1}$$

Here X is of the form (x_1, x_2), a concatenated version of the two binary strings we want

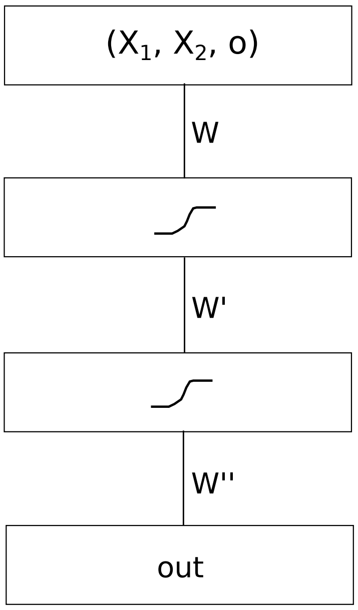


Figure 3.1: The architecture of an equivalent Feed Forward Network to out introduced switching network

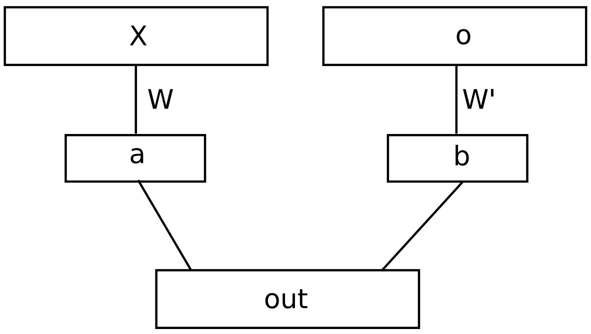


Figure 3.2: The network diagram for the proposed switching module. X is a concatenation of the two input strings. Note that a and b have the same number of hidden units

to perform the operation on, and *o* is a representation of the operation we want to perform. Fig. 3.2 gives a diagram of this architecture.

The major difference between the switching module and the standard feed forward network can be seen from Fig. 3.2. In particular, note the separation between the operator and the operands on which it acts.

In order to evaluate this system we trained a network composing of two switching modules with 30 and 10 hidden units respectively, on a dataset consisting of logical operations on binary problems utilising all the different operations given above. In order to understand the efficacy of the method a truth table presented in Table. 3.1, was generated from the network after training for 5000 epochs using the Adagrad learning method [24]. We establish this truth table by rounding each value to the closest of either 0 or 1. In doing so we can establish that the network is, indeed, able to learn all of the operators presented to it.

Fig. 3.3 presents a comparison of the error rates over training time for our switching network and for an equivalent standard feed forward network. The results presented here are a result of averaging the error outputs at each epoch over 30 experiments each.

It can be seen that each of the systems is able to learn this problem. This is not surprising

Operator	$X:0,Y:0$	$X:0,Y:1$	$X:1,Y:0$	$X:1,Y:1$	Correct?
AND(X, Y)	$1.115 \times 10^{-03}(0)$	$8.738 \times 10^{-04}(0)$	$5.051 \times 10^{-03}(0)$	$9.923 \times 10^{-01}(1)$	True
OR(X, Y)	$4.463 \times 10^{-03}(0)$	$9.944 \times 10^{-01}(1)$	$9.945 \times 10^{-01}(1)$	$9.953 \times 10^{-01}(1)$	True
XOR(X, Y)	$6.747 \times 10^{-02}(0)$	$6.748 \times 10^{-01}(1)$	$7.026 \times 10^{-01}(1)$	$6.451 \times 10^{-03}(0)$	True
X(X, Y)	$3.273 \times 10^{-04}(0)$	$7.359 \times 10^{-04}(0)$	$9.950 \times 10^{-01}(1)$	$9.944 \times 10^{-01}(1)$	True
Y(X, Y)	$2.945 \times 10^{-03}(0)$	$9.923 \times 10^{-01}(1)$	$1.925 \times 10^{-02}(0)$	$9.928 \times 10^{-01}(1)$	True
FALSE(X, Y)	$3.701 \times 10^{-05}(0)$	$2.901 \times 10^{-04}(0)$	$1.391 \times 10^{-04}(0)$	$2.400 \times 10^{-04}(0)$	True
TRUE(X, Y)	$9.981 \times 10^{-01}(1)$	$9.981 \times 10^{-01}(1)$	$9.980 \times 10^{-01}(1)$	$9.981 \times 10^{-01}(1)$	True
NAND(X, Y)	$9.967 \times 10^{-01}(1)$	$9.968 \times 10^{-01}(1)$	$9.969 \times 10^{-01}(1)$	$1.999 \times 10^{-03}(0)$	True
NOR(X, Y)	$9.941 \times 10^{-01}(1)$	$8.132 \times 10^{-05}(0)$	$8.220 \times 10^{-04}(0)$	$4.399 \times 10^{-05}(0)$	True
NXOR(X, Y)	$6.998 \times 10^{-01}(1)$	$1.219 \times 10^{-03}(0)$	$6.478 \times 10^{-02}(0)$	$6.748 \times 10^{-01}(1)$	True
NOT(X)(X, Y)	$9.952 \times 10^{-01}(1)$	$9.948 \times 10^{-01}(1)$	$6.522 \times 10^{-04}(0)$	$7.765 \times 10^{-04}(0)$	True
NOT(Y)(X, Y)	$9.951 \times 10^{-01}(1)$	$3.251 \times 10^{-05}(0)$	$9.958 \times 10^{-01}(1)$	$1.020 \times 10^{-04}(0)$	True

Table 3.1: A Truth Table, Outputted from our Switching Network. We can see that the switching network is able to learn all of the operators presented to it.

