Neural Network Architecture with External Memory and Domain-aware Weight Switching Mechanism

by

Deep Chittranjan Patel

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

Approved April 2020 by the
Graduate Supervisory Committee:

Hani Ben Amor, Chair
Ayan Banerjee
Troy McDaniel

ARIZONA STATE UNIVERSITY

May 2020

ABSTRACT

Humans have an excellent ability to analyze and process information from multiple domains. They also possess the ability to apply the same decision-making process when the situation is familiar with their previous experience.

Inspired by human's ability to remember past experiences and apply the same when a similar situation occurs, the research community has attempted to augment memory with Neural Network to store the previously learned information. Together with this, the community has also developed mechanisms to perform domain-specific weight switching to handle multiple domains using a single model. Notably, the two research fields work independently, and the goal of this dissertation is to combine their capabilities.

This dissertation introduces a Neural Network module augmented with two external memories, one allowing the network to read and write the information and another to perform domain-specific weight switching. Two learning tasks are proposed in this work to investigate the model performance - solving mathematics operations sequence and action based on color sequence identification. A wide range of experiments with these two tasks verify the model's learning capabilities.

DEDICATION

Dedicated to my parents Chittranjan Patel and Snehal Patel and my brother Maharshi Patel for their invaluable contributions in my life.

ACKNOWLEDGEMENTS

I would like to thanks Dr. Heni Ben Amor for inspiring me to pursue this thesis research and constantly guiding me in my journey till here. I am also thankful to Dr. Ayan Banerjee and Dr. Troy McDaniel for providing their invaluable time and being my committee members.

I cannot forget to thank my roommates and my friends Maunil and Satyak who helped and supported me throughout my journey at ASU. Additionally, I would like to be very thankful to my friend and colleague Manthan Bhat, with whom I had very fruitful technical discussions that helped me during my work.

I am very much indebted to my parents, my brother, my grandparents and the rest of my family members who are the four pillars of my life, supporting me in every endeavor and tough times.

Finally, I would like to conclude this section by thanking God, who has provided me with the purpose and people to live for.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Consider a scenario where a human being is shown two pictures one by one - a painting and an incomplete solution of a mathematical problem. In the second step, he is asked to complete the mathematical solution and an incomplete version of the same painting. Although the human saw two images, both of them were from different domains and the information in them were completely different. One image depicted shapes, colours and various art forms, while other involved text, numbers and mathematical symbols. Thus, to complete given tasks, that person must remember two different types of information and recall appropriately when solving either task. This also involves processing each type of information differently and may require utilization of different brain functionalities.

There have been some recent advances where researchers have designed Machine Learning (ML) models and algorithms capable of real-time switching of weights using a single model to tackle multiple domain input data. (Rebuffi *et al.* (2017)) has designed a model that can predict the domain of the input data and accordingly combine appropriate domain-specific weights with domain agnostic weights in a pre-trained network in real-time. Additionally, the field of neural memory has witnessed some breakthrough inventions such as Long Short Term Memory (LSTM) networks (Hochreiter and Schmidhuber (1997)), Neural Turing Machines (NTM) (Graves *et al.* (2014)), etc. which enables an ML model to store information into memory and utilize it to make future decisions. Using these models researchers have significantly improved results on sequential tasks such as speech recognition (Graves *et al.* (2013)).

1.1 Goals and Motivations

A memory augmented Neural Network combined with a weight switching mechanism may be a good start to achieve human-level cognition described previously. This would allow the network to remember information from multiple domains and process them using domain-specific weights whenever required. However, as per my knowledge, no such attempt has been made to develop Neural Network models with such capabilities and this motivates me to pursue it. The dissertation discusses my attempt to build a Neural Network architecture with external memory and domain-aware weight switching mechanism.

1.2 Contributions

The contribution of this dissertation is the Neural Network model capable of switching weights based on the input data domain along with the ability to store current state information for future decision making. I am hopeful that this dissertation will be a reliable reference for further research in this domain.

1.3 Dissertation Outline

This dissertation is structured in the following manner.

Chapter 2 will briefly discuss research progress in the fields related to this dissertation. More specifically, a brief introduction to Recurrent Neural Networks (RNN) such as Long Short Term Memory (LSTM) network and Gated Recurrent Unit (GRU) will be provided. Moreover, Neural Turing Machine and its working will be summarized. Additionally, recent trends in domain aware weight switching mechanism will be briefly discussed in later sections.

In **Chapter 3**, my model will be introduced and described in detail. Mathematical expressions are provided to concretely explain the module mechanisms.

Experiments with model and their results are presented in **Chapter 4**. The model applied to solve two tasks. First is solving mathematical operations sequence and second is to generate actions based on the color tile sequence. Implementation details and dataset description for both the tasks are provided in detail. Furthermore, plots after varying several parameters are also presented. Along with this, detailed memory interaction for both the tasks is illustrated in this chapter.

Finally, **Chapter 5** concludes this dissertation with some future research directions in this domain.

Chapter 2

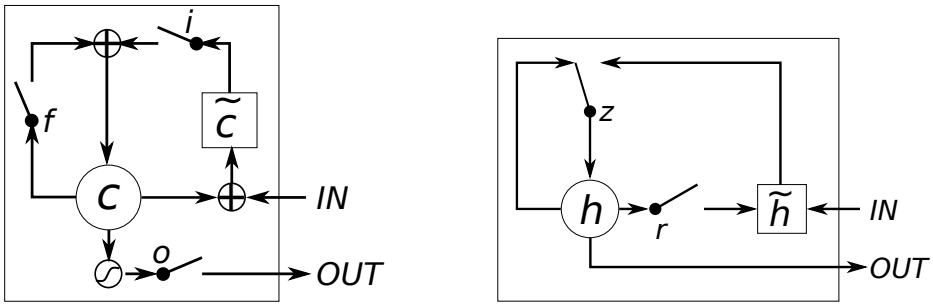
BACKGROUND

This chapter will briefly discuss some existing memory models and weight switching mechanisms from which the current work is inspired.

2.1 Recurrent Neural Networks

Recurrent Neural Network (RNN) is the Neural Networks with a feedback loop of the hidden state. Therefore, the nodes can form connections along the temporal dimension, unlike feed-forward Neural Network which are restricted to spatial dimension only. Due to the recurrence of hidden state, RNN could operate on the variable-length input data and therefore they are applied in many sequential tasks such as speech recognition (Graves *et al.* (2013)).

However, (Bengio *et al.* (1994)) observes that RNN are difficult to train to capture long-term dependencies because the gradients may either vanish or explode. Therefore, to address this concern, (Hochreiter and Schmidhuber (1997)) invented Long Short Term Memory (LSTM) network. Unlike RNN, LSTM maintains memory to store past information and also regulate storage and output using specific gates. Later, (Cho *et al.* (2014)) invented Gated Recurrent Unit (GRU) which is again a gated RNN but does not have a separate memory cell-like LSTM. Conceptual architecture of LSTM and GRU is illustrated in figure 2.1 from (Chung *et al.* (2014)).



(a) Long Short-Term Memory (b) Gated Recurrent Unit

Figure 2.1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and $\tilde{c}$ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and $\tilde{h}$ are the activation and the candidate activation (Chung *et al.* (2014)).

