Differentiable Harvard Machine Architecture with Neural Network Controller

by

Manthan Bharat Bhatt

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2020 by the
Graduate Supervisory Committee:

Heni Ben Amor, Chair
Yu Zhang
Yezhou Yang

ARIZONA STATE UNIVERSITY

May 2020

ABSTRACT

There have been multiple attempts of coupling neural networks with external memory components for sequence learning problems. Such architectures have demonstrated success in algorithmic, sequence transduction, question-answering and reinforcement learning tasks. Most notable of these attempts is the Neural Turing Machine (NTM), which is an implementation of the Turing Machine with a neural network controller that interacts with a continuous memory. Although the architecture is Turing complete and hence, universally computational, it has seen limited success with complex real-world tasks.

In this thesis, I introduce an extension of the Neural Turing Machine, the Neural Harvard Machine, that implements a fully differentiable Harvard Machine framework with a feed-forward neural network controller. Unlike the NTM, it has two different memories - a read-only program memory and a read-write data memory. A sufficiently complex task is divided into smaller, simpler sub-tasks and the program memory stores parameters of pre-trained networks trained on these sub-tasks. The controller reads inputs from an input-tape, uses the data memory to store valuable signals and writes correct symbols to an output tape. The output symbols are a function of the outputs of each sub-network and the state of the data memory. Hence, the controller learns to load the weights of the appropriate program network to generate output symbols.

A wide range of experiments demonstrate that the Harvard Machine framework learns faster and performs better than the NTM and RNNs like LSTM, as the complexity of tasks increases.

DEDICATION

To my parents.

ACKNOWLEDGEMENTS

I want to thank Dr. Heni Ben Amor for guiding me through-out my journey and supporting me at every step. I have had a wonderful learning experience working with you.

I also want to thank the Dr. Yu Zhang and Dr. Yezhou Yang. I am honored to have you on the committee for my thesis.

Lastly, I want to thank my parents and my friends who have shown immense support at all times during this journey. It would not have been possible without all of you.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The human brain processes sequential data received through all the senses, in real time. These signals can be from multiple domains - notes of a song playing or images of a baseball game. From visual senses, the visual cortex can identify edges, judge depth and differentiate objects. Differentiating and identifying phonemes from a continuous stream of auditory signals is executed by the auditory cortex. Therefore, to achieve human-level cognition in machines, processing sequences and finding useful representations of sequences is essential.

Multi-layer perceptrons, or feed-forward neural networks, have achieved success in a wide-range of supervised learning problems in computer vision (Khan *et al.* (2019)). For a given input X, a neural network classifier generates a probability distribution, $P(Y|X)$ over possible classes Y. As an example, from an x-ray image of a person's lungs, a feed-forward network can predict if the person has pneumonia.

In contrast to computer vision tasks, the output probabilities in sequence learning problems depend on the current as well as all the previous inputs. For example, take a look at the sentence *"It is December now and the weather is ___"*. If a neural network had to predict a word between *warm* and *cold* to complete the sentence, it would require information about all the previous words, especially the word *December*.

Recurrent Neural Networks (Elman (1990)), in contrast to feed-forward networks, have a memory unit coupled with an activation unit in each neuron. These memory units can maintain state over time, storing valuable temporal relationships in sequences. RNNs have been noteworthy in tackling problems in Natural Language Processing (Young *et al.* (2018)).

Tight coupling of memory and computation units poses a challenge. For representing long sequences, the computation cost, and as a result the training time, increases exponentially. To counter this, a class of recurrent neural networks, called Memory Augmented Neural Networks (MANNs) were introduced (Ma and Principe (2019)). A MANN has a neural network controller that interacts with an external memory as it parses through a sequence. As discussed in the next section, many MANN frameworks have been proposed so far that have achieved success in a broad range of sequential problems.

In this thesis, I propose a Memory Augmented Neural Network that can capture and represent long-term relationships in a sequence and test this model on a wide range of mathematical and robotic experiments. The architecture is based on the Harvard Machine design (Broesch (2009)) that has two external memories - a read only memory and a read-write memory. This document is divided in five chapters - the first chapter outlines the research done so far in the field of MANNs and lists the motivation and key contributions of this work. Chapter 2 provides the reader with the background knowledge that will be required to fully understand the methodologies explained in this document. In Chapter 3, the architecture of the neural network model is explained in detail and Chapter 4 illustrates the experiments that were performed on the model and results obtained on them. Conclusions and directions of possible future research are listed in the last chapter.

1.1 Memory Augmented Neural Networks - Literature Review

Research on neural networks began in the second half of the twentieth century. However, until late 1980s, no such neural network architecture was developed that could extract and represent temporal relationships from sequences. The Simple Recurrent Neural Network (SRNN) introduced by (Elman (1990)) was one the first

neural networks to claim success in this field. Unlike a feed-forward network, the neurons in the hidden layer of SRNN had recurrent connections; the activation of a hidden neuron at a particular time was a function of its previous activation. As a result, the network was able to store intermediate values, called *states*, that could represent temporal properties. This opened many research areas in the field of sequence learning. Languages are the most common sequences that we encounter in daily life, and beyond any doubt, were of remarkable interest to researchers. Giles *et al.* (1992a) and Giles *et al.* (1992b) showed that RNNs could successfully be trained to represent regular languages. They had proven that they could capture the underlying automaton. Another important area in sequence learning is sequence transduction - transforming a given input sequence to an output sequence. A prototypical example of such a task is machine translation. Bourlard and Morgan (1993) demonstrated that with enough hidden layers, a recurrent neural network can translate long sequences with sparse temporal dependencies.

As discussed in the previous section, such RNNs (called deep RNNs) require enough number of hidden layers to store representations for the longest possible string. As computation units and memory are strongly coupled, this leads to an unwanted added computation cost for shorter strings. Also, for a linear increase in input length, the computation units, and hence computation cost, increase exponentially. Moreover, RNNs were only able to successfully learn regular languages - the first class of languages in Chomsky Hierarchy.

These were the main motivations for decoupling neural networks from memory and led to the first wave of neural network architectures with external memories in the early 1990s. The Neural Network Push Down Automaton (NNPDA) (S. Das and Sun (1992)) was the first work concerned with the inference of Deterministic Context-Free languages, only from training data sequences. The model has an exter-

nal stack memory integrated through a hybrid error function. The stack is continuous and hence could be trained by gradient-descent methods. Training this model was computationally heavy and unstable. S. Das and Sun (1993) put forth some methods to overcome the shortcomings of NNPDA by inducing *a priori* knowledge about the tasks during training like using incremental learning and biasing network with task-specific initial weights. Zeng *et al.* (1994) also proposed a neural network coupled with an external stack memory for learning deterministic context-free grammars. However, the proposed stack was discrete, unlike NNPDA. While the previous papers focused on learning push-down automata, Mozer and Das (1993) proposed a neural net architecture that learns to encode the structure of symbol strings via reduction transformations. An example of a reduction transformation is reducing English language sentences to noun phrases and verb phrases.

Following these works were two decades that saw little progress in this field. In the second decade of the 21st century, there was a sudden rise in neural network related research, especially in computer vision, due to availability of high performing GPUs. In 2014, two works were published that inspired research for following years in the field of Memory Augmented Neural Networks. These were Memory Networks (Weston *et al.* (2014)) and the Neural Turing Machine (NTM) (Graves *et al.* (2014)). Memory Networks were motivated by the fact that LSTMs performed poorly in capturing long-term dependencies in sequences. The framework has four modules - to convert input strings into an internal representation, to write this internal representation into a memory, to produce an output given a query and state of the memory at that time, and to convert the output to the desired format. The authors tested their model on question-answering tasks with 14 million statements and 35 million questions.

The Neural Turing Machine (Graves *et al.* (2014)) aimed to train a neural network model to perform algorithmic tasks. It has a neural network controller, either feed-

forward or recurrent, that interacts with an analog memory through continuous read and write heads. This architecture was an implementation of the Turing Machine (Turing (1936)). However, as the memory read and write operations are also continuous and differentiable, unlike the Turing Machine, the entire architecture could be trained by gradient descent based methods. The NTM out-performed LSTM on algorithms like copy, repeat copy, associative recall and priority sort.

Another noteworthy paper published during the same time was Joulin and Mikolov (2015). Reviving the work done in 1990s (S. Das and Sun (1992), S. Das and Sun (1993), Zeng *et al.* (1994), Mozer and Das (1993)), the paper investigated the limitations of Recurrent Neural Networks in sequence transduction problems. They proposed a model with an external continuous stack that outperformed RNNs for a range of sequences. They extend the idea of a differentiable stack to a queue and a dequeue (double ended queue). Just like a stack, the queue has a push and a pop operations where the pop occurs at the bottom of the stack. The deque too has the same operations, but the states of read and write heads are represented by 2-dimensional vectors; the extra dimension indicates where to push or pop from (*top* *bottom*). The results on tasks like Bigram Flipping, Subj–Verb–Obj to Subj–Obj–Verb conversion and Gender Conjunction showed that the queue and dequeue generalize better than the stack and much better than Deep RNNs, especially for longer sequences.