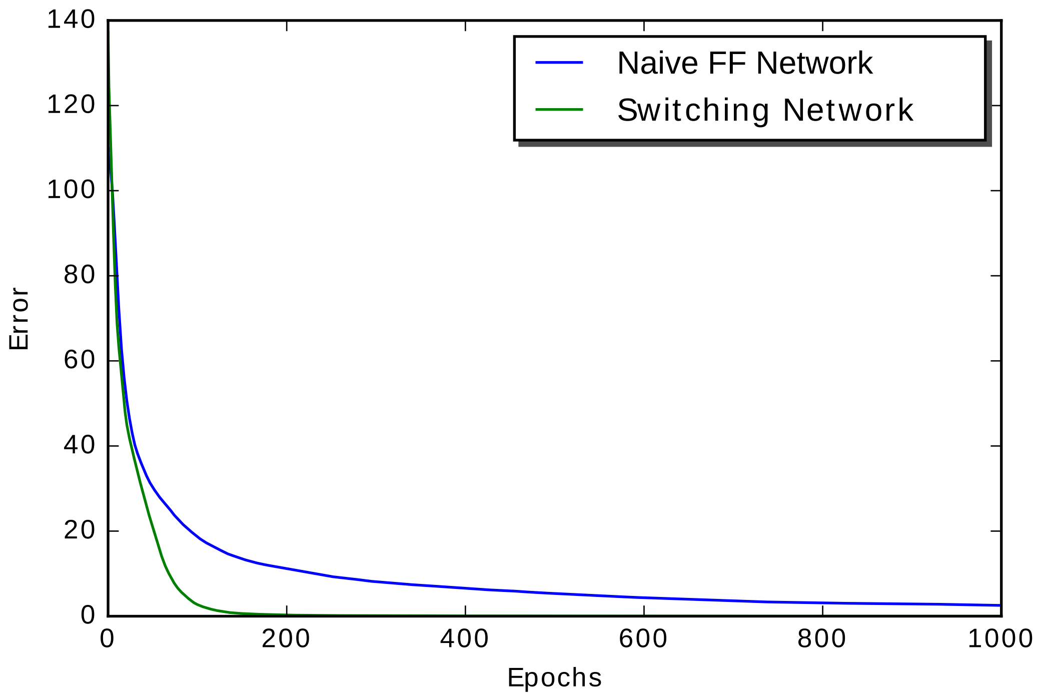


Figure 3.3: The softmax error rate of a standard Feed Forward Network vs the introduced Switching Network. The switching network solves this problem faster than the feed forward model

- feed forward neural networks have been proven to be very effective at binary operators previously. However the standard network takes longer to learn this problem (2500 epochs for the feed forward model compared to 100 for the Switching Network model). The speed of our own algorithm can possibly be attributed to the introduction of the prior knowledge that is the separation of the operands and operator.

3.2 Sequence Learning

Based on the previous proposal, it is trivial to extend the architecture proposed in Section. 3.1 to process arbitrarily long sequences, as no memory is required for logical operations on Booleans. Instead of supplying X as a concatenation of the two binary sequences we want to apply our operation to, we instead supply to the network an n long sequence of two paired bits, one from each of given binary sequences (the operands). At each step in the sequence, we supply one of these 2 bit vectors, and a vector representing the operator, recording the output from the network. The operator here maintains the representation from our previous networks. Note that this problem is harder than the previous one due to the fact that the network only receives an error signal at the *end* of a given sequence. This means that any errors will be compounded and therefore harder to correct.

Such an architecture can, in principle, support processing of arbitrarily long binary sequences because no part of the model depends on the sequence length n. We can thus use such a model to test the ability of our network to generalise to unseen sequence lengths. To do this, instead of testing and training our network on the same sequence lengths, we instead train and test on a number of sequence lengths (i.e. training on sequences randomly chosen between 2 and 10 bits long, and testing on sequences between 50 and 5000 bits long).

To demonstrate the efficacy of this method in terms of generalisation, in Figure. 3.4 we present some examples of the outputs of this network after training on short sequences for 5000 epochs using the Adagrad method with a learning rate of 0.2. These representative examples show that the network is able to generalise well to sequences with lengths greater than those it was trained on.

Fig. 3.5 gives the error over time of the sequenced switching network. It can be seen that the system retains its ability to learn these problems, even when testing on sequences of lengths up to 5000 bits, despite taking slightly longer to learn (around 700 epochs).

This architecture is inefficient however. Given Equation. 3.1, we can see that, at every step, the network calculates a correct answer in a for *every* operator we wish to calculate. b functions to pull out the correct answer from this matrix. This means that a must scale with the number of operators we require. Through the process of *mixing* proposed in Section. 3.4 we address this inefficiency.

3.3 Memory

The next step in our project was to move away from simple bitwise operators and onto more complex operations. In particular, we would like to learn operations which require some level of memory. We focus on Addition, Subtraction, and Multiplication. What makes these tasks somewhat more difficult is the removal of one of the key assumptions of our previous tasks - independence between the bits in a sequence. Taking the example of binary addition of two bit strings, adding any two given digits may give a remainder, and the network has to *remember* this remainder in some fashion so that it can be applied in subsequent iterations to achieve a correct result. To this end, we modify the design of our switching module further to incorporate memory. As with the switching module output, this memory is dependent on

X: 1111110110010111000110010101010010001010100001110l
Y: 1110001111011001000011001100011101010011000100000l
Operator: NAND
Expected: 0001111001101110111101111011101111111101111111110
Output: 0001111001101110111101111011101111111101111111110
Correct: True

X: 1011100110010010010110011101110110010101111000100l
Y: 1101010011100101100111101011110000011010100101010l
Operator: XNOR
Expected: 1001001010001000001110001001111001110000100010001l
Output: 1001001010001000001110001001111001110000100010001l
Correct: True

X: 11100011000111011011000101100111001101100110110000
Y: 0001001010111000111100110111010111101111010010001l
Operator: AND
Expected: 00000010000110001011000101100101001001100100100000
Output: 00000010000110001011000101100101001001100100100000
Correct: True

Figure 3.4: Representative Example Outputs from the Recurrent Neural Network, augmented with switching modules. It can be seen that the network generalises to sequences longer than those in the training set.

the operator given to the network allowing different operators to remember different values. Equations 3.2 gives a mathematical definition of the proposed layer for this problem.

$$
\begin{aligned}
a_t &= \sigma((X_t, c_{t-1}) \cdot W) \\
b_t &= \sigma((X_t, c_{t-1}) \cdot W') \\
g_t &= \sigma(o_t \cdot W'') \\
c_t &= b \odot g \\
out_t &= a \odot g
\end{aligned}
\tag{3.2}
$$

Here c_{t-1}, a $1 \times H$ vector, is the "memory" from the previous iteration of the network. Figure. 3.6 gives this memory network as a diagram.

Because the length of the output from an RNN must equal the sequence length, we encounter an interesting problem - any addition between two n bit numbers can produce an $n+1$ bit sequence. In order to get around this, we instead do a fixed width addition, truncating any bits which fall out of our original length. Similarly, for subtraction, we enforce that for any operation $x - y$, $x > y$, so as to avoid representation problems with negative numbers. We train the network on sequence lengths of between 2 and 10 bits, and test on sequences between 50 and 5000 bits long, to ensure that the system is able to generalise rather than just memorise the correct outputs.