2.2 Neural Turing Machines

The advent of Neural Turing Machines (NTM) (Graves *et al.* (2014)) was a significant breakthrough in the field of Memory Augmented Neural Networks (MANNs). Although Recurrent Neural Networks (RNN) are Turing complete (Siegelmann and Sontag (1992)), which means they can implement several functions, the NTM authors doubted its practical possibility which prompted them to build NTM. As seen in figure 2.2, the Neural Controller controls memory read and write operations with Read and Write heads respectively. The output is computed using the input to the network and the information read from the memory.

For reading and writing to the memory, NTM employs content-based addressing and location-based addressing to calculate address weights. Both types of addressing schemes are explained in the following subsections.

2.2.1 Content-based Addressing

Content-based addressing compares each memory cell to find memory contents similar to the key vectors emitted by the read and write heads. Cosine Similarity

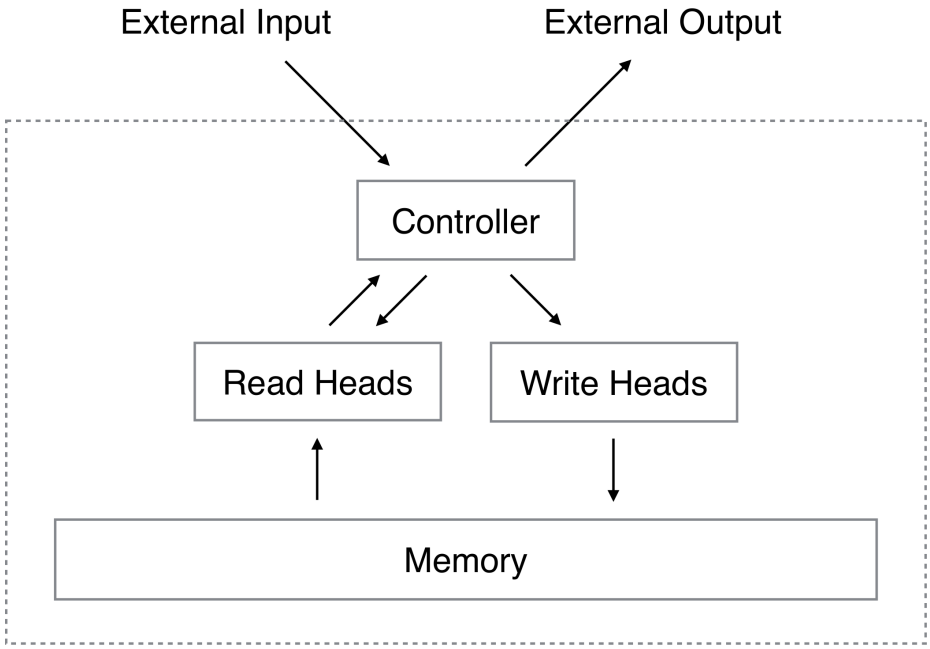


Figure 2.2: Neural Turing Machine Architecture (Graves *et al.* (2014))

is used as a similarity measure between a key vector and memory contents. After computing similarity vector, $Softmax$ operation is applied to it to normalize the vector contents between 0 and 1 and ensure that they sum to 1. It is tantamount to calculating probability values of how similar the key vector is with the contents.

As mentioned in (Graves *et al.* (2014)), content-based addressing simplifies information retrieval, merely requiring the controller to produce an approximation to a part of the stored data, which is then compared to memory to yield the stored value.

2.2.2 Location-based Addressing

NTM is capable of building a representation of the functions it is assigned to learn. For example, consider a function $f(x, y)$. Since this function operates on variables x and y which are locations storing some values and not the values itself, content-based addressing does not come handy in these situations. Therefore NTM is also capable to perform location-based addressing along with the content-based. To facilitate

location-based addressing, NTM can shift its head pointers to the desired memory location. The movement of the heads is carried out by convolution operation of gated weights $\mathbf{W}_g^t$ with *shift weighting* S_t emitted by the head itself. The mathematical representation of location-based addressing is provided below:

$$\tilde{w}_t(i) \longleftarrow \sum_{j=0}^{N-1} w_g^t(j)\, s_t(i-j). \tag{2.1}$$

here, gated weights $\mathbf{W}_g^t$ are calculated as follows:

$$\mathbf{W}_g^t \longleftarrow g_t \mathbf{W}_c^t + (1 - g_t). \tag{2.2}$$

The above equation is basically interpolation between content based weights $\mathbf{W}_c^t$ and previous weights $\mathbf{W}^{t-1}$ carried out by gated parameter g_t emitted by the head.

2.2.3 Extensions and Applications of NTM

After the introduction of NTM, many researchers applied it to several problems in various domains. One of the earliest applications of NTM was in the meta-learning domain. (Santoro *et al.* (2016)) used NTM for one-shot classification of Omniglot dataset images (Lake *et al.* (2015)). Since NTM was used to store unseen information after a single presentation to the model, location-based addressing was absent in the author's version because according to them, content-based addressing was sufficient. Moreover, the Least Recently Used Access mechanism was used to generate write weights to avoid loss of important information.

Further, the Differentiable Neural Computer (DNC) (Graves *et al.* (2016)), yet another MANN was introduced two years after NTM. Architecturally, it is heavily influenced by NTM and some of its functionalities such as content-based addressing are also derived from NTM. However, many significant features that lacked in NTM such as freeing up of unused memory, preserving of sequential information even after write head move on to another location, etc. are implemented in DNC.

2.3 Domain-aware Weight Switching Mechanism

Apart from the externally augmented memory, another crucial part of my model is domain aware weight switching mechanism. This mechanism allows loading and unloading of task-specific weights to and from Neural Network during run-time. The input given to my architecture at every step could be of different domains and weight parameters stored in the weight memory module have to be pre-trained on their respective domain inputs to achieve good results. Therefore, at the input step, the domain of the input must be detected and appropriate weights should be selected to load into Neural Network.

(Rebuffi *et al.* (2017)) attempted to solve a similar problem in which they trained a single model to classify images from different domains. In the experiments, they showed that learning ResNet (He *et al.* (2016)) or similar networks directly on multiple domain data may not perform well although they may be good feature classifiers. Therefore, the authors trained separate domain-specific weight modules and augmented them with domain agnostic weights during classification. Moreover, during testing, a domain of the data may not be given, however, this information is crucial to select the appropriate weights to augment. Therefore, the authors recommended training a separate classifier to detect the data domain which could then be used to select the appropriate weights.

Chapter 3

PROPOSED MODEL ARCHITECTURE

The External Neural Memory and Weights Architecture (ENMWA) developed by me is a hybrid Neural Network architecture that stores the current state results and utilizes it to calculate the new results whenever a similar state is encountered in the future. The results are calculated by a feed-forward Neural Network using state-specific weight parameters. In other words, the Neural Network itself decides the best weight parameters to load by determining the current state details (here, the data domain) to calculate the results. This chapter describes ENMWA in detail.

The chapter is organized as follows. The overall model architecture and algorithm will be described in section 3.1. Section 3.2 describes the Neural Controller architecture and its functions. The detailed working of the External Memory Module including memory read and write operations and their weight computations are explained in section 3.3. The external weights memory module, which stores the domain-specific pre-trained weights is discussed in detail in section 3.4. Finally, section 3.5 provides a brief description of the feed-forward Neural Network module.