In the next couple of years, researchers worked on enhancing/extending the capabilities of these MANNs. Memory Network (Weston *et al.* (2014)) described earlier required supervision at each of its modules and could not be trained end-to-end. Sukhbaatar *et al.* (2015) presented a version of Memory Networks that could be trained end-to-end through back-propagation. While Memory Networks used discrete *max* operators to generate the most similar feature representation for a question, End-To-End Memory Networks (Sukhbaatar *et al.* (2015)) replaced these operators with

a Softmax layer. Instead of having soft attention mechanisms to address the memory in NTM, Zaremba and Sutskever (2015) propose discrete weightings for the read and write head. As the architecture cannot be trained end-to-end by back-propagation, they use Reinforcement Learning to train the addressing mechanisms. Their model, Reinforcement Learning Neural Turing Machine, can be extended to other discrete interfaces like a search engine or a database. Another notable work, the Differentiable Neural Computer (DNC) (Graves *et al.* (2016)) improves NTM in a few ways. First, it introduces a new method to address memory, dynamic memory allocation, to prevent interference of memory locations. Second, it uses free gates to free already used memory that is no longer required, and third, it maintains the order in which sequential information was stored, in a temporal link matrix. Sparse Access Memory (SAM) (Rae *et al.* (2016)) introduced a memory architecture similar to that of the NTM, but more scalable when the memory size increases. Firstly, the authors only focus on content-based addressing and secondly, unlike NTM whose weightings are a distribution over all the locations of the memory, SAM's read and write vectors are a distribution over a subset of all the memory locations. This subset is chosen by a combination of a nearest-neighbor and a least recently used algorithm. As a consequence, the forward and backward passes occur in $\Theta(logN)$ time and constant space is utilized per time-step.

More recent works have been in applications of MANNs in various fields, in improving the efficiency and ease of training of previous MANNs, and in the field of Memory Augmented Generative Models. Le *et al.* (2018a) proposed a Variational Memory Encoder Decoder(VMED). In their architecture, an external memory acts as an interface between an encoder and a decoder. The memory captures dependencies in latent variables across time-steps. The encoder writes multi-modal latent representations in the memory from which the decoder reads and constructs the in-

put. To model a multi-modal space, the authors use Mixture of Gaussians. Le *et al.* (2018b) leveraged external memory to fuse features from two views for multi-view sequential learning. Each view has its own controller and a memory, and the paper puts forth algorithms for early and late fusing features from the two memories. Based on Kanerva's sparse distributed model (Kanerva (1988)), Wu *et al.* (2018) presented a generative memory model. To reduce redundancy in storing latent information, the authors derived a Bayesian memory update rule that optimizes the trade-off between losing old values and writing new ones. Csordas and Schmidhuber (2019) overcame three shortcomings that were present in the DNC - having a key-to-key comparison in content based addressing instead of value-to-key by masking portions of values and generating keys, de-allocating memory with low usage counters so that they don't affect the results of content-based addressing, and sharpening links in the temporal link matrix. The Neural Stored-Program Memory (Le *et al.* (2020)) is a fully differentiable implementation of the Harvard Machine architecture. By having two memories, one for storing data and another for programs, the architecture claims to be truly universal. The controller and the data memory have function like in an NTM. However, unlike the NTM where only one program(behavior) is implemented by the controller, the controller in Le *et al.* (2020) can implement multiple programs. The parameters of these programs are stored in the program memory. Depending on the state of the memory and the input at any time-step, the controller can switch between its behaviors.

1.2 Motivation and Contributions

In mathematics, often a complex computation can be thought of as a repetitive combination of simple computations. Take for example the task of finding the n^{th} term in an Arithmetic-Geometric Progression (A.G.P.). It may seem overwhelming

at first, especially when compared to simple mathematical operations like addition and multiplication. However, a closer look at the terms of an A.G.P.

$$a,\ (a+d)r,\ (a+2d)r^2,\ (a+3d)r^3, ..., [a+(n-1)d]r^{n-1}$$

suggests that it can be represented by a series of addition and multiplication operations. Another example is the multiplication operation itself which can be expressed by repeated additions.

When building a convoluted software, engineers divide it into atomic, reuse-able modules called functions. These functions are then called from a controller, or the *Main* class, which implements an algorithmic outline of the business logic.

In the field of robotics, particularly in reinforcement learning, tasks are sequential in nature, i.e. the agent gets a single input from the environment at each time-step. More often than not, these tasks can also be broken into smaller sub-tasks. Examine a robot that moves around in a park, picks up trash and throws it in a trash-can. To train such a robot is intricate. However, different modules can be trained to recognize trash, pick an object, navigate to the nearest trash-can and drop an object. Training neural networks for these tasks individually is much less involving than training the agent as a whole. Furthermore, just like the *Main* class described above in the case of software development, a controller neural network can be trained that learns to implement any given algorithm by loading different modules depending on the state of the system and input from the environment. This was the fundamental motivation behind the work presented in this document.

To that end, the contribution of this thesis is two-fold:

1. Propose a memory augmented neural network architecture that can learn complex sequential tasks, both continuous and episodic, by learning granular components individually and by learning to integrate these components.

2. Show that the proposed architecture successfully learns to evaluate mathematical sequences and various robotic tasks in simulation.

1.3 Related Works

The fundamentals of this thesis are directly based on the Neural Turing Machine (Graves *et al.* (2014)). However, the most closely related work is the Neural Stored-Program Memory (Le *et al.* (2020)), which was published a few weeks before this thesis. The framework in Le *et al.* (2020) also extends NTM and proposes a differentiable implementation of the Harvard Architecture. While the number of stored programs and parameters are learn-able in Le *et al.* (2020), they are fixed in the framework proposed in this work, which makes it easier to understand the underlying semantics of the model and the training less complicated.

As mentioned earlier, the aim of this thesis is to propose a neural network architecture that can learn complex sequential tasks encompassing long sequences. Rae *et al.* (2016), Le *et al.* (2019) and Csordas and Schmidhuber (2019) have proposed methods with similar goals. Moreover, these works also are, in varying magnitudes, based on the Neural Turing Machine and the Differentiable Neural Computer.

Another major aim of this thesis was to train complex robot policies. Parisotto and Salakhutdinov (2017), Pritzel *et al.* (2017), Oh *et al.* (2016), Gupta *et al.* (2017), Beck *et al.* (2020) are some works that have been successful with complex robotic tasks. While these works propose architectures specific to a category of tasks, the model proposed in this thesis is more generic and can be used for any sequence transduction task.

Chapter 2

BACKGROUND KNOWLEDGE

In this chapter, a few concepts are described that are of importance for the understanding of the work presented in this thesis. An assumption has been made that the reader has some familiarity with artificial neural networks (ANNs) and the two main classes of ANNs - feed-forward networks and recurrent networks.

2.1 von Neumann Architecture

The von Neumann Architecture (von Neumann (1993)) is a computer architecture that was introduced by Jon von Neumann in the year 1945. It is a type of a stored-program computer, i.e. it stores program instructions in a memory instead of a control panel. It consists of three main components (Figure 2.1)

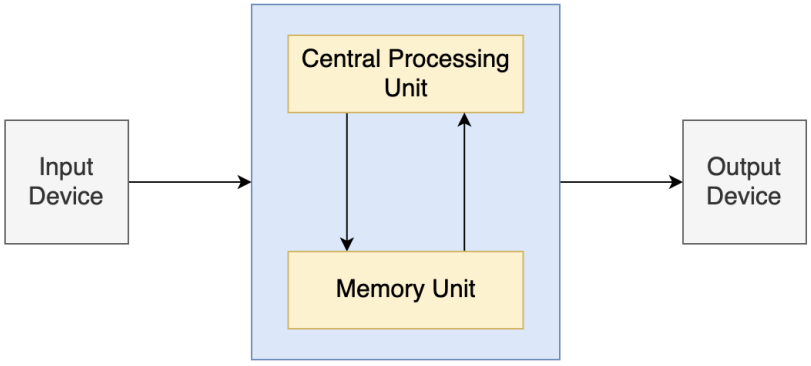


Figure 2.1: von Neumann Architecture

1. A Central Processing Unit(CPU) that has an Arithmetic and Logical Unit(ALU) to compute arithmetic operations and a Control Unit(CU) that has an instruction register to store current instruction and a program counter register

2. A memory that stores data and program instructions

3. Mechanisms to interact with input and output devices

From the architecture diagram of the von Neumann Architecture (Figure 2.1), it is evident that program instructions and data are fetched by the same bus. However, CPUs execute instructions much faster than the speeds at which memories operate. Hence, the CPU is never able to reach its maximum possible threshold. This phenomenon is called the von Neumann bottleneck.

2.2 Harvard Architecture

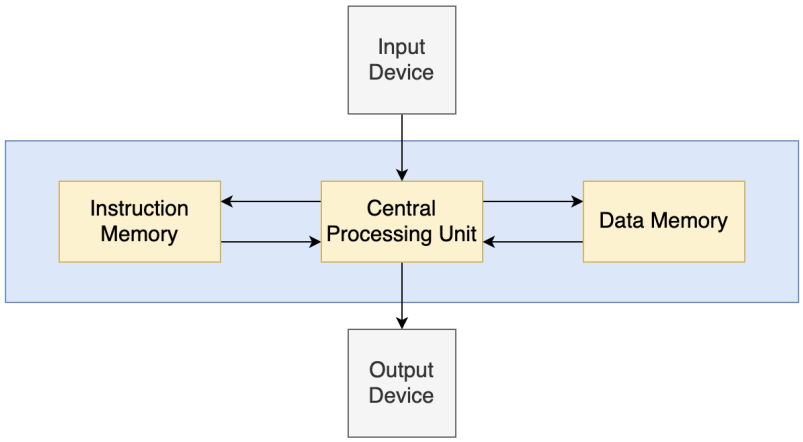


Figure 2.2: Harvard Architecture

The Harvard Architecture (Broesch (2009)) is also a stored-program computer architecture. Unlike the von Neumann Architecture, the Harvard Architecture has different buses for program instructions and data (Figure 2.2). In fact, the instruction memory and data memory share a completely different memory space. By the virtue of this design, the Harvard Architecture does not face a bottleneck in CPU throughput that the von Neumann Architecture faces.