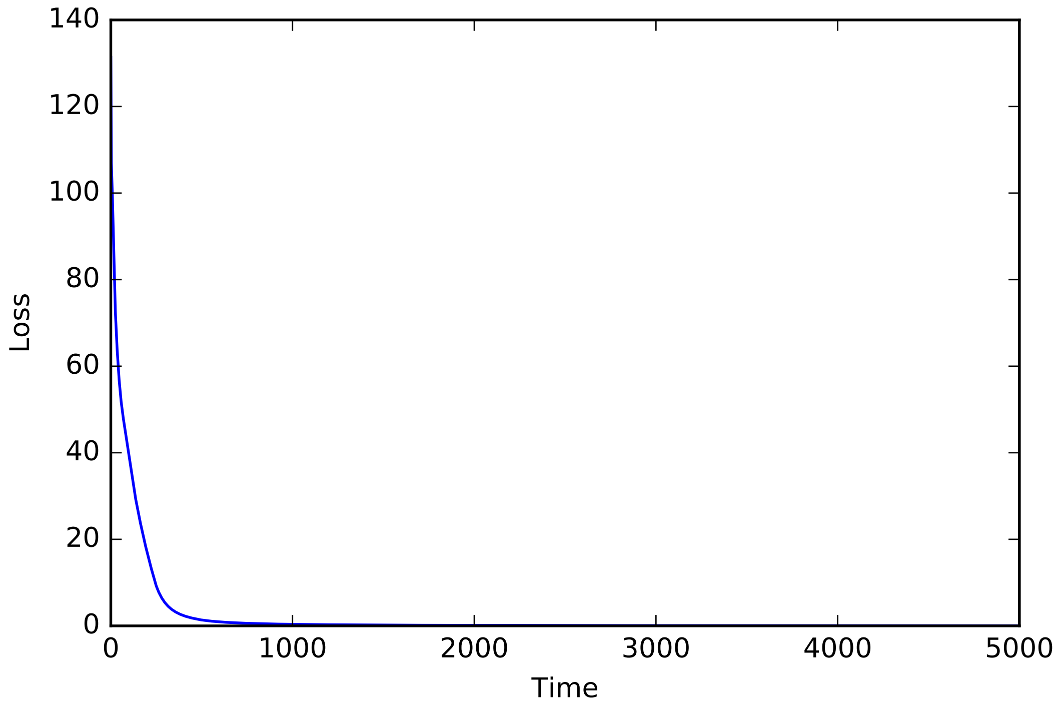


Figure 3.5: The softmax error rate of the sequenced version of the switching network. The network learns slightly slower than the original version given in Fig. 3.3

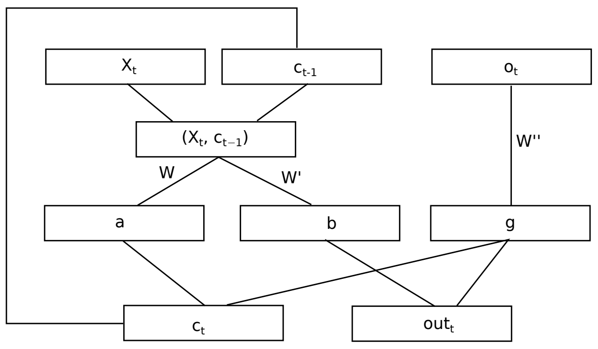


Figure 3.6: The network diagram for the proposed memory network

After training a network with two of these layers, with 30 and 10 hidden units respectively for 5000 epochs using the Adagrad method with a learning rate of 0.2, we present some representative examples from the network in Figure. 3.7. The network is able to learn both addition and subtraction on reasonably long sequences, despite not being trained on such sequences. On the other hand, the network fails in the multiplication of binary sequences. We hypothesise that this is due to the architectural problems of feed forward networks encountered in the literature.

In Fig. 3.8, we present a comparison between the switching network on all tasks, the switching network without trying to learn multiplication, and a standard feed forward neural network on both these tasks. A peculiar feature of this problem is that each of the operations we learn are reasonably independent. This means that each operator can be learned independently. As such, we find that algorithms tend to perform well in the early stages of learning, as they learn the operators that they can, and then plateau, failing consistently on the operators that they cannot learn. This can be seen in the standard feed forward network, as well as our sequence based network when introducing multiplication. Removing multiplication allows our switching network to converge properly.

X: $01000000110001101110010101100101001010101101010111101001101110_2$
Y: $10100011000001111010010000110010100110110111110110010011110100 1_2$
Operator: ADD
Expected: $100100100000011011001100111010101011101000001101111001011000100_2$
Output: $100100100000011011001100111010101011101000001101111001011000100_2$
Correct: True

X: $10110110100110111110011111111110010110101111110101000011010111 0_2$
Y: $01110011011100101111010111110000111011001000010100000001100110 10_2$
Operator: SUBTRACT
Expected: $010000110010100011110010000011010110111001111000010000000010100_2$
Output: $010000110010100011110010000011010110111001111000010000000010100_2$
Correct: True

X: $11000000000011100111010101111010000110110001110011100101101001 0_2$
Y: $01111011011000110011011001111001111101011100110001110011101100 0_2$
Operator: MULTIPLY
Expected: 101110010010001011000001110011110100101010000111101111110101111
Output: $101100001000100010011110110011101011011100100001110000110000001_2$
Correct: False

Figure 3.7: Examples of output from our memory based switching network

Multiplication itself is a surprisingly difficult problem for neural networks. In operations such as addition and subtraction each bit only directly affects its pair in the other sequence. However in multiplication, every bit in one sequence affects every bit in the other. For example, if we consider a multiplication by 2, every bit in the second sequence has to be shifted left one. This model of problem - where a single entry in the sequence affects every other - is not well suited for standard feed forward architecture in which the network is only

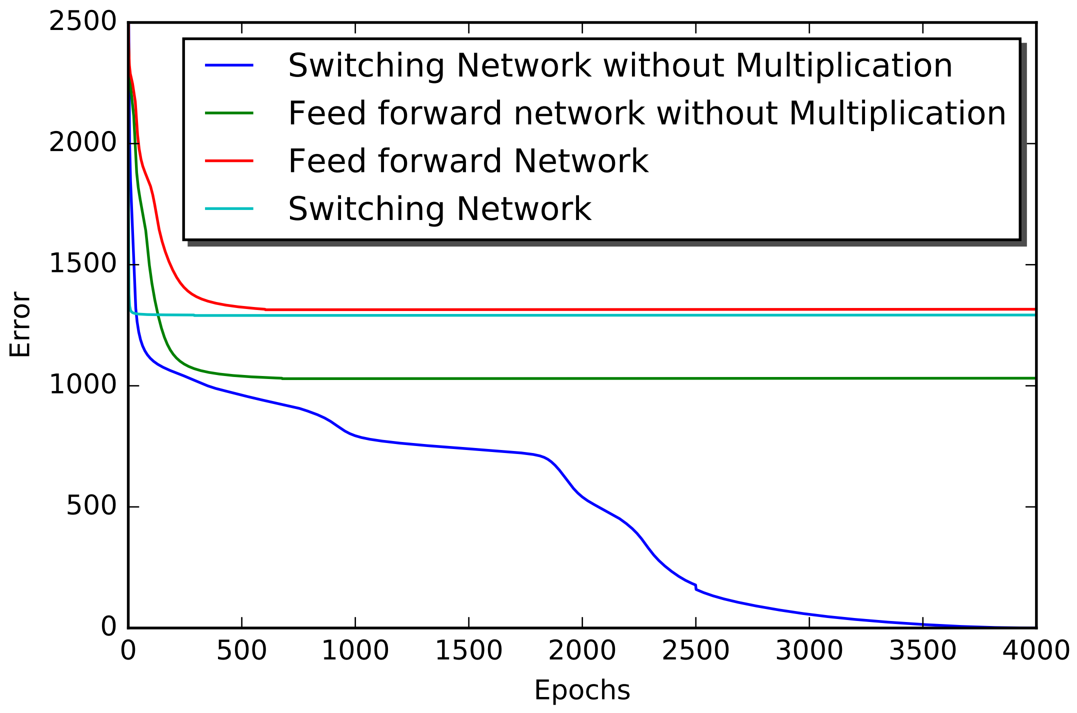


Figure 3.8: The softmax error rate of the memory switching network

presented with each bit pair in the input sequence once.