3.1 Model Architecture

The ENMWA architecture is illustrated in figure 3.1. The Neural Controller module co-ordinates all other modules in the architecture and thus is a crucial part of it. Given an input, the Neural Controller outputs *Access Key*, *Weight Probabilities* and *Network Input* for the feed-forward Neural Network. The *Weight Probabilities* are the probability distribution on the pre-trained domain-specific weights stored in the Weight Memory. These probability values are calculated based on the domain of

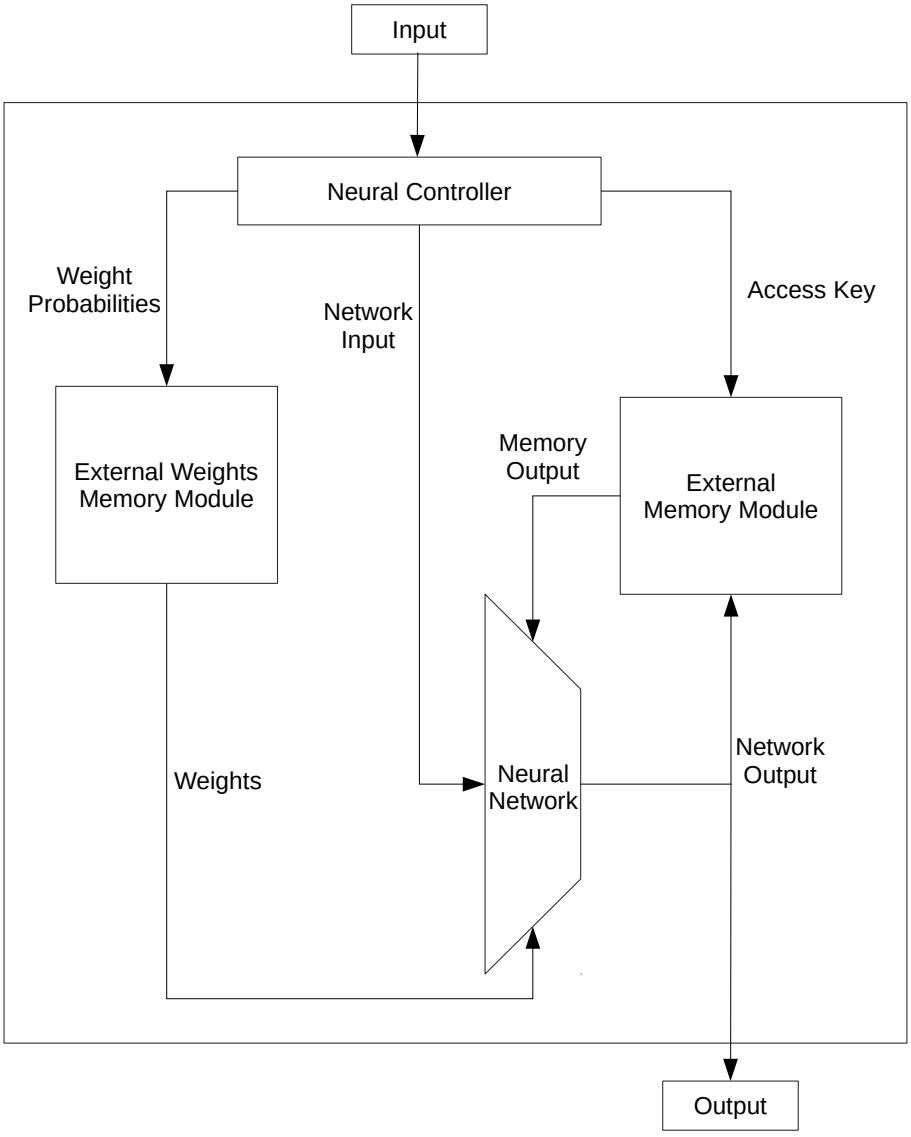


Figure 3.1: External Neural Memory and Weights Architecture (ENMWA) diagram.

the input data. Thus intuitively, the best weights corresponding to the input data domain (or the highest probability) are loaded into the Neural Network. On the other hand, the *Access Key* is nothing but the input data features calculated by the Neural Controller. These features serve the purpose of the current state information and are used as the key to store and retrieve the results computed by the Neural Network to and from the Memory Module. Additionally, note that the *Network Input* from

Algorithm 1 External Neural Memory and Weights Architecture

Input: Current Input Sample $\mathbf{X}$.

Output: Computed result $\mathbf{R}$ from the feed-forward Neural Network.

1: Compute *Weight Probabilities*, *Network Input* and *Access Key(s)* from the controller as $\mathbf{P}_w$, $\mathbf{I}_N$, $\mathbf{K}$ = *Neural_Controller(**X**)*.

2: Retrieve stored information $\mathbf{I}_{mem}$ by doing *Read_Memory(**K**)* operation.

3: Load weights $\mathbf{W}_p$ from weight memory to feed-forward Neural Network using $\mathbf{P}_w$.

4: Now compute $\mathbf{R} = Neural\_Network(\mathbf{W}_p, \mathbf{I}_{mem}, \mathbf{I}_N)$.

5: Store $\mathbf{R}$ back to the memory by performing *Write_Memory(**K**)* operation.

the Neural Controller may not be explicitly required for tasks where *Memory Output* from the Memory Module is sufficient for the feed-forward network to compute a new result (implicitly using the input).

The feed-forward Neural Network using the loaded weight parameters and inputs from the Memory Module and Neural Controller computes the new result and this is then stored back into the memory against the *Access Key*. Finally, algorithm 1 depicts the general functioning of ENMWA.

3.2 Neural Controller

Neural Controller coordinates other modules of the architecture by sending various control signals. It takes the input $\mathbf{X}$ and outputs *Weight Probabilities* $\mathbf{P}_w$, *Network Input* $\mathbf{I}_N$ and *Access Key(s)* $\mathbf{K}$. By design, the Neural Controller is implemented using either a simple feed-forward Neural Network or a Convolutional Neural Network depending on the nature of the input data.

The *Access Key(s)* $\mathbf{K}$ are features of the input data which serves as the key in the memory against which the Neural Network outputs or initial values will be stored. As

per the requirements, at every current state the model should retrieve the computed values corresponding to the similar states encountered in the past from the memory. This requires a unique key for each state, which when queried from the memory, don't interfere with other state keys. Based on the fact that a good feature classifier generates unique features for dissimilar input pairs, the input features generated by the Neural Controller are thus used as the state keys to query from the memory.

The weight memory module stores set of weight parameters each trained to execute inputs specific to a domain or state type. Therefore before loading weight parameters to the feed-forward Neural Network, it has to be decided which one to load. Thus, the Neural Controller after identifying domain of input data generates probability distribution $\mathbf{P}_w$ over the set of weight parameters and accordingly those are selected. From the experiments it is observed that, training Neural Controller to classify states for weight parameters selection will also train it to generate unique input features for each state which then could be used as the *Access Key* $\mathbf{K}$ to the memory module.

The *Network Input* $\mathbf{I}_N$ is used by Neural Network to generate output at current state. The input to ENMWA in some scenarios contain separate input for Neural Network and state information to be fed to Neural Controller. Thus $\mathbf{I}_N$ can bypass Neural Controller in this case. The other scenario may require Neural Controller to explicitly generate $\mathbf{I}_N$. While in some scenarios, the Neural Network may use output from the External Memory Module only. In these cases, input to the ENMWA indirectly influence final output via the External Memory Module.