2.3 Turing Machine

A Turing Machine is a mathematical model of computation that defines an abstract machine (Turing (1936)), invented by Alan M. Turing in 1936. It has an infinite array of cells called a memory tape, and a *head* that it uses to interact with the tape. Given a symbol read at a particular cell by the head, the machine can either move the tape left or right, write to the current cell or halt the computation. Despite its simple design, a Turing Machine can simulate the logic of any computer algorithm.

2.4 Neural Turing Machine

Neural Turing Machine (Graves *et al.* (2014)) is Memory Augmented Neural Network (MANN) that has a single read-write memory block and a neural network controller that uses soft attention mechanisms to address the memory, as discussed in Chapter 1. It is an implementation of the Turing Machine which follows the von Neumann design; there is only a single memory to store both data and instructions. In this section, the details of its architecture and the methods through which it interacts with the memory are discussed. Because the Harvard Machine proposed in this thesis is an extension of the principles of the NTM, it is of importance that the reader is familiar with them.

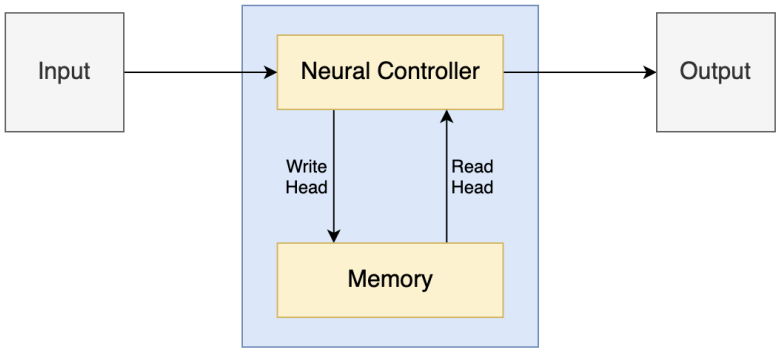


Figure 2.3: Neural Turing Machine - Architecture Overview

2.4.1 Architecture Overview

The NTM has a neural network controller that reads sequential inputs and generates outputs. This neural network can either be a feed-forward network or a recurrent network. However, as the output at time t depends on the memory contents at time $t-1$, the entire architecture is recurrent. At each time step t, the controller emits a read head(s) and a write head(s). These heads act like attention mechanisms over memory locations for reading and writing respectively. Figure 2.3 depicts a high level view of the architecture.

2.4.2 Reading from Memory

Let $\boldsymbol{M}_t$ be the contents of the $N \times M$ memory at time t, where N is the number of memory locations and M is the size of each location. The read head emits a weighting over the N locations of the memory, represented by vector $\boldsymbol{w}_t$ such that:

$$\Sigma_i w_t(i) \;=\; 1, \quad 0 \leq w_t(i) \leq 1$$

Then, the read vector $\boldsymbol{r}_t$ returned by the read head can be defined as follows:

$$\boldsymbol{r}_t \;=\; \Sigma_i w_t(i) \boldsymbol{M}_t(i)$$

2.4.3 Writing to Memory

Like LSTMs, the write operation is broken down into two - an erase operation followed by an add operation. To that end, the write head emits two vectors, an erase vector $\boldsymbol{e}_t$ and an add vector $\boldsymbol{a}_t$, in addition to weightings $\boldsymbol{w}_t$. The write operation can then be decomposed into the following two operations:

$$\tilde{M}_t(i) \;=\; M_{t-1}(i)[\mathbf{1} \;-\; w_t(i)\boldsymbol{e}_t]$$
$$M_t(i) \;=\; \tilde{M}_t(i) \;+\; w_t(i)\boldsymbol{a}_t$$

where **1** is a row vector of 1s and the subtraction and multiplication operations occur point-wise.

2.4.4 Addressing Mechanisms

Memory Networks introduced in Chapter 1, addressed memory locations by calculating similarities between vector representation of the query and the contents of the memory. Such an addressing scheme is called content-based addressing. Although simple, just content-based addressing may not be enough for certain class of problems. Sometimes the state of a variable me be arbitrary but the variable still needs to be stored in the memory. Searching for the most similar answer for a question may be accomplished through content-based addressing, but storing the result of multiplication of two variables cannot be.

The NTM has one other type of addressing mechanism in addition to content-based addressing, location-based addressing, where a memory location can either be selected iteratively or can be randomly jumped to.

For content-addressing, each head produces a key vector $\boldsymbol{k}_t$ of length M. The similarity between this key vector and each location of the memory $M_t(i)$ is calculated by similarity measure $K[.,.]$. A scalar β_t determines the scale of attention to each location.

$$w^c{}_t(i) \;=\; \frac{exp(\beta_t K[\boldsymbol{k}_t, M_t(i)])}{\Sigma_j exp(\beta_t K[\boldsymbol{k}_t, M_t(j)])}$$

In the paper Graves *et al.* (2014), the similarity measure used was Cosine Similarity.

Location based addressing is a combination of three separate operations:

1. **Interpolation**: A scalar g_t, which takes values in the closed interval [0,1], acts as a gate to interpolate weights produced by content-addressing and the weights

from previous time-step.

2. **Shifting**: To focus on a particular memory location, the heads emit a vector $\boldsymbol{s}_t$ that is a normalized weighing over the allowed shits. For example, if the allowed shifts are in the range $[-2, 2]$, the distribution will be over $[-2, -1, 0, 1, 2]$, where the sign indicates the direction in which to shift the attention (up is positive, down if negative) and the absolute value indicates how many memory locations to shift.

3. **Sharpening**: As the shift vector may get dispersed over time, the final weightings are sharpened by performing a softmax-like operation. To that end, the head emits a scalar $\gamma_t \geq 1$ that defines the extent of sharpening.

The above mentioned operations can be formalized by the following equations:

$$\boldsymbol{w}^g{}_t \;=\; g_t \boldsymbol{w}^c{}_t \;+\; (1 - g_t)\boldsymbol{w}_{t-1}$$

$$\tilde{w}_t(i) \;=\; \Sigma_j w^g{}_t(j) s_t(i - j)$$

$$w_t(i) \;=\; \frac{\tilde{w}_t(i)^{\gamma_t}}{\Sigma_j \tilde{w}_t(i)^{\gamma_t}}$$

Chapter 3

METHODOLOGY

Let us revisit the example from Chapter 1 of a robot in a park collecting trash and depositing it in trash-cans, in more detail. Say the robot is an arm mounted on a mobile platform and is equipped with a depth camera like Kinect, a GPS sensor and a compass. Furthermore, it also knows the location(GPS coordinates) of all the trash-cans in the park. To train a policy for this robot, end-to-end using reinforcement learning, is intricate. However, training the robot to perform the following tasks may not be as involving:

1. For a given image of an object, classify it as trash or not trash.
2. Pick an object after seeing depth images
3. Plan a path from current location to nearest trash-can
4. Avoid obstacles when navigating on a path

Therefore, neural networks, called Task Networks from hereon, can be trained individually to learn the above mentioned policies. Each task network generates a distribution over actions to select from, based on the input signals it receives from the environment.

Let an episode be defined as the agent picking up one trash object and placing it in a trash-can. Each episode can then be represented as a sequence of states and actions, $(\boldsymbol{\chi}_t, \boldsymbol{a}_t)$, where $\boldsymbol{\chi}_t$ is the state and $\boldsymbol{a}_t$ is the optimal action to be taken at time t. A recurrent neural network, the Controller Network, can learn to transduce the sequences of inputs into sequences of actions. The controller receives inputs from

the environment and reads the current state of its memory. It then decides which task network should be designated to generate the next action. In order to make the model trainable by back-propagation, instead of selecting one task network, the controller generates a distribution over task networks. Optionally, depending on the nature of the task, the controller can chose to write the output at each time-step in the memory. The supervision is performed on the output generated at every time-step, i.e. the model is trained end-to-end.

More formally, each input $\boldsymbol{\chi}_t$ presented to the controller network at time t, the controller learns to generate the probability distribution $P(\boldsymbol{a}_t|\boldsymbol{\chi}_t, \theta_t^C, \theta^{T_1}, \theta^{T_2}, ..., \theta^{T_k})$, where θ_t^C are the parameters of the controller network and $\theta^{T_1}...\theta^{T_k}$ are the parameters of the task network for k different tasks.

The model has Six main components:

1. **Controller**: The controller, N^c, is a neural network that reads sequential input symbols and generates output symbols. Analogous to the Turing Machine, the input and output symbols can be assumed to be coming from an infinitely long tape. The controller also emits weightings to address the data and the program memory. Either a feed-forward or a recurrent neural network can be used as the controller. Since the state of the program memory is maintained over time, the entire architecture is recurrent irrespective of the choice of the controller network.

2. **Task Network**: A Task Network, N^t, is a neural network that has been trained to perform a sub-part of the entire task. For example, a network that has been trained to add two numbers can be used to evaluate mathematical expressions. There can be a single task network (train the same architecture for all the sub-tasks) or multiple task networks.

3. **Program Memory**: It is a read-only memory that stores learned parameters for different sub-tasks. At each time-step, the appropriate parameters are chosen from this memory and loaded into the task network(s).

4. **Data Memory**: It is a read-write memory that the controller uses to store information that may be of importance in the future. For example, while evaluating mathematical expressions, it can store intermediate results that will be required for future calculations.

5. **Program Memory Head**: It is a continuous read head that addresses the Program Memory.

6. **Data Memory Heads**: These are continuous read and write heads that address the Data Memory.

In this chapter, two different architectures are discussed. The need for two architecture stems from the nature of tasks. In some problems, the outputs generated at intermediate steps may be required to calculate outputs in the future. Therefore, the controller must decide if the output at any given time step should be written in the data-memory. Also, the inputs in the task networks should be a function of the contents of the data memory. Alternatively, some problems do not necessitate the storage of intermediate outputs. In such cases, the only inputs to the task networks are the inputs that the controller sees at every time-step. To that end, the following two sub-sections describe these to architectures in detail.