In Chapter 4 we address how to solve this problem, by applying the switching module to the established Neural GPU model.

3.4 Mixing

As mentioned previously, our current representation of operations builds orthogonality into the vector representations of the operators. This results in an inefficient solution to the problem of learning multiple operators, in that it is possible for the network to simply use the naive solution of utilising different rows of W and W' for each problem (Fig. 3.2). This is equivalent to learning a single network for each different operator. Clearly there are some problems in which this results in duplication of effort (e.g. binary operators and their negatives will be learned seperately, as if they were completely independant, when they clearly are not).

Instead, we would like to calculate only the required value to output and remember at each step. To this end, we redefine W and W' as $Z \times (F_r + H) \times 1$ vectors (previously they are defined as $(F_r + H) \times 1$ matrices), and add two new parameter matrices to our network - M and M', two $F_o \times Z$ matrices. Here Z is an integer, less than the number of operators we have. By mixing these Z $(F_r + H) \times 1$ vectors together we can construct weights matrices for each operator. In effect, instead of transforming the input vectors and then pulling the answer we want out of the resultant matrix, we calculate the transformation from the operator vector (Instead of learning it directly). Applying this transformation allows us to calculate only the answer for the operation. With these parameters, we define our new switching module in Equation 3.3.

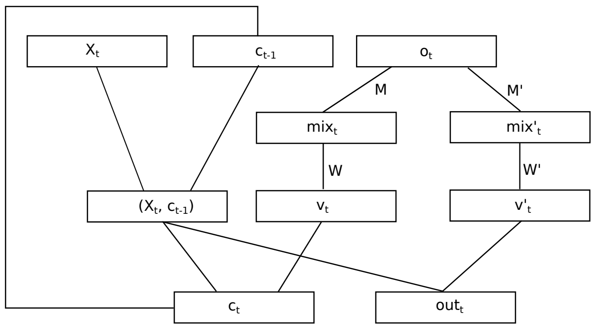


Figure 3.9: The network diagram for the proposed mixing network

$$\begin{aligned}
\text{mix} &= o_t \cdot M \\
\text{mix}' &= o_t \cdot M' \\
v &= \sum_i mix_i \odot W_i \\
v' &= \sum_i mix'_i \odot W'_i \\
c_t &= \sigma((X_t, c_{t-1}) \cdot v) \\
out_t &= \sigma((X_t, c_{t-1}) \cdot v')
\end{aligned} \tag{3.3}$$

In this way, we define mix and mix$'$ as mixing vectors which are the used to calculate weights matrices v and v' to pull from W_i and W'_i.

Fig. 3.9 gives a graphical representation of this network. It is useful to compare this diagram to Fig. 3.6. It can be seen that in the mixing model instead of learning W and W' to transform the operands, we use W, W' and the mixing vectors calculated from the operators to produce the transformations on the input v and v'.

Chapter 4

Switching Neural GPUs

Following on from Chapter. 3, in this chapter we detail efforts to learn operations that are much more complicated, such as Multiplication. In order to do this, we modify the existing Neural GPU model which has been shown by Kaiser & Sutskever [15] to be able to solve this problem. In fact, the standard Neural GPU model is the *only* model to date which is able to solve this problem in the general case (the ability to generalise to sequences thousands of bits long).

4.1 Multiplication

One of the operations which our model fails on is multiplication. This problem has been attempted before in the literature [15, 13, 14] and exposes a fundamental problem with our feed forward architecture. A Feed Forward Neural Network can only solve problems which require at most one pass of the input, and constant memory. Multiplication and Division do not meet these requirements, both requiring at least O(log n) passes through the input [15]. As such, in roder to learn these operators successfully a much richer architecture is required.

The standard Neural GPU model[21], constructed of two "Convolutional Gates Recurrent Units"(CGRU) is given in Equation 4.1.

$$\begin{aligned} \mathrm{CGRU}(s) &= u \odot s + (1-u) \odot \tanh(W * (r \odot s) + B) \\ u &= \sigma(U' * s + B') \\ r &= \sigma(U'' * s + B'') \\ U * s[x,y,i] &= \sum_{u=\lfloor -k_w/2 \rfloor}^{\lfloor k_w/2 \rfloor} \sum_{u=\lfloor -k_h/2 \rfloor}^{\lfloor k_h/2 \rfloor} \sum_{c=1}^{m} s[x+u, y+v, c] \cdot U[u,v,c,i] \end{aligned} \tag{4.1}$$

Note that $U * s$ represents a convolution of s, by the convolution bank U. In this model, each layer has three parameters (U, U' and U'') which serve as four dimensional "banks" of convolutions. At each step these are applied to convolve the neural model (the current state of the system, s) to the next iteration. We will now alter the architecture of the Neural GPU described in Section. 2.6, and utilising the ideas of our mixing model in Section 3.4. Instead of learning a specific U, U' and U'' for our problem, we instead define these as five dimension tensors. Using the operator as before, we can construct a mixing vector for each problem in order to calculate specific values for each. The revised version of this model is given in Equation 4.2.

$$
\begin{aligned}
\text{mix} &= o_t \cdot M \\
\text{mix}' &= o_t \cdot M' \\
\text{mix}'' &= o_t \cdot M'' \\
V &= \sum_i \text{mix}_i \times U_i \\
V' &= \sum_i \text{mix}'_i \times U'_i \\
V'' &= \sum_i \text{mix}''_i \times U''_i \\
\text{GRU}(x, s) &= u \odot s + (1 - u) \odot \tanh(W * (r \odot s) + B) \\
u &= \sigma(V' * s + B') \\
r &= \sigma(V'' * s + B'') \\
V * s[x, y, i] &= \sum_{u=\lfloor -k_w/2 \rfloor}^{\lfloor k_w/2 \rfloor} \sum_{u=\lfloor -k_h/2 \rfloor}^{\lfloor k_h/2 \rfloor} \sum_{c=1}^{m} s[x+u, y+v, c] \cdot U[u, v, c, i]
\end{aligned}
\tag{4.2}
$$

Here mix, mix$'$ and mix$''$ form the mixing vectors which are used to pull the appropriate convolutions from our U tensors. Note the differences between Equation. 4.2 and Eq. 4.1. Instead of learning the convolution banks directly, we compute them based on a transformation of the input operator o. This is analogous to the transformation made in Section 3.4 to the feed forward model and accomplishes two things:

1. Allows the new Switching Neural GPU model to learn multiple operators
2. Forces the Switching Neural GPU to utilise the similarities between operators and effectively use the space available to it

These two enhancements effectively bring the Neural GPU model in line with the feed forward model introduced in 3.4. To test this new model, we apply similar tests as before. We can now however introduce new operators (i.e. $\times$). It is worth mentioning that some of the operators we evaluate our system with can produce outputs longer than the input sequences (e.g. adding two n bit numbers can produce an $n + 1$ bit number). In this model, as with the previous, we simply extend the input sequence to be as long as the maximum length of the output by padding the extra size with 0's. We present the loss over time during the training of this model in Fig. 4.2.