3.3 External Memory Module

The External Memory Module stores the values computed at the input states against key vectors which are nothing but input data features computed by Neural Controller. The memory is effectively an $N \times M$ matrix, where N is the Number of

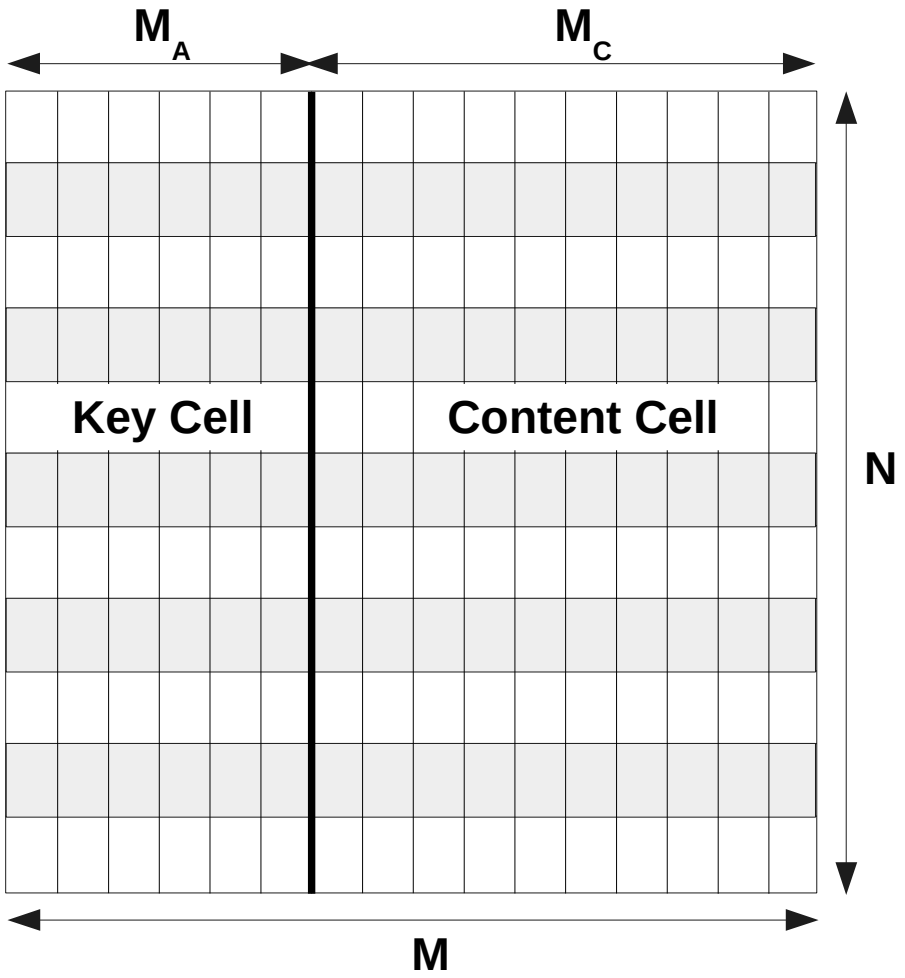


Figure 3.2: $N \times M$ Memory block.

cells and M is the size of each cell. As apparent from figure 3.2, each M sized cell is divided into two parts of size M_A and M_C respectively. The first section stores the Key and the second section stores the corresponding content. It is on the key section, over which attention is performed during the reading phase to retrieve the content.

Unlike NTM and DNC (Graves *et al.* (2014, 2016)), reading and writing operations are disjoint. Thus, a separate command must be given to the module for reading and writing to it. The reading operation is content-based, in other words, attention over the key cells in the memory is performed using the current state key to find the specific address and eventually retrieve the content stored at that position. The

writing operation involves attention to find and erase the older content and Least Recently Used Access (LRUA) (Santoro *et al.* (2016)) based write weight calculation to write newer content into the memory.

3.3.1 Reading

Let, $\mathbf{M}_t \in \mathbf{R}^{N \times M}$ be memory state and $\mathbf{W}_R^t \in \mathbf{R}^N$ be read weights at time t. Since, the $\mathbf{W}_R^t$ will be normalized, each of its ith value $w_R^t(i)$ will satisfy following constraint:

$$\sum_{i=1}^{N} w_R^t(i) = 1, \qquad 0 \leq w_R^t(i) \leq 1, \forall i. \tag{3.1}$$

Considering this, the intermediate read vector $\mathbf{r}_t \in \mathbf{R}^M$ is calculated as weighted combination of row vectors of $\mathbf{M}_t$ w.r.t $\mathbf{W}_R^t$ as follows:

$$\mathbf{r}_t \longleftarrow \sum_{i=1}^{N} w_R^t(i)\mathbf{M}_t(i). \tag{3.2}$$

However, $\mathbf{r}_t$ would be a combination of key vector and content vector as it is directly fetched from the memory. Thus, content vector must be extracted from it first and then be returned as final read output as shown below:

$$\mathbf{r}_t^{Final} = \mathbf{r}_t[M_A : M_A + M_C]. \tag{3.3}$$

3.3.2 Writing

Writing into the memory is done in two stages. In the first stage, existing content against the given key is erased and in the second stage, the new content is written against that key. Leveraging erase operation depends on the tasks the model is applied to. Some tasks may require multiple information to be stored for the same state and use all of them in future decisions. Thus, it is advised to use erase operation appropriately after determining task needs. However, to exhaustively describe each

component of the model, erase operation is described in this sub-section. Let, $\mathbf{W}_E^t \in \mathbf{R}^N$ be erase weights and $\mathbf{W}_W^t \in \mathbf{R}^N$ be write weights at time t. Like $\mathbf{W}_R^t$, $\mathbf{W}_E^t$ and $\mathbf{W}_W^t$ are also normalized, thus both satisfy following constraints:

$$\sum_{i=1}^{N} w_E^t(i) = 1, \qquad 0 \leq w_E^t(i) \leq 1, \forall i, \tag{3.4}$$

and,

$$\sum_{i=1}^{N} w_W^t(i) = 1, \qquad 0 \leq w_W^t(i) \leq 1, \forall i. \tag{3.5}$$

where $w_E^t(i)$ and $w_W^t(i)$ are the ith values of $\mathbf{W}_E^t$ and $\mathbf{W}_W^t$ respectively. Now let, $\mathbf{K}_t \in \mathbf{R}^{M_A}$ and $\mathbf{C}_t \in \mathbf{R}^{M_C}$ be key vector and content vector to be written in the memory at time t. Both the stages of writing operation is mathematically depicted in equations 3.6 and 3.7 respectively.

$$\tilde{\mathbf{M}}_t(i) \longleftarrow \mathbf{M}_{t-1}(i)[1 - w_E^t(i)\mathbf{e}_t]. \tag{3.6}$$

$$\hat{\mathbf{M}}_t \longleftarrow \tilde{\mathbf{M}}_t(i)[1 - w_W^t(i)\mathbf{e}_t], \tag{3.7a}$$

$$\mathbf{M}_t \longleftarrow \hat{\mathbf{M}}_t(i) + w_W^t(i)[\mathbf{K}_t; \mathbf{C}_t]. \tag{3.7b}$$

3.3.3 Addressing Mechanism

The read and write weights $\mathbf{W}_R^t$, $\mathbf{W}_E^t$, $\mathbf{W}_W^t$ are basically the cell addresses which are affected during read and write operations in memory. However, computing these addresses requires different approaches. The mathematical representation of these approaches will be provided in the following sub-sections.