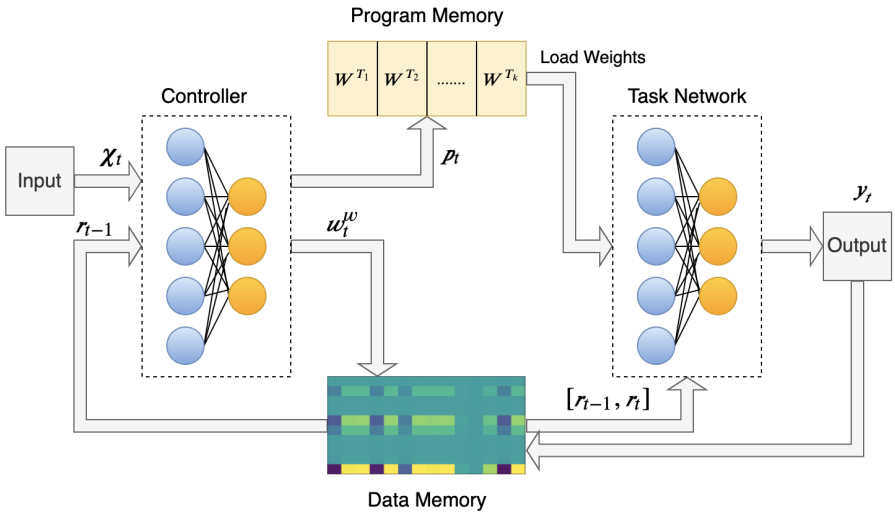


Figure 3.1: Architecture-1 of the Proposed Differentiable Harvard Machine. The Output from the Task Network Can Be Written to the Data Memory.

3.1 Architecture - 1

3.1.1 Controller

The controller, N^c, can be a feed-forward or a recurrent neural network. In all the experiments listed through-out this thesis, a feed-forward network was used. At every time-step t, the controller receives input $\boldsymbol{\chi}_t$, and outputs parameters $\boldsymbol{\xi}_t$ and $\boldsymbol{\zeta}_t$ that parameterize a read-write memory $\boldsymbol{M}_t$ and a pointer vector $\boldsymbol{p}_t$ to a read-only memory $\boldsymbol{M}^r$, respectively. The read-only memory $\boldsymbol{M}^r$ has K locations that contain learned parameters for the K tasks, N^{t_1}... N^{t_K}. The output of the penultimate layer of the controller, O^{L-1} is used to generate parameters $\boldsymbol{\xi}_t$ and $\boldsymbol{\zeta}_t$ for the data and program memories respectively. The architecture is represented in Figure 3.1.

$$
\begin{aligned}
\boldsymbol{\xi}_t &= W_t^{\xi}[O_t^{L-1}] \\
\boldsymbol{\zeta}_t &= W_t^{\zeta}[O_t^{L-1}] \\
(\boldsymbol{\xi}_t, \boldsymbol{\zeta}_t) &= N^c(\boldsymbol{\chi}_t; \theta_t^c)
\end{aligned}
$$

3.1.2 Addressing Mechanisms

There are two memories to be addressed here, $\boldsymbol{M}_t$ and $\boldsymbol{M}^r$. The weights for $\boldsymbol{M}_t$, $\boldsymbol{w}_t^r$ (read weights) and $\boldsymbol{w}_t^w$ (write weights), are computed from $\boldsymbol{\xi}_t$ similar to how Neural Turing Machines do it. The vector $\boldsymbol{\zeta}_t$ has two components:

$$\boldsymbol{\zeta}_t \;=\; [\mu_t; \boldsymbol{p}_t]$$

$\boldsymbol{p}_t$ are the weightings that address the pointers to $\boldsymbol{M}^r$ and μ_t is a scalar that will be used to interpolate the add vector with output. μ is of significance only if the architecture allows writing the predicted output into data memory. More about μ is discussed in the next sub-section.

3.1.3 Reading from and Writing to the Data Memory

Let $\boldsymbol{M}_t$ be the $N \times W$ data matrix with N locations of size W. At time t, the controller emits read weights $\boldsymbol{w}_t^r$ which define a weighing over the N locations of the memory. Since all the weightings are normalized, therefore:

$$\Sigma_i w_t^r(i) \;=\; 1, \quad 0 \leq w_t^r(i) \leq 1$$

Then, the read vector at time t,

$$\boldsymbol{r}_t \;=\; \Sigma_i w_t^r(i).M_t(i)$$

For writing, the controller also emits normalized write weights $\boldsymbol{w}_t^w$. Apart from the weights, it also emits two vectors - an erase vector $\boldsymbol{e_t}$ and an add vector $\boldsymbol{a_t}$. The erase vector determines memory from which locations can be removed and the add vector determines what is to be written to the memory. Also, the scalar ρ_t controls how much output at time t affects the value of a_t. Hence, at some time-steps, the controller can choose to write the output of previous iteration to the memory.

$$\boldsymbol{a_t} = \boldsymbol{a_t} * \mu_t + Out_t * (1 - \mu_t)$$
$$M_t{}^e(i) = M_{t-1}(i)[\mathbf{1} - w_t^w(i).\boldsymbol{e_t}]$$
$$M_t(i) = M_t{}^e(i) + w_t^w(i).\boldsymbol{a_t}$$

3.1.4 Reading from Program Memory

The read vector of the program memory encodes the learned parameters for the task network. This vector is obtained in the same way as for the data memory - a linear combination of attention weightings and contents of the memory location.

$$\boldsymbol{\theta}_t^T = \Sigma_i p(i).M^r(i)$$

Here $\boldsymbol{\theta}_t^T$ are the parameters that will be loaded in the task network.

3.2 Architecture - 2

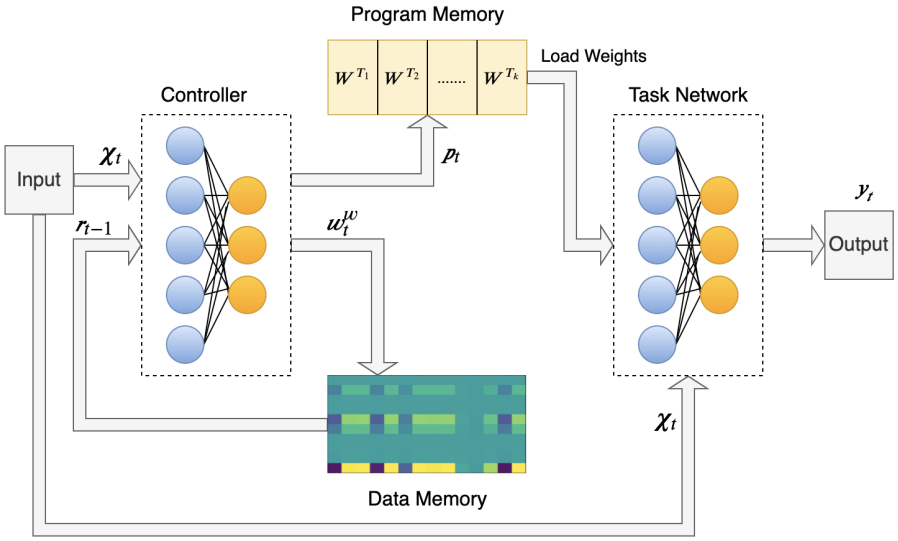


Figure 3.2: Architecture-2 of the Proposed Differentiable Harvard Machine

As both the architectures have the same components and underlying principles, only the differences between them are highlighted in this section. An illustration of the architecture can be seen in Figure 3.2.

3.2.1 Controller

Just like the first architecture, the second to last layer of the controller emits two vectors that parameterize the data and the program memory. However, because the task network is not allowed to write to the data memory, the scalar μ_t is not generated. Therefore, the following equations now define the controller:

$$\begin{aligned} \boldsymbol{\xi}_t &= W_t^{\xi}[O_t^{L-1}] \\ \boldsymbol{p}_t &= W_t^{p}[O_t^{L-1}] \\ (\boldsymbol{\xi}_t, \boldsymbol{p}_t) &= N^c(\boldsymbol{\chi}_t; \theta_t^c) \end{aligned}$$

3.2.2 Reading from and Writing to the Data Memory

The reading operation is the same in both the architectures. However, because there is no interpolation from the output, the write operation now is exactly the same as in the Neural Turing Machine - an erase operation followed by an add operation. The add vector is directly generated by the controller and is a part of $\boldsymbol{\xi}_t$

Chapter 4

EXPERIMENTS, RESULTS AND ANALYSIS

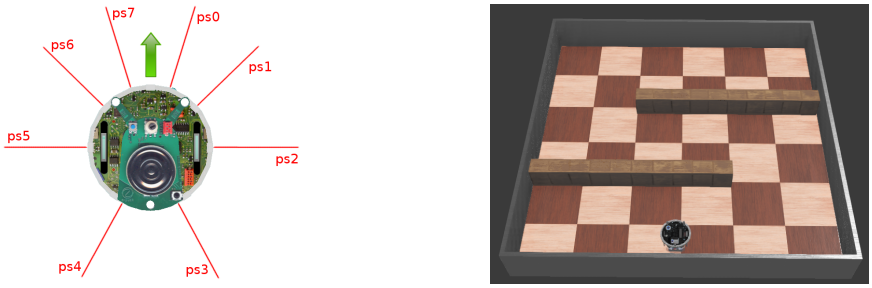
In this chapter, the details of different experiments performed and results obtained on them are discussed. Each section represents a class of experiments and is divided into three sub-sections - the experimental setup, results obtained and the analysis of those results. Experiments were performed on both the architectures described in the previous chapter. A comparison of performance between the Harvard machine architecture presented in this thesis, Neural Turing Machine and Long Short Term Memory(LSTM) Networks is provided in terms of learning rate and accuracy. The simulator used for the purpose of robotic experimentation is Webots.