From Fig. 4.2 we can see that the system converges, having learned the operators presented to it. In contrast to the learning models presented in Chapter. 3 however, this model takes significantly longer to converge. Both the standard Neural GPU model and our own switching Neural GPU took in the order of 24 hours in order to learn the multiplication problem. Fig. 4.1 gives some example inputs and outputs to this new model after training it. It can be seen from Fig. 4.1 that the system is able to generalise to sequences far longer than those which it is trained on. The system is trained on sequences up to 20 bits long. The presented examples represent 63 bit multiplication, howeverm we evaluate the system on bit sequences up to 2000 bits long with the same success.

4.2 Tree Parsing

Extending our previously developed model to be able evaluate these trees is done as follows: Consider our above modified Neural GPU model as a function $f(x, y, o)$, which takes two

X: 100110100100111111110000000101011011001100110000111011000011001
Y: 001101000100010001100000011111100011001000011111101000010010011
Operator: MULTIPLY
Expected: 111111000000010111010000101010001110011100000001000010010001110111011
01011010101000101010010100101110111101110010000001011011
Output: 111111000000010111010000101010001110011100000001000010010001110111011
01011010101000101010010100101110111101110010000001011011
Correct: True

X: 11000100001100010111111000011110110011010101101000001110101110
Y: 111001101111001001000000001111001000110101011111100111010111
Operator: MULTIPLY
Expected: 1011000011111110000111110101001110101101000101101101010001100110110
1001101000010010100110001000011100000101101010100100010
Output: 1011000011111110000111110101001110101101000101101101010001100110110
1001101000010010100110001000011100000101101010100100010
Correct: True

X: $110000000000111001110101011110100001101100011100111001011010010_2$
Y: $011110110110001100110110011110011111010111001100011100111011000_2$
Operator: ADD
Expected: 101110010010001011000001110011110100101010000111101111110101111
Output: 101110010010001011000001110011110100101010000111101111110101111
Correct: True

Figure 4.1: Examples of output from the Switching Neural GPU model

values x and y and applies an operator o to them. If we consider a parse tree t with a non-leaf node n, then n has two children, c_1 and c_2 (we limit t to containing only binary operators) and n itself represents an operator o. Thus the subtree $\langle n, c_1, c_2 \rangle$ represents an instance of $f(c_1, c_2, o)$, where c_n can be either leaf nodes (numerical values) or another non leaf node in the tree. Considering this we can define a function $g(t)$ which takes a tree and returns a function f_t which is the result of a bottom-up breadth-first iteration through the tree t, replacing each non leaf node with an f. Consider the example parse tree given in Fig. 4.3 g will iterate through the tree in the order a, b, c. First g encounters node a. In this case, $c_1 = 7$, $c_2 = 3$ and $o = +$, giving the substitution for node a $f(7, 3, +)$. Similarly for node b, we get an output of $f(5, 2, -)$. When g encounters c, $c_1 = f(7, 3, +)$(node a), $c_2 = f(5, 2, -)$(node b), and $o = \times$. Thus the final output of $g(t)$ is $f(f(7, 3, +), f(5, 2, -), \times)$.

In Fig. 4.4 we give examples of the solutions produced by the above network after having been trained using the Adagrad learning method with a learning rate of 0.2.

For shallow trees (i.e. up to depth 2), this works well, consistently giving correct results. If we look at the last entry in this Table 4.6 however, we can see that when we get to more complicated trees with many different operators, we start to get inaccuracies in the results produced by the network. In fact, in our experiments with very large trees (up to depth 16, i.e. 65536 leaf nodes and 65535 equations) we see many bit errors. This is possibly due to the

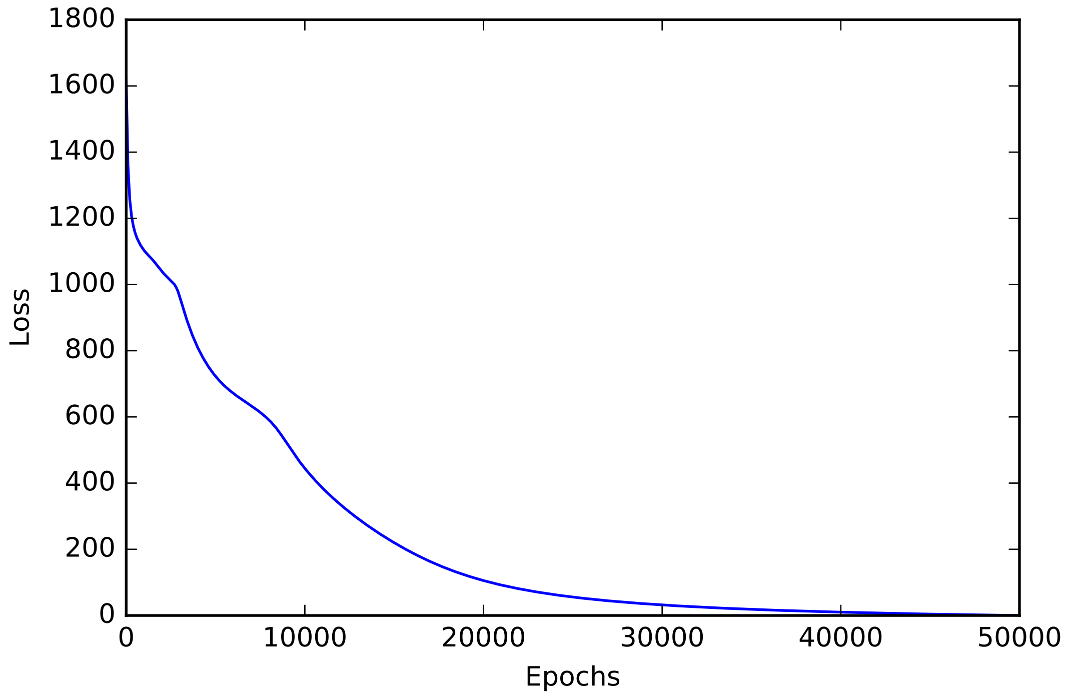


Figure 4.2: The loss over time for learning multiplication with out Switching Neural GPU