3.3.3.1 Calculating Read Address

Read Address calculation employs *Focusing by Content* strategy given in (Graves *et al.* (2014, 2016)). In this strategy given a key vector $\mathbf{K}_t$, attention is performed on the memory cells w.r.t $\mathbf{K}_t$ to identify the cells having similar keys and eventually

retrieve content from them. However, this model performs attention over $\mathbf{M}_A$ sized *Key Cell* of the memory rather than performing attention over whole $\mathbf{M}$ sized memory strip as done in (Graves *et al.* (2014, 2016)). Each read weight value $w_R^t(i)$ is computed as shown in the following equation:

$$w_R^t(i) \longleftarrow \frac{\exp\Big(\beta \cdot F\big(\mathbf{K}_t, \mathbf{M}_t(i)[0:\mathbf{M}_A]\big)\Big)}{\sum_j \exp\Big(\beta \cdot F\big(\mathbf{K}_t, \mathbf{M}_t(j)[0:\mathbf{M}_A]\big)\Big)}. \tag{3.8}$$

Notice that the above operation applies $Softmax(\cdot)$ function to the output of $F(\cdot)$ to calculate $\mathbf{W}_R^t$. Here depending on the application, $F(\cdot)$ could either be calculated using Euclidean Distance between input vectors as done in equation 3.9 or Cosine Similarity as calculated in equation 3.10. Moreover, β is a scaling constant which could be set to stabilize results.

$$F(\mathbf{X}, \mathbf{Y}) = 1 - \|\mathbf{X} - \mathbf{Y}\|. \tag{3.9}$$

$$F(\mathbf{X}, \mathbf{Y}) = \frac{\mathbf{X} \cdot \mathbf{Y}}{\|\mathbf{X}\| \cdot \|\mathbf{Y}\|}. \tag{3.10}$$

3.3.3.2 Determining Erase Address

The erase address or erase weights $\mathbf{W}_E^t$ are also calculated using content focusing as depicted in equation 3.8. Therefore these weights are equivalent to reading weights $\mathbf{W}_R^t$, but the key $\mathbf{K}_t$ may be different than that used during reading. Thus, they may differ from $\mathbf{W}_R^t$ due to this.

3.3.3.3 LRUA Mechanism for Computing Write Address

The write weights must ensure that the content it is going to write does not affect the recently written contents into the memory and thus use the least recently used location to write new content. Inspiring from (Santoro *et al.* (2016); Gulcehre *et al.*

(2018)) this work utilizes a modified LRUA mechanism to compute the write weights or write address $\mathbf{W}_W^t$. In every pass at time t, usage weights $\mathbf{W}_U^t$ are calculated according to the following relation:

$$\mathbf{W}_U^t \longleftarrow \gamma \mathbf{W}_U^{t-1} + \mathbf{W}_R^t + \mathbf{W}_W^t \tag{3.11}$$

here γ is the decay factor. The usage weights records the location used in the current pass. Consider the notation $m(\mathbf{V}, n)$, which indicates n^{th} smallest element in vector $\mathbf{V}$. Now, the least used weights $\mathbf{W}_{LU}^t$ could be calculated as follows:

$$w_{LU}^t(i) = \begin{cases} 0 & \text{if } w_U^t(i) > m(\mathbf{W}_U^t, n) \\ 1 & \text{if } w_U^t(i) \leq m(\mathbf{W}_U^t, n) \end{cases} . \tag{3.12}$$

here n is set to be the number of reads from the memory. Finally the write weights $\mathbf{W}_W^t$ are set to be $\mathbf{W}_{LU}^{t-1}$.

3.4 External Weights Memory Module

The External Weights Memory Module is an interesting part of the ENMWA because it allows real-time loading and unloading weight parameters to and from the feed-forward Neural Network. As observed from Figure 3.3 the module stacks linearized pre-trained weight parameters for each task in itself. Let, $WP(i)$ represent i^{th} weight parameter having dimension $1 \times D_{WP}$. Also, assume that there are R such weight parameters stacked on top of each other. Thus, overall weight parameter volume would be of dimension $R \times D_{WP}$. Now, weight parameters selection for loading into the Neural Network is done as follows:

$$W^{final} = \sum_{i=1}^{R} P_w(i) \cdot WP(i) \tag{3.13}$$

where $P_w(i)$ is i^{th} element of R dimensional probability vector P_w generated by Neural Controller. However, W^{final} is still $1 \times D_{WP}$ dimensional vector, but it will be split and reshaped into appropriate dimensions before loading into the Neural Network.

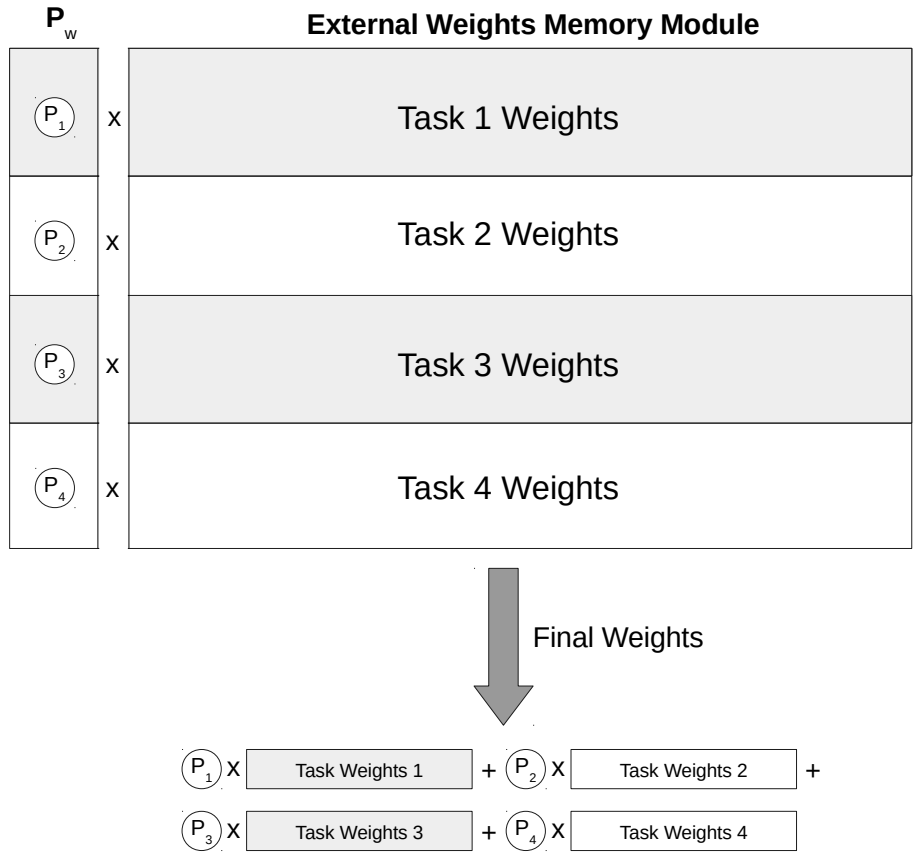


Figure 3.3: External Weights Memory Module.