4.1 Obstacle Avoidance and Wall Following

4.1.1 Experimental Setup

E-puck is a miniature mobile robot originally developed at EPFL for teaching purposes. It has two differential wheels and eight infra-red distance sensors. The robot's top view can be seen in Figure 4.1.

The robot is put into an arena created in Webots simulator (Figure 4.1(b)). The



(a) E-Puck Model (b) WeBots Arena

Figure 4.1: An Overview of the E-puck Robot and the Arena Created in Webots.

arena has a floor surrounded by four walls and two wooden obstacles. With the values of the infra-red distance sensors as input, the following three distinct networks are trained.

1. **Obstacle Avoidance**: Keep moving straight and keep avoiding walls/wooden blocks if encountered.

2. **Wall Following - Clockwise**: Reach the nearest wall and follow it in a clockwise direction.

3. **Wall Following - Anti-clockwise**: Reach the nearest wall and follow it in an anti-clockwise direction.

Combining the above trained policies, Harvard Machine, Neural Turing Machine and LSTM are trained on the following tasks:

1. **Task 1**: Keep moving straight until a wall/wooden box is encountered. Follow the wall in a given direction once encountered.

2. **Task 2**: Keep moving straight and avoid walls/wooden boxes if encountered. Keep a count of the number of times a box/wall is avoided. As the count reaches five, follow the next wall encountered in a given direction.

3. **Task 3**: Keep moving straight and avoid walls/wooden boxes if encountered. Keep a count of the number of times a box/wall is avoided. As the count reaches two, follow the next wall encountered in a given direction. While following the wall, if an end of either of the wooden boxes is reached, revert to obstacle avoidance behavior again.

The direction in which the walls are to be followed is given to the robot at $t = 1$ as a signal. The robot has to remember that signal until it switches to wall following behavior.

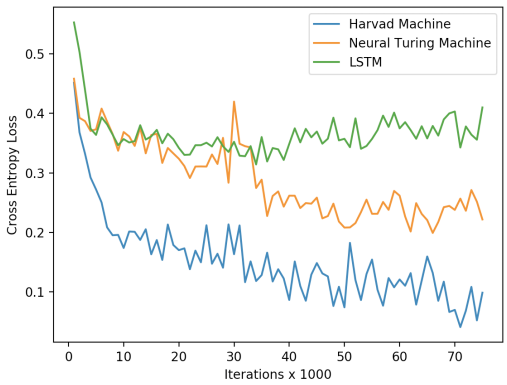
Model	Accuracy	Accuracy: Critical Points
Harvard Machine	**98.14%**	**89.98%**
Neural Turing Machine	89.08%	63.71%
LSTM	81.43%	36.75%

Table 4.1: Accuracy of Harvard Machine, Neural Turing Machine and LSTM on the First E-puck Task.

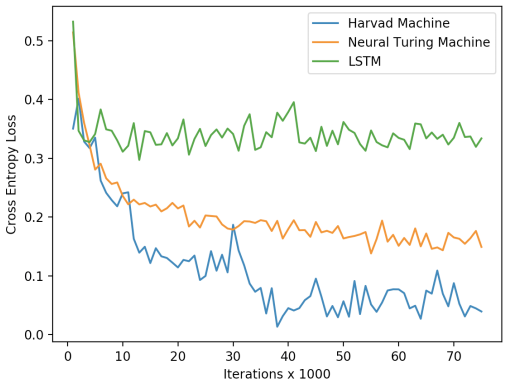
4.1.2 Results

It can be clearly observed that the tasks have an increasing order of complexity. The average sequence lengths for the above tasks were approximately *342*, *687* and *834* respectively. The consequence of such long sequences can be seen in the Training Loss vs Iterations graphs in Figure 4.2. While LSTM converges well before the optimal behavior, the Harvard Machine learns faster and converges to a better solution than the NTM.

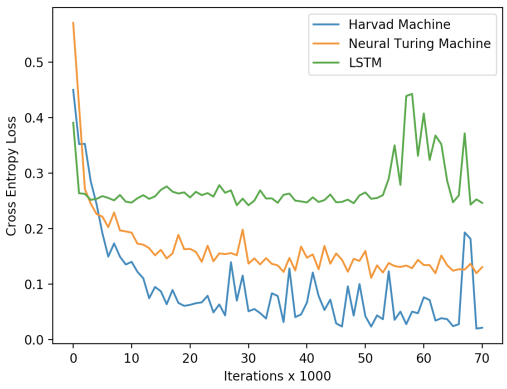
Tables 4.1, 4.2 and 4.3 show the accuracy attained after training each model on the above mentioned tasks. The accuracy, however, is not the best metric to demonstrate the definiteness of these models. Take for example a task in which the robot has to keep following a linear path for 99 steps and then turn left. If a network predicts the first 99 actions correctly (going straight) but ends up predicting the last action as going right, the robot may end up being in a completely different place. The accuracy of such a model would still be 99%. Thus, the accuracy of a model at such critical points is important.



(a) Task-1

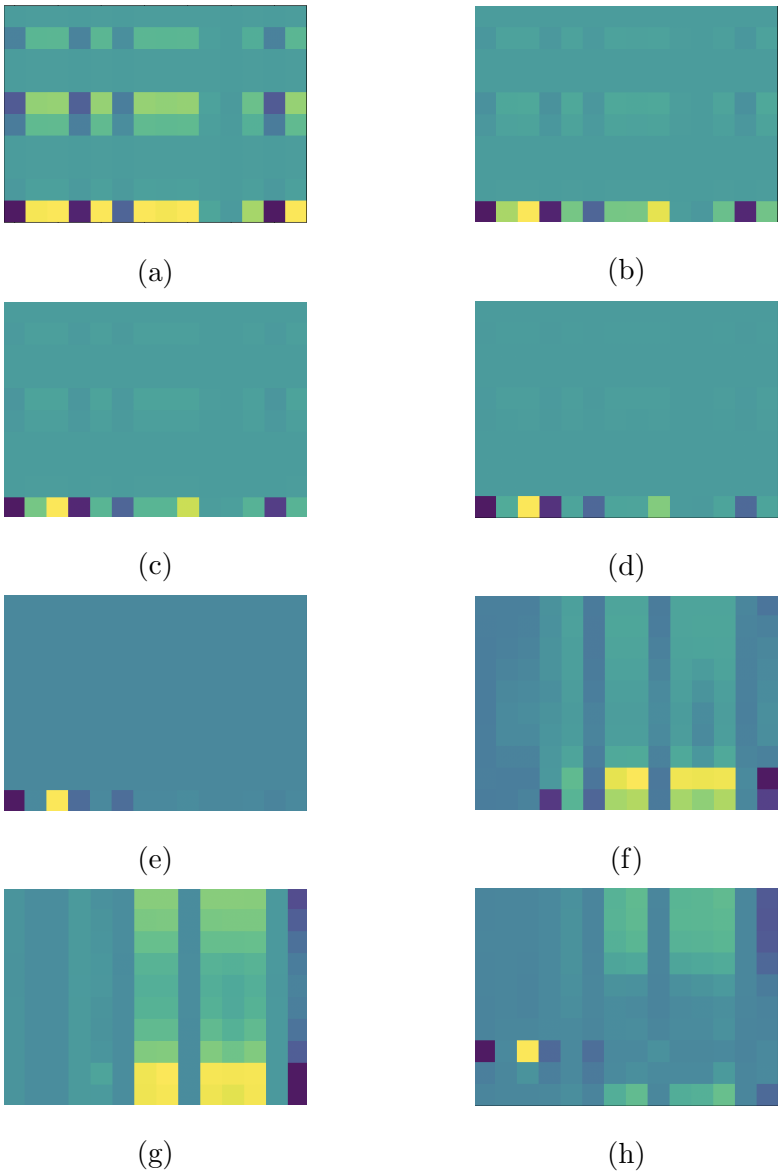


(b) Task-2



(c) Task-3

Figure 4.2: Training Error Vs Iteration Plots Obtained in the Three Tasks for Harvard Machine, Neural Turing Machine and LSTM.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 4.3: Changes in the Data Memory as the Harvard Machine Parses over a Sequence, for the Third Task with E-puck Robot.

Model	Accuracy	Accuracy: Critical Points
Harvard Machine	**97.86%**	**86.02%**
Neural Turing Machine	92.47%	57.72%
LSTM	84.73%	12.98%

Table 4.2: Accuracy of Harvard Machine, Neural Turing Machine and LSTM on the Second E-puck Task.

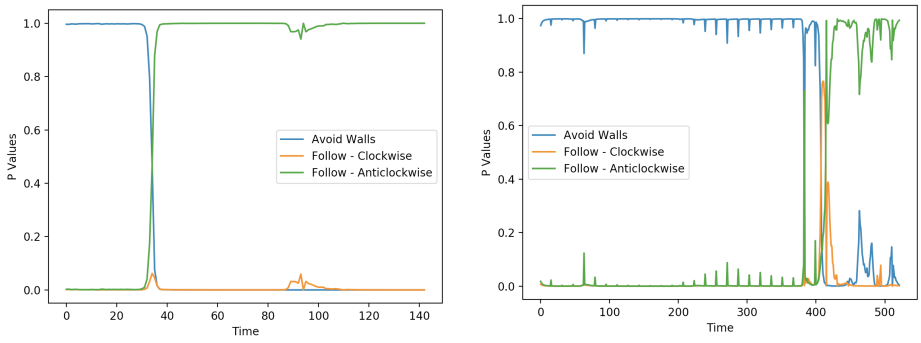
Model	Accuracy	Accuracy: Critical Points
Harvard Machine	**98.65%**	**86.9%**
Neural Turing Machine	92.63%	60.19%
LSTM	88.79%	8.5%

Table 4.3: Accuracy of Harvard Machine, Neural Turing Machine and LSTM on the Third E-puck Task.