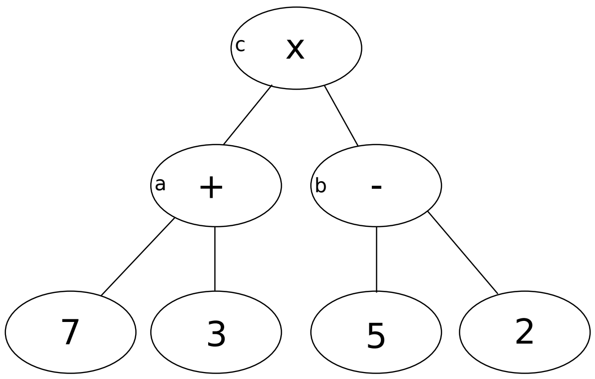


Figure 4.3: An example numerical parse tree

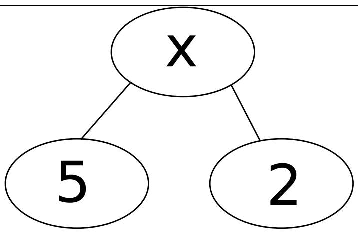
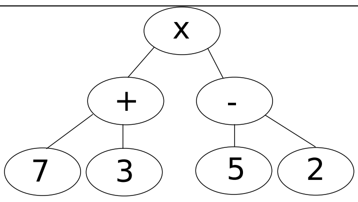
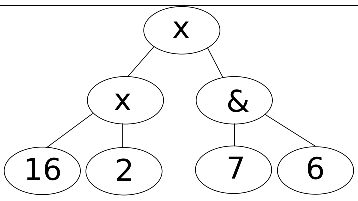
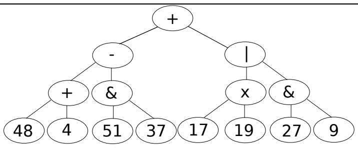
Parse Tree	Expected Result	Result	Correct
	01010	01010	Yes
	11110	11110	Yes
	11000000	11000000	Yes
 + - | + & x & 48 4 51 37 17 19 27 9	101011110	100111110	No

Figure 4.4: Some example parse trees and their outputs

error signal delay inherent in these problems - the system must evaluate *all* of the equations in the tree before receiving an error signal. This means that any errors near the base of the tree can cause large errors in higher levels. Fig. 4.5 gives the training error over time for learning this problem.

It can be seen in Fig. 4.5 that the overall error eventually converges. We do see however that the system suffers from the long training times of the previous Switching Neural GPU experiments, given in Section. 4.1.

Fig. 4.6 gives the average number of bit errors across 100 problems at each of the given tree depths.

We can see in Table. 4.6 that, as we increase the depth of the tree we feed into the network, we start to see larger and larger innacuracies in the results produced for the tree.

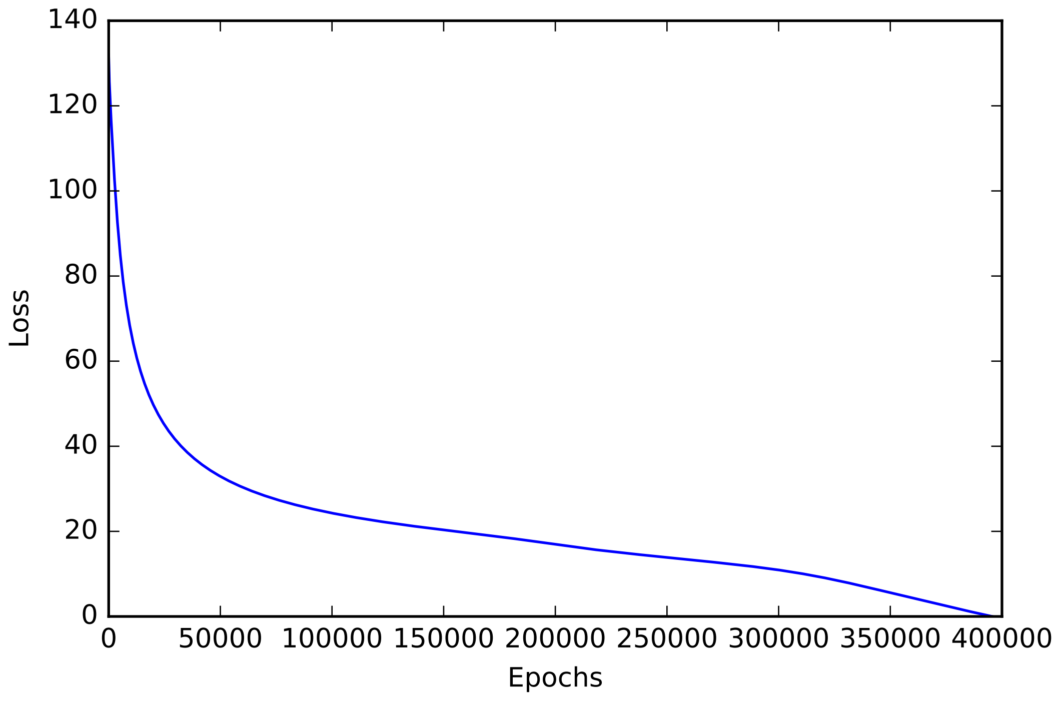


Figure 4.5: The loss over time for the Switching Neural GPU when learning tree parsing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0.33	0.56	1.15	1.85	2.12	3.24	3.12	3.25	3.44	4.12	4.34	4.86	5.03	5.14

Figure 4.6: The average error rate at different tree depths

Chapter 5

Analysis

In this section, we present an analysis of some of the parameters learned by our mixing network, introduced in Section. 3.4. In doing so, we shed some light on the inner workings of the new architecture.

5.1 How Mixing Works

In Section 3.4 we introduced the idea of mixing as a performance improvement to our original model. The idea is that by forcing the network to mix weights matrices together in order to produce the final transformation of the input, we enable the network to utilise similarities between operators in order to learn them. Fig. 3.9 gives the network architecture of the mixing switching module. It can be seen in this Figure that the operator is transformed into two mixing vectors - one for calculating the output of the module, and the other for calculating the value to remember. It stands to reason that similar operators should produce similar mixing vectors. If this is indeed the case then we can show that the operations learned by the network are being compressed in an interpretable way.