3.4.1 Updation of Switchable Weight Parameters

Since, the memory module performs content-based focusing during reading, the read weights W_R may be noisy due to the presence of other contents in the memory and thus resulting output from the memory may also possess noise. Now, it could be possible that the pre-trained weight parameters in the External Weights Memory Module are trained on the noise-free data and thus may generate poor results when noisy memory outputs are fed to them. However, during experiments, it was observed that the pre-trained weights could be updated when backpropagation of the gradients is turned on in them. This phenomenon is an unintended side-effect of the model, with a benefit. It allows certain weights to adapt without affecting other weights.

3.5 Feed-forward Neural Network

The feed-forward Neural Network module performs the computation in the architecture. It operates on the memory output and input from the Neural Controller by using the pre-trained weights from the External Weight Memory Module. The output of the Neural Network is then stored back in the memory against the generated Key vector. Layers of the network and size of each hidden layer vary according to the complexity of the task. This size must be predetermined because it becomes the reference size for the pre-trained weight parameters in weight memory.

Chapter 4

EXPERIMENTS AND ANALYSIS

This chapter presents the experiments performed on the ENMWA and their results. Two tasks were selected for the experiments. The first task requires the model to generate action vectors based on the input color sequences. The second task requires the model to solve a sequence of the mathematical operations (addition, subtraction, division, and multiplication) on real numbers. Each section of this chapter describes an experiment including dataset description, task description, implementation details, results, and observations.

4.1 Actions Based on Color Sequence

4.1.1 Input Data Description

The input data for this task would be randomly generated sequences of colored tiles arranged in random order; an example of which is shown in figure 4.1. The tiles could be red, green or white with a majority of them being white. Moreover, randomly selected pixels in each tile could be replaced with truncated Gaussian noise samples. Since the dataset is functionally generated, the parameters that controls its generation are *sequence length*, *number of red and green tiles* in a sequence, and *amount of noisy pixels* in each tile.

4.1.2 Task Description

Three pre-trained weights are stored in the weight memory of ENMWA, each associated with a specific tile color. When a colored tile is passed to the model,

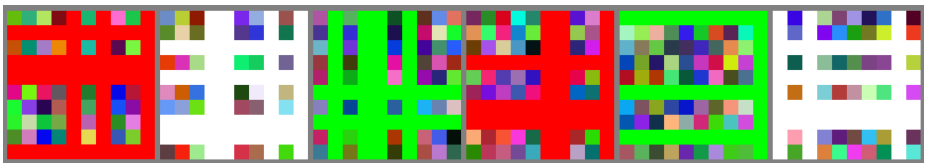


Figure 4.1: A sample input sequence of noisy color tiles (Green, Red and White).

Table 4.1: Sample input-output pair combination of weight parameters.

Weights corresponding to colored tiles	Input	Output
Red	[1, 0, 0]	[0, 0, 1]
	[0, 0, 1]	[1, 0, 0]
Green	[1, 0, 0]	[0, 1, 0]
	[0, 1, 0]	[1, 0, 0]
White	[1, 0, 0]	[1, 0, 0]

the model is expected to load corresponding weight from the weight memory module and information stored during the last encounter of the similar colored tile; from the external memory module. Using this information and weight parameters, the Neural Network will calculate the new output and store it back to the memory.

The input and output to the feed-forward Neural Network are one-hot vectors. Each pre-trained weight parameter is trained to output specific configurations of one hot vector for a given input. Moreover, no two weight parameters would have the same input-output pair combination. This constraint ensures that the model must choose appropriate weight parameters based on the given tile color to generate the correct output. For example, for the current task, let weight parameters corresponding to each colored tile has input-output relation as shown in table 4.1, then input $[0, 0, 1]$ which is meant for red tile will yield incorrect output when evaluated using green weights. Further, to measure model performance, the F1 score of the model output has been used.

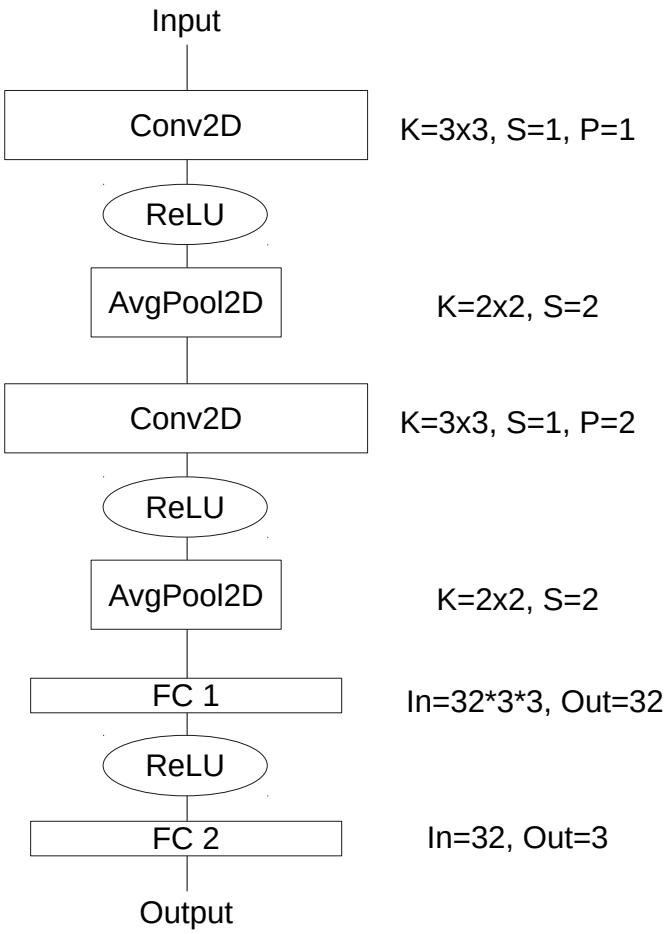


Figure 4.2: Controller Network Architecture.

4.1.3 Implementation Details

Input features and a probability distribution over pre-trained task-specific weights, which are crucial for the operation of the external memory module and weight memory module respectively, are both generated by the neural controller module. For the current task, the controller network was implemented using Convolution Neural Network (CNN) with two convolutional layers. Figure 4.2 illustrates the architecture of controller network. *FC2* layer outputs the probability distribution over weight parameters (after taking $Softmax$) and the same output is used as memory keys because it also represents the unique features of the input.

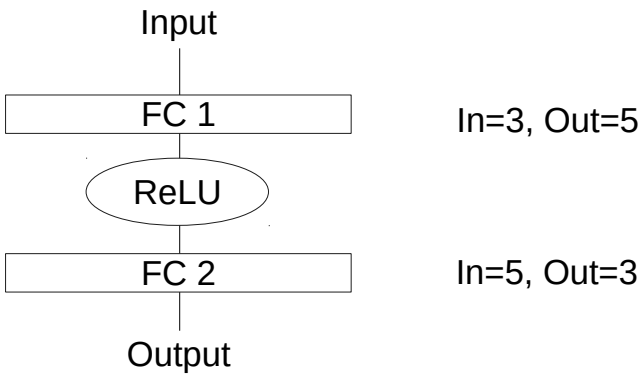


Figure 4.3: Feed-forward Neural Network architecture (complying with pre-trained weight parameters stored in weight memory module).