4.1.3 Analysis of Results

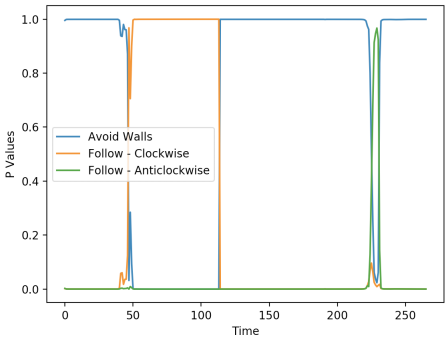
In Figure 4.3, the state of the data memory at different time-steps is presented for *Task 3*. A similar pattern was seen for all the tasks - the controller writes something in the memory which becomes more prominent over time, and when the robot switches behaviour, the contents of the memory change drastically.

To analyze the performance of the model, the most important factor is the confidence with which it selects weights from the Program Memory. If the right network is chosen at a given time-step, the output is guaranteed to be correct because the task network reads inputs from the input tape rather than the data memory. Figure 4.4 depicts how the values of $p(i)$ change over time, for each of the 3 tasks. It can be concluded that at any given time, the model has high confidence in choosing a task



(a) Task-1

(b) Task-2



(c) Task-3

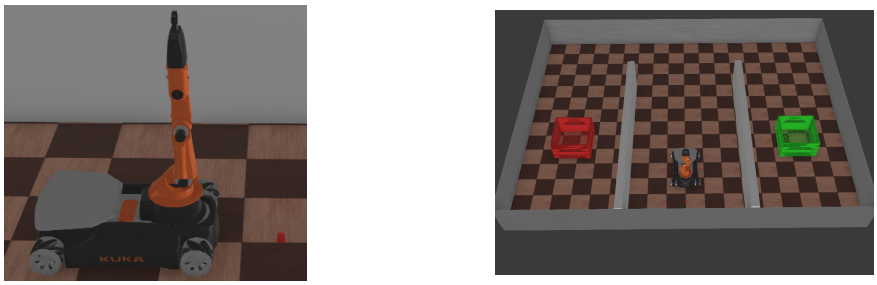
Figure 4.4: Variation in P Values over Time for the E-puck Tasks.

network, as the values of $p(i)$ do not overlap.

4.2 Pick and Sort Boxes

4.2.1 Experimental Setup

In this task, A mobile arm equipped with a camera is presented with an object. The object can either be a red or a green colored box. There are two bins (colored red and green), which are placed at different locations in an arena. The robot learns to pick the object, plans to navigate to the correct bin and then drops the object in the bin. An important point to note is that once the object is picked up, it is no longer in the robot's field of view. Therefore, it has to remember the object it had picked until it places it in a bin. This task is similar to the trash collection task discussed



(a) Kuka YouBot (b) WeBots Arena

Figure 4.5: An Overview of the Youbot Robot and the Arena Created in Webots for Sort Items Experiment.

in Chapter 1.

The mobile arm used for this task is Kuka's YouBot. It is an arm with two fingers and five degrees of freedom. The arm is mounted on a unidirectional platform. The simulator used to create the arena is Webots. Figure 4.5 depicts the robot and the arena in more detail. As it can be seen from the figure, the bins are not visible to the robot while picking the object.

For this experiment, the following four task networks were trained:

1. Given images from an RGB camera, strafe towards the object and execute a grip.

2. Given GPS coordinates of the robot, navigate to the red bin and drop the object.

3. Given GPS coordinates of the robot, navigate to the green bin and drop the object.

While training on sequences, the Harvard Machine is presented with images from the camera and GPS coordinates of the robot. At each time-step, it generates a distribution over the following actions - move forward, move backward, strafe left, strafe right, pick the object, and drop the object.

Architecture-2 of the Harvard Machine, as described in the previous chapter, was used as the model. Furthermore, the network to navigate to bins and to pick a box have different architectures. Therefore, it is not possible to load weights like it was for the E-puck tasks. One way to go about having multiple task networks is to look at the problems from another angle - instead of picking the weights for a single task network, the controller can choose one output from multiple task networks. This can be seen as an equivalent of choosing one network to generate an output at a particular time-step. Therefore, the following equation:

$$\boldsymbol{\theta}_t^T \;=\; \Sigma_i p(i).M^r(i)$$

now becomes

$$\boldsymbol{y}_t \;=\; \Sigma_i p(i).f_T^i(\boldsymbol{\chi}_t; \boldsymbol{\theta}^{T_i})$$

Where $\boldsymbol{y}_t$ is the output generated at time t, $f_T^i()$ is the i^{th} task network and $\boldsymbol{\theta}^{T_i}$ are its learned parameters stored in the program memory. If the task networks have different output(action) spaces then they can be stacked together in a *Softmax* layer, instead of a linear combination. That is:

$$\boldsymbol{y}_t \;=\; Softmax([p(0).f_T^0(\boldsymbol{\chi}_t; \boldsymbol{\theta}^{T_0}),\; p(1).f_T^1(\boldsymbol{\chi}_t; \boldsymbol{\theta}^{T_1}),\; \ldots\; , p(i).f_T^i(\boldsymbol{\chi}_t; \boldsymbol{\theta}^{T_i})])$$

4.2.2 Results

A comparison of learning curves of the Harvard Machine and the Neural Turing Machine are shown in Figure 4.6. The figure indicates that the Harvard Machine converges much faster than NTM. A comparison of their accuracy can be found in Table 4.4.

Model	Accuracy	Accuracy: Critical Points
Harvard Machine	**96.9%**	**96.36%**
Neural Turing Machine	78.04%	70.76%

Table 4.4: Accuracy of Harvard Machine and Neural Turing Machine on the Sort Items Experiment.

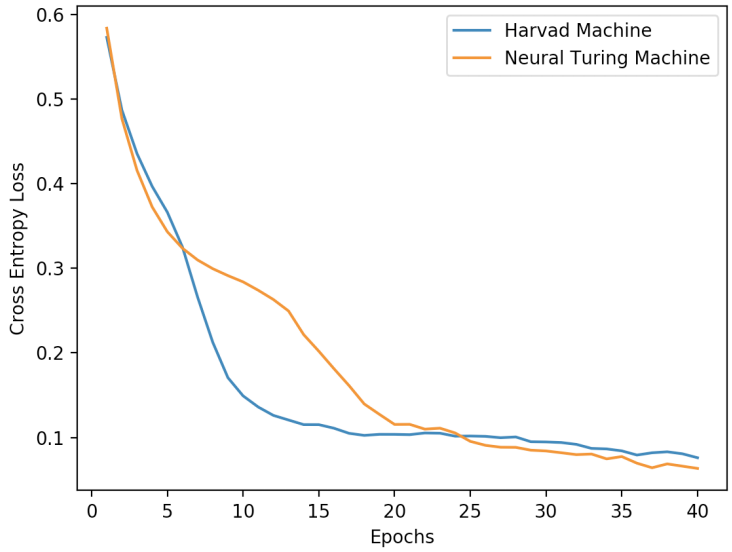


Figure 4.6: Training Error Vs Epochs - Comparison Between Harvard Machine and Neural Turing Machine.

4.2.3 Analysis of Results

One major observation that can be made from the results is the relation between training loss and accuracy for Neural Turing Machine. Although NTM converges to a better optima, its accuracy is much less than that of the Harvard Machine. This happens because the loss is the average of losses for picking up a green box and a red box. The Neural Turing Machine almost always over-fits to one of the classes. Therefore, its loss for the other class increases while the average is not affected by a large quantity. Furthermore, its confidence while taking actions for the other class is

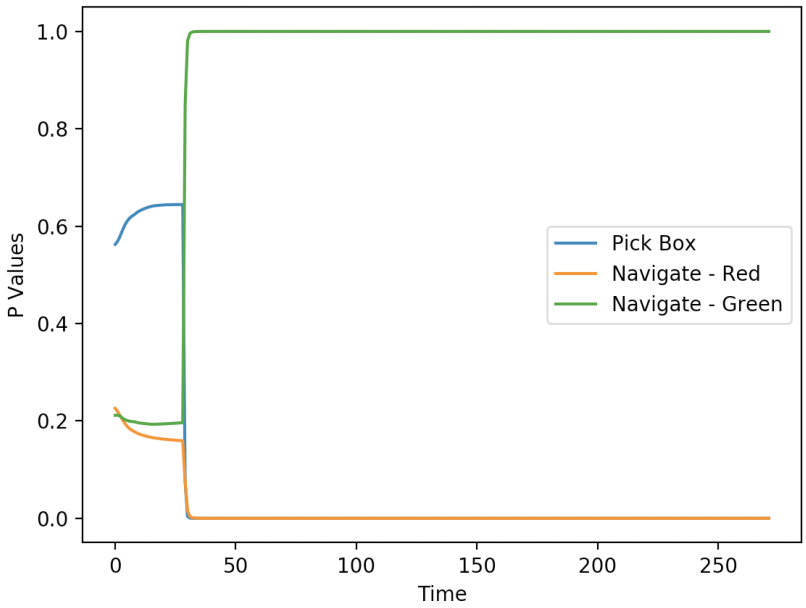


Figure 4.7: Variation in P Values over Time for the Sort Boxes Task.

also low. That is the reason why the cross-entropy loss remains less.

To analyze the confidence of the Harvard Machine, once again the variation in P values with time is depicted in Figure 4.7. Initially, the pick box task, illustrated by the blue curve, has the highest value. This suddenly drops when the box is picked up. Following that, the value for the network that navigates the robot to the green bin rises close to 1 (green curve in the figure.)