5.2 Logical Operators and their negations

The most obvious place to look for these dependencies is in the logical operators we learn. We can consider the AND and NAND operators in one run of the mixing network introduced in Section 3.4. Recall that the mixing vector for an operation specifies how the learned matrices are combined in order to produce the final transformation of the input to be outputted. The mixing vectors for these operations are $\langle 0.27, 0.51, 0, 0, 0, 0.14, 0, 0.08\rangle$ and $\langle 0.18, 0.57, 0, 0, 0.17, 0.14, 0, 0\rangle$ respectively. These two vectors are very similar, which indicates that the network performs a similar computation for each of these operators, with NAND utilising the 5th matrix instead of the 8th used in the AND operator. A similar behavior is produced by and OR and NOR operators which produce mixing vectors of $\langle 0.64, 0, 0, 0.23, 0, 0.06, 0.07, 0\rangle$ and $\langle 0.47, 0, 0, 0.16, 0.30, 0.04, 0.03, 0\rangle$ wherein the two vectors are again very similar, save that NOR uses the 5th matrix as well. Considering that both NAND and NOR use the 5th matrix, if we look at the mixing vector produced for the NOT operation ($\langle 0, 0, 0, 0.03, 0.78, 0, 0.09, 0.1\rangle$) we can infer that the 5th matrix holds some level of the logic for performing a negation. The vectors for these operations can be seen graphically in Fig. 5.1.

In Fig. 5.1 we can visually see the similarities between the mixing vectors for the logical operators. We can clearly see the similarities between operators such as AND and NAND, OR and NOR, and XOR and OR. Interestingly we can also see that quite a few of the matrices

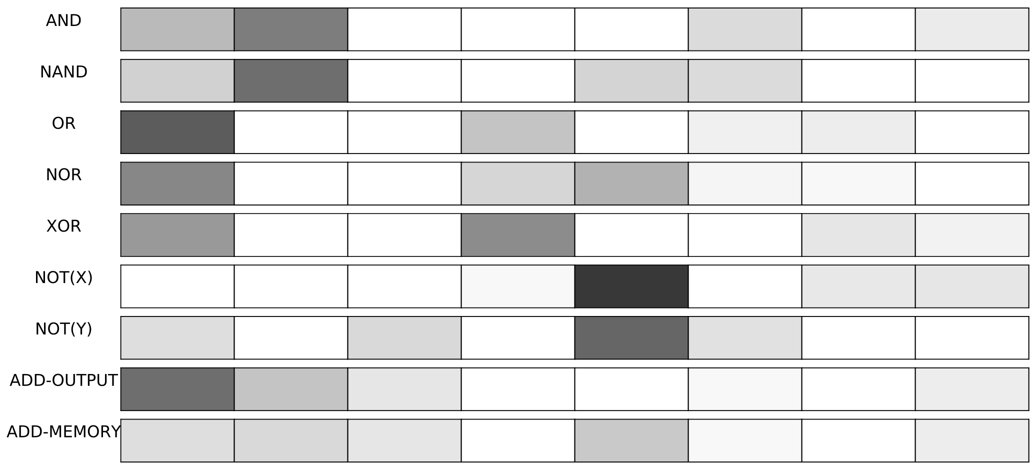


Figure 5.1: The mixing vectors for the operations in the mixing network. Darker squares indicate a heavier reliance on the values in the corresponding matrix

are very litely used by many of the operators, unlike matrix 5 which is used heavily by only a couple. This implies that instead of specialising, these matrices instead hold more general information which is applicable to lots of operators.

5.3 Similarity Measures

We can draw a parallel here to the idea of Word2Vec proposed by Mikolov et al [16], a system designed to calculate continuous vectors from words. In that paper the authors demonstrate that the vectors produced by their learned function have an additive property: they can be added and subtracted to produce semantically meaningful results. For example, given the learned function f which takes a word and produces a continuous vector representation of it, the authors show that $f(\text{Berlin}) - f(\text{Germany}) + f(\text{France})$ gives a vector very similar to $f(\text{Paris})$. Such a property lends itself to the interpretatability of the learned operators and their similarties.

If we take the vectors learned by our system, we can show that a similar property appears in our representations of operations. For example, taking the NAND and NOT(X) mixing vectors given above, if we subtract them we get $\langle 0.27, 0.51, 0, -0.03, -0.61, 0.14, -0.09, -0.1\rangle$. Taking the cosine similarity of this vector with the AND vector, we get a value of 0.67, a higher similarity than any of the other operators mixing vectors in the system. We can perform a similar operation for the NOR operator, subtracting the NOT(X) mixing vector to get $\langle 0.47, 0, 0, 0.13, -0.48, 0.04, -0.06, -0.1\rangle$ which gives a cosine similarity of 0.69 with the OR operator again higher than any other operator.

Perhaps a more interesting example is the XOR operator. Taking the mixing vector for the XOR operator ($\langle 0.40, 0, 0, 0.45, 0, 0, 0.10, 0.05\rangle$) and adding the AND operator gives a result of $\langle 0.67, 0.51, 0, 0.45, 0, 0.14, 0.1, 0.13\rangle$ which gives a cosine similarity of 0.80 compared to the OR operator.

Table. 5.1 gives a full listing of the cosine similarities between these transformations and the original operators.

In Table. 5.1, it can be see that for the logical operators we learn this additive property

	AND	NAND	OR	NOR	XOR	NOT(X)	NOT(Y)	ADD-OUTPUT	ADD-MEMORY
ADD-OUTPUT	0.75	0.60	0.85	0.74	0.60	0.01	0.23	1.00	0.63
ADD-MEMORY	0.64	0.74	0.39	0.68	0.29	0.68	0.80	0.63	1.00
ADD-OUTPUT + AND	0.93	0.82	0.70	0.60	0.49	0.02	0.20	0.94	0.68
XOR + AND	0.80	0.69	0.82	0.69	0.81	0.05	0.17	0.83	0.57
NOT(X) + NOT(Y)	0.07	0.31	0.11	0.60	0.09	0.98	0.97	0.11	0.75
NAND - NOT(X)	0.67	0.49	0.18	-0.20	0.08	-0.71	-0.58	0.42	-0.08
NAND - NOT(Y)	0.70	0.57	0.07	-0.25	0.04	-0.58	-0.58	0.32	-0.06
NOR - NOT(X)	0.30	0.02	0.69	0.24	0.55	-0.70	-0.50	0.60	-0.21
NOR - NOT(Y)	0.26	-0.00	0.72	0.31	0.67	-0.56	-0.51	0.55	-0.22
AND - NOT(X)	0.59	0.36	0.25	-0.19	0.14	-0.79	-0.66	0.44	-0.16
AND - NOT(Y)	0.63	0.42	0.16	-0.24	0.12	-0.71	-0.69	0.37	-0.16
AND - OR	0.43	0.54	-0.62	-0.53	-0.61	-0.01	-0.09	-0.20	0.17
OR - XOR	0.34	0.24	0.45	0.40	-0.04	-0.05	0.18	0.64	0.28