The whole controller network was trained end-to-end using cross-entropy loss and ADAM optimizer (Kingma and Ba (2014)). The input data was divided into 1000 batches with a batch size of 100 samples. In each batch, there was roughly an equal number of tiles of each color (White, Green, and Red). In every 10x10 shaped tile, 25 random pixels were selected and replaced with truncated Gaussian noise. The network yielded around 94% classification accuracy.

Pre-trained weight parameters in the weight memory module were independently trained on corresponding input-output pair combinations given in table 4.1. The training involved optimizing cross-entropy loss using the ADAM optimizer (Kingma and Ba (2014)). The network architecture of each of the pre-trained weights is the same and is illustrated in figure 4.3. To be compatible with the pre-trained weight parameter size, the feed-forward Neural Network does also has the same architecture.

The memory matrix in the external memory module is a simple PyTorch (Paszke *et al.* (2019)) array having each memory strip of size 35 (also denoted by M) with 32 allocated to address field and 3 allocated to content field. The address field in memory is initialized with an array of zeros and the content field with [1, 0, 0]. The weight decay factor γ is set to 0.8. Moreover, for Cosine Similarity based attention in memory, the β was set to 10.0.

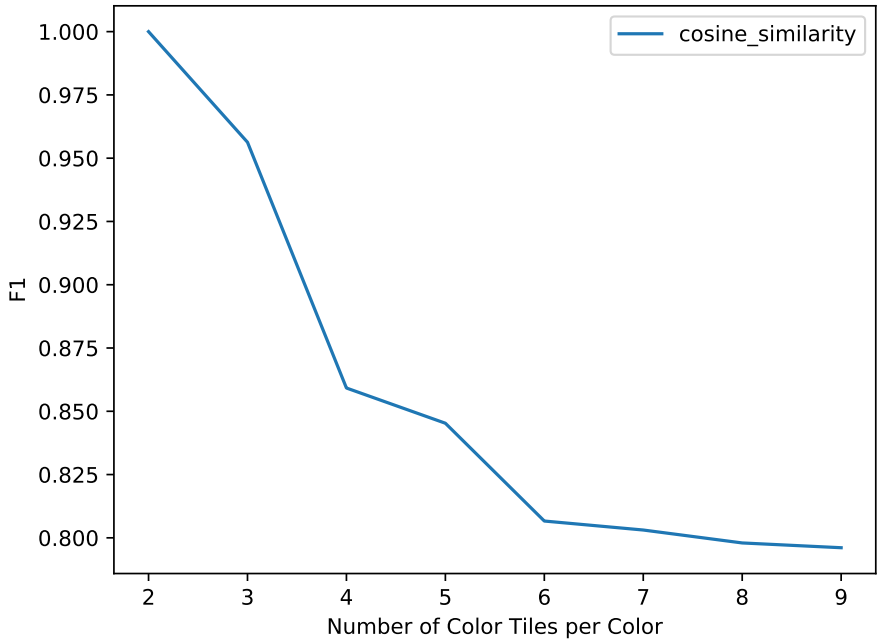


Figure 4.4: F1 score vs. number of red and green blocks in a sequence. Noise level=12.5%, Memory size=25, Input samples=1000.

4.1.4 Results

To perform attention during memory read and write, the model can either use Cosine Similarity or compute similarity using Euclidean distance between keys. However, for this task, Cosine Similarity is used for attention as it performs better than that using Euclidean distance when comparing vectors. Plots for F1 score vs model parameters are depicted in figure (4.4, 4.5, 4.6). Mainly, plots were created by varying parameters such as *number of Green and Red blocks in a sequence*, *memory size N*, *number of noisy pixels in a tile*.

As the number of colored blocks in the sequence increases, so does the requirements to store them in memory. Due to frequent read and write operations, interference may result in memory contents and this may affect the performance of the model which is evident from figure 4.4. Moreover, as observed from figure 4.5 as the memory size increases so does the performance of the model. Small memory creates limited space

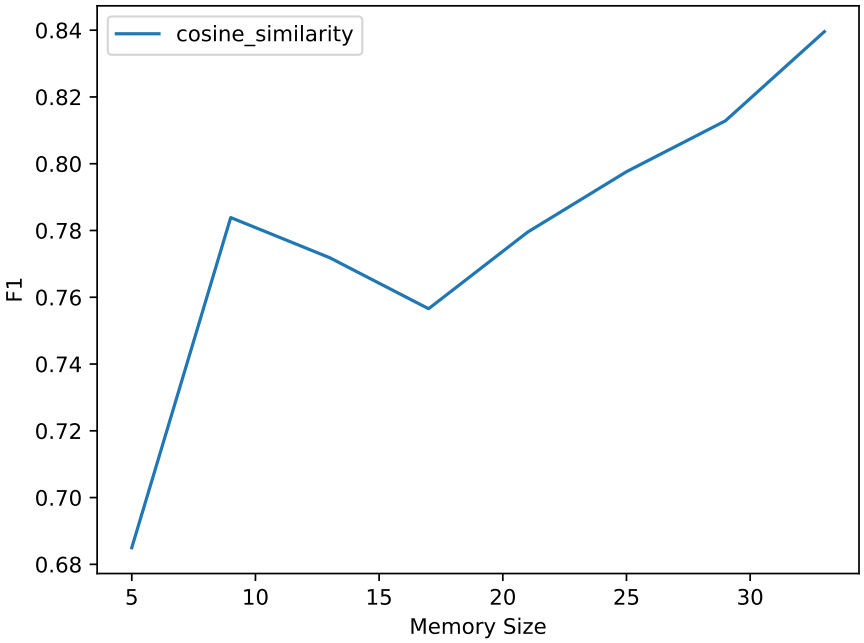


Figure 4.5: F1 score vs. memory size. Noise level=12.5%, Sequence length=20, Number of blocks each of red and green=8, Input samples=1000.

for storage, this may result in overwriting of old information as the new information is stored. Moreover, in small memory, information written two or three steps ago may be classified as least recent by LRUA mechanism and thus become vulnerable to be overwritten, hence negatively affecting the model performance. Figure 4.6 shows that as the noise in the tiles is increased, the model performance is severely affected. The input noise hampers the classification capability of the Neural Controller. Therefore the memory keys and weight probability distribution vector which are directly associated with Neural Controller gets severely affected. Eventually, this results in poor performance of the model.