4.3 Mathematical Expressions

4.3.1 Experimental Setup

The task network, N^t, has been trained to learn two tasks: add and multiply two numbers in the range $[1, 9]$. The controller receives a sequence of operands and symbols as an input that is a combination of these two tasks. A sample input is given below:

$$(op_1 \ + \ op_2) \ * \ (op_3 \ + \ op_4)\$$$

Where op_1, op_2, op_3 and op_4 are four operands chosen randomly in $[1, 9]$. The '$' symbol acts as a signal to the network that signifies the end of an input sequence. Also, each input symbol is converted into a one-hot vector of size 14; 9 places for the digits, 2 for opening and closing parenthesis, 2 for addition and multiplication operations and one for the delimiter '$'.

As storing the values of intermediate add operations is required to evaluate such an operation, the first architecture of the Harvard Machine proposed in this thesis is used to learn this task.

4.3.2 Results

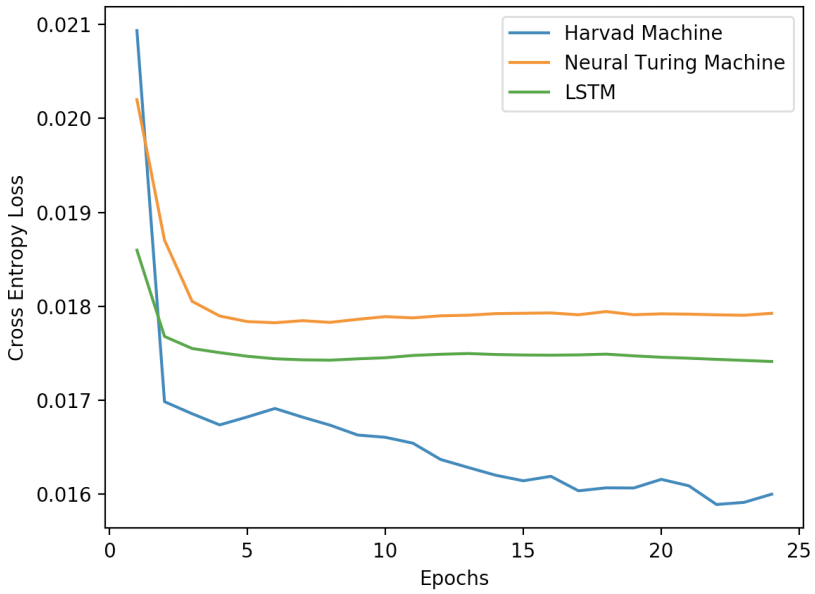


Figure 4.8: Training Loss Vs Epochs for Mathematical Expression Evaluation Task.

The results of these experiments are reported in Table 4.5. The error term represents average divergence of a model from the correct answer. For example, if the correct answer was 127 and the model predicted it as 133, the error, which is the absolute difference, is 6. This is divided by the maximum possible error (321 in the scenario with four operands) to get the percentage of average error.

Model	Average Error (%)
Harvard Machine	7.28
Neural Turing Machine	14.28
LSTM	16.9

Table 4.5: Comparison of Harvard Machine, Neural Turing Machine and Lstm on Math Expressions Evaluation Task. The Error Term Represents Average Divergence of a Model from the Correct Answer - Absolute Difference Between Actual and Predicted Output, as a Percent of the Maximum Possible Error.

Symbol	P-Value
First ')'	[1.000000e+00 9.085198e-14]
Second ')'	[9.9975413e-01 2.4582000e-04]
'$'	[9.165004e-09 1.000000e+00]

Table 4.6: Change of P Vector over Time-steps.

The learning curves for the three models are shown in Figure 4.8. Harvard Machine converges to a significantly better minimum than NTM and LSTM.

4.3.3 Analysis of Results

As it can be seen from the table, the Harvard Machine performs significantly better while evaluating the expression. The patterns observed in its data-memory is shown in Figure 4.9. Initially, all the cells of the memory are assigned with a value of 10^{-6}. As it can be seen from the figure, the controller writes new operands to the memory and reads them while performing computations at a later time-step. This indicates that the Harvard Machine understands the semantics of a task better than NTM and LSTM.

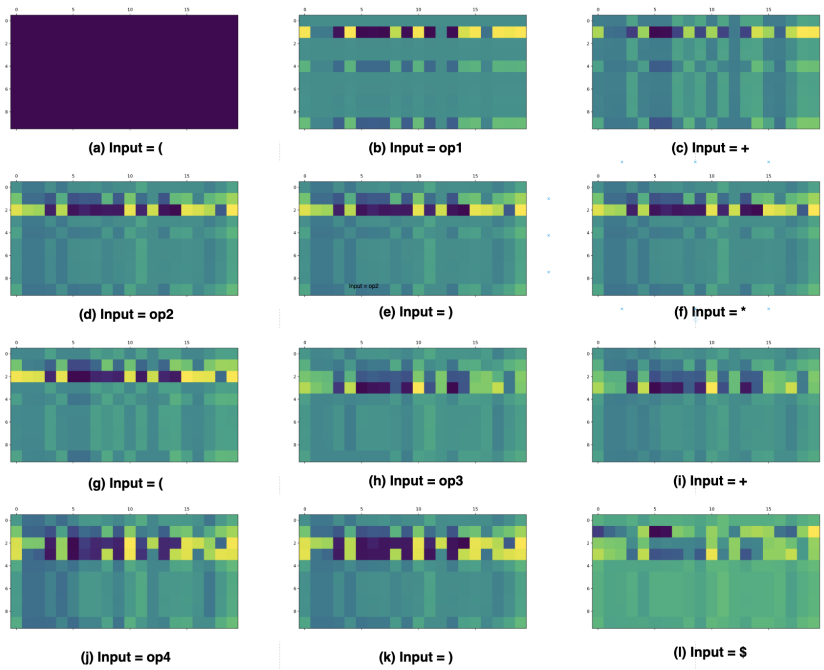


Figure 4.9: State of the Data Memory after Each Element of Input Sequence

In Figure 4.9, the state of the data memory after each input symbol is presented. Whenever a new operand is encountered, the controller writes its representation in the memory, which is later retrieved while performing computations. However, just the trace of the memory is not enough to judge if the model is learning useful representations.

Table 4.6 has a list of P values generated by the controller during different stages of the input sequence. Only three are shown here because the values at those instances are more significant in evaluating the overall expression. P values are a vector in which the first elements corresponds to the parameters for add operation and the second element for multiplication operation. When the addition sub-expressions end, the controller confidently picks the parameters for the add network. Conversely, when the $ symbol is encountered, the network loads parameters for the multiplication network. The same pattern can be seen in the values of μ. This is illustrated in Figure 4.10. The controller chooses to write outputs of the intermediate results to its

data memory.

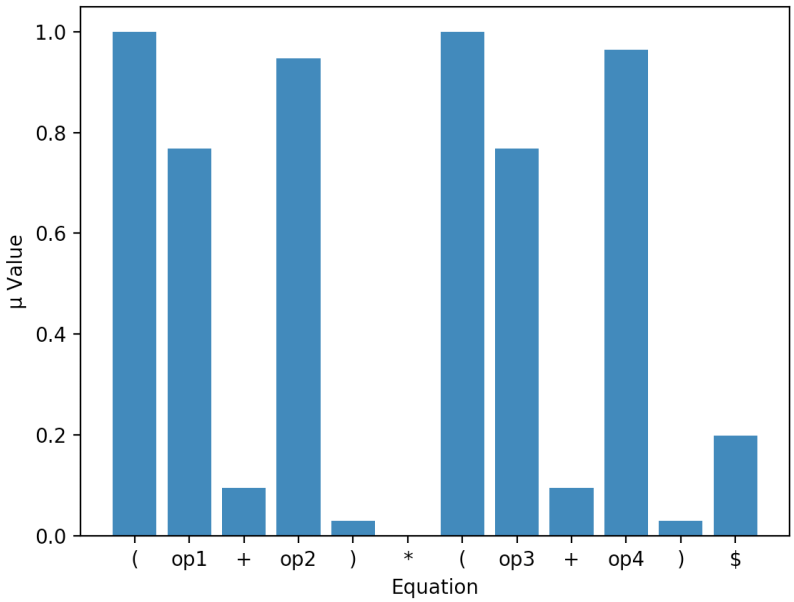


Figure 4.10: Change of the Interpolation Scalar over Time for the Mathematical Expression Evaluation Task.

What do not transpire as per expectations are the values of the operands passed to the task network. Recollect that in the Architecture-1 described in Chapter 3, the input to the task network is a function of the Data Memory. In this task, the input is created by concatenating the read heads and current and previous time-step. To ensure that correct outputs are generated at each time-step, two conditions should be met:

1. The controller should load correct weights in the task network.

2. The controller should choose inputs such that the task network generates desired output.

Figure 4.11, a graph is shown that illustrates the inputs generated by the controller and the desired inputs, for the task network at the last time-step. As it can be seen, there is a significant difference between them. It can be concluded that the controller

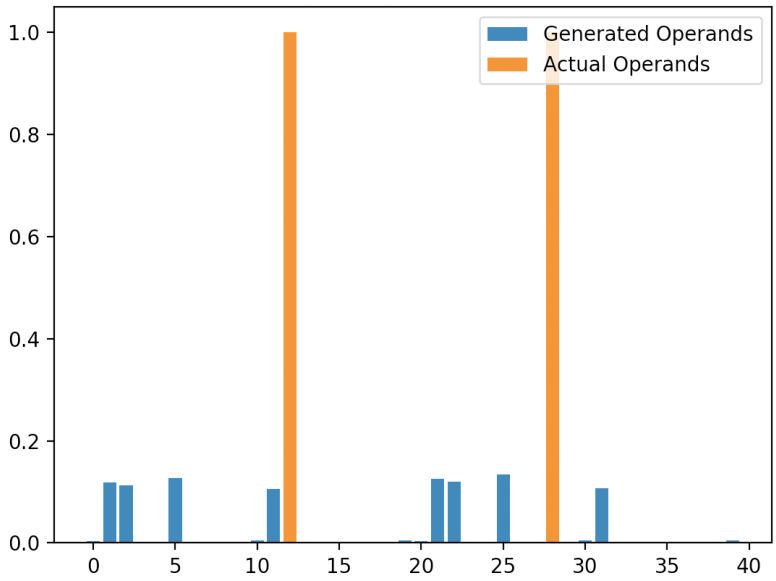


Figure 4.11: A Comparison Between Actual Operands That Were Supposed to Be Passed to the Task Network Vs the Operands Generated by the Controller.

does not actually learn to write and load intermediate results. Instead it learns to load such operands that it thinks will produce the most appropriate output. This analysis is of extreme importance because it substantiates that even though it is easier to visualize the representations stored by the Harvard Machine over recurrent neural networks like LSTMs, it still does not learn the precise semantics of this task.