Table 5.1: Cosine Similarities between the operators learned and some transformations of those operators

holds quite well. In many of the transformations, the expected operator has the highest cosine similarity when compared to the other operators. It does not hold as well for our more complex operators. The addition operator has two relevant parts: calculating the bit to output and calculating the value to remember. Optimally, the former represents a three-way XOR between the two input bits and the memory from the previous element in the sequence (i.e. if exactly one, or all of the bits are 1 bits, the output is a 1) The latter represents a three-way AND operation (i.e. if more than one bit is 1, the output is a 1). We can take the cosine similarity between the mixing vector for addition which calculates the value to output for a given entry in the sequence ($\langle 0.57, 0.23, 0.1, 0, 0, 0.03, 0, 0.7 \rangle$) and the XOR operator we get a value of 0.60, which is less than this vectors similarity to other operators, namely AND, NAND, OR and NOR all have higher cosine similarties. Similarly if we add the AND operator vector as we did with the XOR vector above, and take a cosine similarity of the result with the OR operator we get a value of 0.70, again less than the similarity to AND and NAND. Because of this, we can infer that despite being functionally similar, the system learns the addition operations independantly of the logical ones.

It is also interesting to note that some transformations seem to drawing meaning where there is none. Taking some non-sensical transformations, we can see that the resultant transformed vectors have large cosine similarities to other operations, despite this not making sense. Taking the example of NOT(X) - NOT(Y) for example, we can see in Table. 5.1 that this transformation has a high similarity to both of the NOT operators, and the memory operation for addition. The transformation AND - OR shows this same behaviour, in that it does not have a similarity with any of the operators except for AND and NAND which is non intuitive.

Chapter 6

Conclusions

In this report we have followed a step by step process to develop the first neural networks which are able to simultaneously learn a number of different algorithmic operators. The main innovation which has made this possible is the Switching Module - an internal module which can be applied to any neural network structure.

We first applied this Switching Module to a standard feed forward neural network, showing that the network is able to simultaneously learn a number of logical operations on binary strings, and to do this faster than an equivalent feed forward network without this switching module. We then extended this network to perform sequence-to-sequence learning, proving its ability to generalise to arbitrarily long bit sequences while maintaining accuracy. Thirdly, we applied this switching module to a recurrent neural network in order to solve more complex mathematical operations which require memory to be solved properly, such as binary addition and subtraction. We demonstrated that this augmented network has the ability to learn these operations, on top of the previously learned binary operations, whereas an equivalent standard recurrent neural network cannot. Using this model, we also show that both the augmented and standard recurrent neural networks display behaviour exhibited in previous work in that they cannot solve the multiplication problem [13, 14]. This is due to architectural limitations with these recurrent neural networks. In order to solve this problem, we modified the established Neural GPU model [15] which has been proven to be able to solve the multiplication problem . We augment this model with a switching module in order to create a model which is able to solve the multiplication problem, in addition to the operators learned by the previous networks. Finally we demonstrated that this learned model can be chained together in order to produce a model which is able to successfully evaluate mathematical parse trees.

We note however that chaining the model together in this way starts to fail in large parse trees. As the depth increases, we start to see an increasing number of bit errors. This presents a deficiency in generalisation ability for this task - errors which can be tolerable at lower depth trees can be compounded in larger trees to produce incorrect results.

We also proposed an improvement to the original switching module which allows more efficient internal representations of operations. This modification allows the switching networks to develop more space efficient representations than simply learning a single neural network for every operation, utilising the similar underlying subtasks of the algorithmic operations presented to it. We demonstrated, using the evaluation method proposed for the Word2Vec system [16], that these internal representations actually hold semantic meaning in that they can be combined through simple addition and subtraction to produce meaningful results showing that the learned internal primitives are learned to represent common subtasks of the operators presented to the network.

Through this process we have taken a significant step towards to long term goal of developing Neural Networks which have the ability to *reason* in a compositional manner, and hense to make decisions in an algorithmic fashion. Neural Networks augmented with switching modules offer, for the first time, networks which are able to change the operation they perform on the fly. These models can feasibly be conjoined with the perceptual neural networks which currently hold the state of the art. Should it be developed such a system would produce *both* the features and operations in order to produce structures which are able to reason from raw environmental inputs: A solution to the Symbol Grounding problem.

Bibliography

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] S. Ho, M. Xie, and T. Goh, "A comparative study of neural network and box-jenkins arima modeling in time series prediction," *Computers & Industrial Engineering*, vol. 42, no. 2, pp. 371–375, 2002.

[3] X. Cai, N. Zhang, G. K. Venayagamoorthy, and D. C. Wunsch, "Time series prediction with recurrent neural networks trained by a hybrid pso–ea algorithm," *Neurocomputing*, vol. 70, no. 13, pp. 2342–2353, 2007.

[4] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3128–3137, 2015.

[5] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille, "Deep captioning with multimodal recurrent neural networks (m-rnn)," *arXiv preprint arXiv:1412.6632*, 2014.

[6] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2625–2634, 2015.

[7] 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*, pp. 6645–6649, IEEE, 2013.

[8] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," *arXiv preprint arXiv:1402.1128*, 2014.

[9] P. Jyothi, L. Johnson, C. Chelba, and B. Strope, "Distributed discriminative language models for google voice-search," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5017–5020, March 2012.

[10] S. Chopra, M. Auli, A. M. Rush, and S. Harvard, "Abstractive sentence summarization with attentive recurrent neural networks,"

[11] L. Kaiser and I. Sutskever, "Googles neural machine translation system: Bridging the gap between human and machine translation," 2016.

[12] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba, "Sequence level training with recurrent neural networks," *arXiv preprint arXiv:1511.06732*, 2015.

[13] W. Zaremba and I. Sutskever, "Learning to execute," *CoRR*, vol. abs/1410.4615, 2014.

[14] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus, “Learning simple algorithms from examples,” *CoRR*, vol. abs/1511.07275, 2015.

[15] L. Kaiser and I. Sutskever, “Neural gpus learn algorithms,” *CoRR*, vol. abs/1511.08228, 2015.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[17] A. Graves, G. Wayne, and I. Danihelka, “Neural turing machines,” *CoRR*, vol. abs/1410.5401, 2014.

[18] S. Harnad, “Symbol-grounding problem,” *Encyclopedia of cognitive science*, 2003.

[19] C. Dyer, A. Kuncoro, M. Ballesteros, and N. A. Smith, “Recurrent neural network grammars,” *CoRR*, vol. abs/1602.07776, 2016.

[20] 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*, pp. 6645–6649, May 2013.

[21] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[22] S. Riedel, M. Bošnjak, and T. Rocktäschel, “Programming with a differentiable forth interpreter,” *arXiv preprint arXiv:1605.06640*, 2016.

[23] S. R. Bowman, C. Potts, and C. D. Manning, “Recursive neural networks for learning logical semantics,” *CoRR*, vol. abs/1406.1827, 2014.

[24] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.