Chapter 5

CONCLUSION AND FUTURE RESEARCH

To solve complex sequential learning problems, a hierarchical learning method was proposed in this thesis. A big, convoluted task is divided into simpler sub-problems and separate neural networks are trained to learn these sub-problems. These neural networks are called Task Networks in this thesis. A neural network controller, augmented with an external memory, then goes through the symbols of the sequence one at a time and learns to load learned parameters for the appropriate task network at each time-step. This proposed architecture is called the Differentiable Harvard Machine as it is inspired from the Harvard Machine design introduced in Chapter 2. Two separate architecture of the Differentiable Harvard Machine are proposed to account for different characteristics of tasks.

The directions of future research for both these architectures are listed in this chapter.

5.1 Architecture 1

The first architecture allows the output generated by the task network to be written to the data memory. Hence, it can be used for tasks that require intermediate outputs in future time-steps, like the task of evaluating mathematical expressions. However, the model failed to understand the underlying semantics of the task and over-fitted to the form of the given equation. Two research paths for this task can be:

1. **Training with variable length sequences:** One reason for the controller over-fitting may be because the equation has the same format. Hence, instead

of looking at what the symbols are actually, it can memorize the format and generate specific outputs depending on the inputs. This will not happen if the format of the equation is variable.

2. **Loss function formulation**: The loss function can be formulated in such a way that it forces the controller to produce desired intermediate outputs.

5.2 Architecture 2

The second architecture performs well in robotic experiments, remembering information over long sequences. Some further improvements that can be made in the architecture are as follows:

1. **Training with the reinforce algorithm**: Supervised learning is not always feasible for robotic tasks because of a lack of training data. Hence, training the model with reinforcement learning is preferable. The model then represents the Quality Function - $Q(s, a)$.

2. **Transferring to a real-world robot**: All the experiments performed in this thesis were on a simulator. However, it is important that the learned policy can be transferred to a real robot.

3. **Write-able program memory**: If the program memory can be made write-able, the model can learn actively - while it is exploring an environment. It will be interesting to see how the model represents sub-problems for a given task.

REFERENCES

Beck, J., K. Ciosek, S. Devlin, S. Tschiatschek, C. Zhang and K. Hofmann, "Amrl: Aggregated memory for reinforcement learning", in "International Conference on Learning Representations", (2020), URL https://openreview.net/forum?id=Bkl7bREtDr.

Bourlard, H. A. and N. Morgan, *Connectionist Speech Recognition: A Hybrid Approach* (Kluwer Academic Publishers, USA, 1993).

Broesch, J. D., *Digital Signal Processing* (Newnes, Burlington, 2009), URL URL http://www.sciencedirect.com/science/article/pii/B9780750689762000080.

Csordas, R. and J. Schmidhuber, "Improving differentiable neural computers through memory masking, de-allocation, and link distribution sharpness control", in "International Conference on Learning Representations", (2019), URL https://openreview.net/forum?id=HyGEM3C9KQ.

Elman, J. L., "Finding structure in time", in "Cognitive science", vol. 70, p. 190–198 (1990), URL https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402$_1$.

Giles, C. L., C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun and L. Y. C., "Learning and extracting finite state automata with second-order recurrent neural networks", in "Neural Computation", vol. 4 (1992a).

Giles, C. L., C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen and L. Y. C., "Extracting and learning unknown grammar with recurrent neural networks", in "Advances in Neural Information Systems", No. 4 (1992b).

Graves, A., G. Wayne and I. Danihelka, "Neural turing machines", CoRR **abs/1410.5401**, URL http://arxiv.org/abs/1410.5401 (2014).

Graves, A., G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu and D. Hassabis, "Hybrid computing using a neural network with dynamic external memory", Nature **538**, 7626, 471–476, URL http://dx.doi.org/10.1038/nature20101 (2016).

Gupta, S., J. Davidson, S. Levine, R. Sukthankar and J. Malik, "Cognitive mapping and planning for visual navigation", in "2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)", pp. 7272–7281 (2017).

Joulin, A. and T. Mikolov, "Inferring algorithmic patterns with stack-augmented recurrent nets", in "Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1", NIPS'15, p. 190–198 (MIT Press, Cambridge, MA, USA, 2015).

Kanerva, P., *Sparse Distributed Memory* (MIT Press, Cambridge, MA, USA, 1988).

Khan, A., A. Sohail, U. Zahoora and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks", CoRR **abs/1901.06032**, URL http://arxiv.org/abs/1901.06032 (2019).

Le, H., T. Tran, T. Nguyen and S. Venkatesh, "Variational memory encoder-decoder", in "Advances in Neural Information Processing Systems 31", edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi and R. Garnett, pp. 1508–1518 (Curran Associates, Inc., 2018a), URL http://papers.nips.cc/paper/7424-variational-memory-encoder-decoder.pdf.

Le, H., T. Tran and S. Venkatesh, "Learning to transduce with unbounded memory", in "Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining, KDD '18", p. 1637–1645 (ACM, 2018b), URL http://doi.acm.org/10.1145/3219819.3219981.

Le, H., T. Tran and S. Venkatesh, "Learning to remember more with less memorization", in "In International Conference on Learning Representations", (2019), URL https://openreview.net/forum?id=r1xlvi0qYm.

Le, H., T. Tran and S. Venkatesh, "Neural stored-program memory", in "International Conference on Learning Representations", (2020), URL https://openreview.net/forum?id=rkxxA24FDr.

Ma, Y. and J. C. Principe, "A taxonomy for neural memory networks", IEEE transactions on neural networks and learning systems (2019).

Mozer, M. C. and S. Das, "A connectionist symbol manipulator that discovers the structure of context-free languages", in "Conference on Neural Information Processing Systems (NIPS)", (1993).

Oh, J., V. Chockalingam, S. Singh and H. Lee, "Control of memory, active perception, and action in minecraft", in "Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48", ICML'16, p. 2790–2799 (JMLR.org, 2016).

Parisotto, E. and R. Salakhutdinov, "Neural map: Structured memory for deep reinforcement learning", ICLR (2017).

Pritzel, A., B. Uria, S. Srinivasan, A. P. Badia, O. Vinyals, D. Hassabis, D. Wierstra and C. Blundell, "Neural episodic control", in "34th International Conference on Machine Learning", vol. 70, pp. 2827–2836 (2017), URL http://proceedings.mlr.press/v70/pritzel17a.html.

Rae, J. W., J. J. Hunt, T. Harley, I. Danihelka, A. Senior, G. Wayne, A. Graves and T. P. Lillicrap, "Scaling memory-augmented neural networks with sparse reads and writes", in "Proceedings of the 30th International Conference on Neural Information Processing Systems", NIPS'16, p. 3628–3636 (Curran Associates Inc., Red Hook, NY, USA, 2016).

S. Das, C. L. G. 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 the Cognitive Science Society", (1992), URL https://clgiles.ist.psu.edu/papers/Cog.Sci.conf.14th.NNPDA.pdf.

S. Das, C. L. G. and G. Z. Sun, "Using prior knowledge in a nnpda to learn context-free language", in "Conference on Neural Information Processing Systems (NIPS)", (1993).

Sukhbaatar, S., a. szlam, J. Weston and R. Fergus, "End-to-end memory networks", in "Advances in Neural Information Processing Systems 28", edited by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama and R. Garnett, pp. 2440–2448 (Curran Associates, Inc., 2015), URL http://papers.nips.cc/paper/5846-end-to-end-memory-networks.pdf.

Turing, A. M., "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of the London Mathematical Society **2**, 42, 230–265, URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf (1936).

von Neumann, J., "First draft of a report on the edvac", IEEE Ann. Hist. Comput. **15**, 4, 27–75, URL https://doi.org/10.1109/85.238389 (1993).

Weston, J., S. Chopra and A. Bordes, "Memory networks", URL http://arxiv.org/abs/1410.3916, cite arxiv:1410.3916 (2014).

Wu, Y., G. Wayne, A. Graves and T. Lillicrap, "The kanerva machine: A generative distributed memory", in "International Conference on Learning Representations", (2018), URL https://openreview.net/forum?id=S1HlA-ZAZ.

Young, T., D. Hazarika, S. Poria and E. Cambria, "Recent trends in deep learning based natural language processing [review article]", IEEE Computational Intelligence Magazine **13**, 55–75 (2018).

Zaremba, W. and I. Sutskever, "Reinforcement learning neural turing machines", CoRR **abs/1505.00521**, URL http://arxiv.org/abs/1505.00521 (2015).

Zeng, Z., R. M. Goodman and P. Smyth, "Discrete recurrent neural networks for grammatical inference", in "IEEE Transactions on Neural Networks", vol. 5 (1994).