4.1.5 Observations

Figure 4.7 illustrates the read and write activity in the memory for a given input sequence. Initially, when the memory is empty, none of the colored block features will

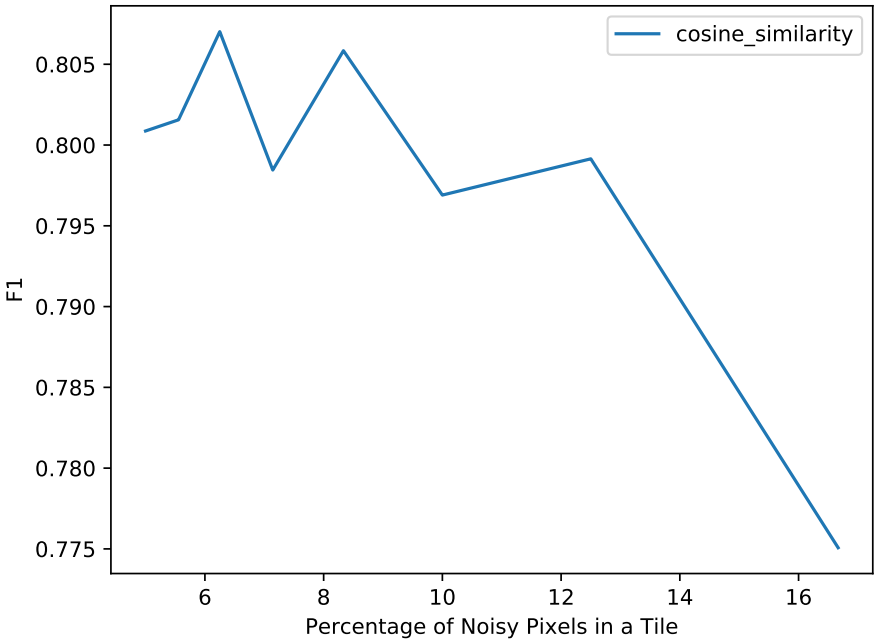


Figure 4.6: F1 score vs. noise level in tiles. Memory size=25, Sequence length=20, Number of blocks each of red and green=8, Input samples=1000.

be present in the memory. Therefore, the read weights will be blank (here depicted as fully black in color) and the default values $[1, 0, 0]$ will be given to the feed-forward Neural Network from the external memory module. Based on the given input, the Neural Network will compute certain output which will then be stored into the memory according to write weights. When the same colored blocks are again encountered by the model, the read weights will have hits (visualized using white color pixels) at locations where their corresponding model outputs were stored in the past. The values stored at the appropriate locations (previous network output) will be retrieved and given out by the memory module. Again the Neural Network will compute new outputs and store them at locations pointed by the write weights.

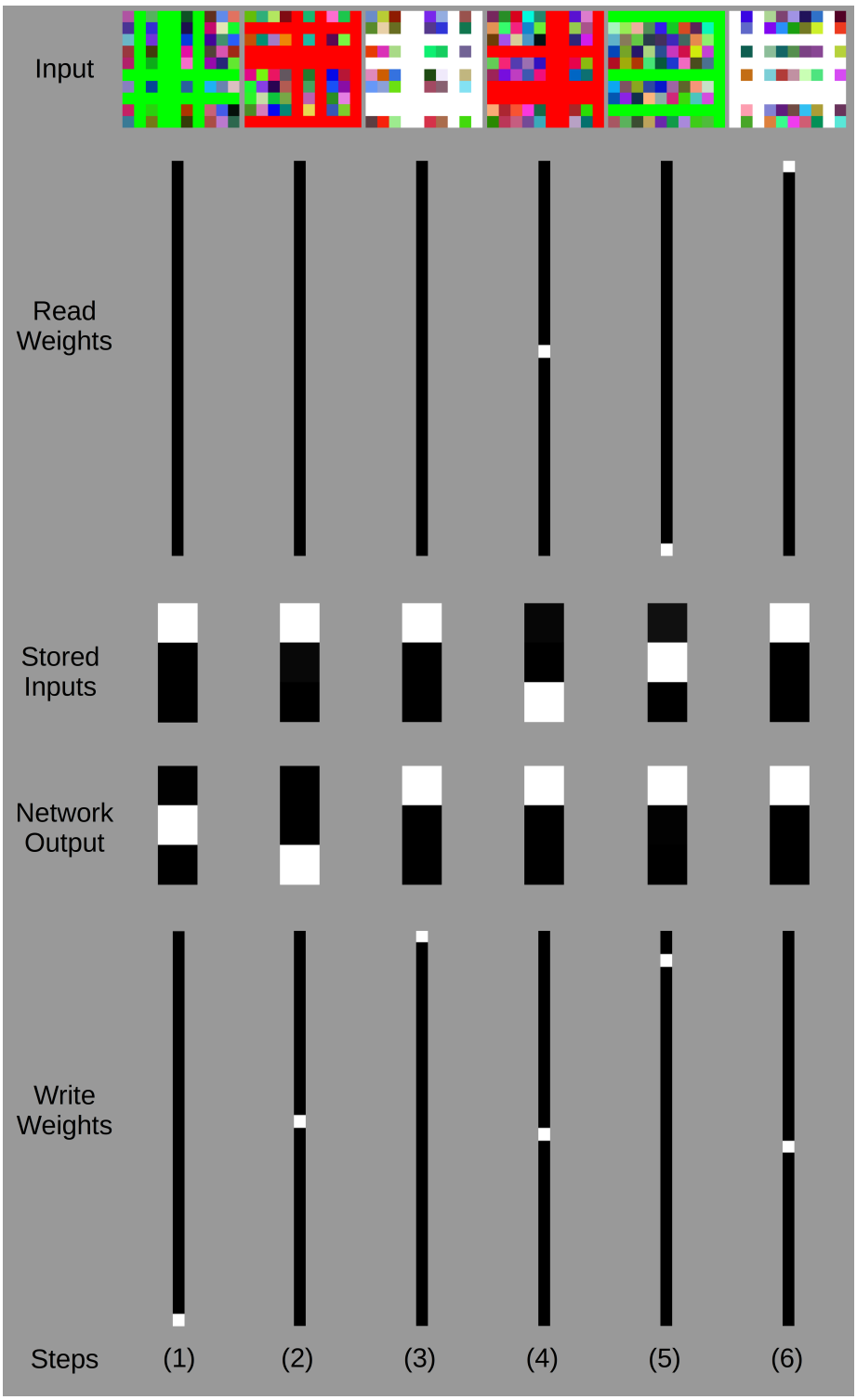


Figure 4.7: The figure shows the steps carried out in the memory when a sequence of colored tiles are given to the model.

4.2 Sequence of Mathematical Operations

4.2.1 Input Data Description

The dataset for the mathematical operations task consists of sequences of mathematical operations on real numbers. A sequence of mathematical operation is functionally generated by combining all four mathematical operations (*addition*, *subtractions*, *multiplication* and *division*) with randomly generated real numbers in random order. However, during data generation, all the mathematical rules are followed, for example avoiding 0 in denominator during division. Number of *max operations* and *max digit range* are the parameters that control data generation. A sample math operation sequence from the dataset is shown below.

$$(((19 * 5)/(15/8))/((11/19) * (16 - 3)))$$

4.2.2 Task Description

As shown in figure 4.12 (a), input expression is first converted to the expression tree before inputting to the model. Every node of an expression tree has a unique address (randomly assigned), operation and operands. The operands could either be numbers (leaf nodes) or address of child nodes. Therefore before evaluating a parent node, its child nodes must be evaluated first. At each step, node selection from the tree is done using post-order traversal and that node is presented to the model for evaluation. Based on the given operation, the controller network selects the appropriate pre-trained math operation weights and load it in the feed-forward Neural Network. The feed-forward Neural Network will either directly receive operand inputs (when they are numbers) or from external memory (when operands are to be retrieved from memory at given address). After computing final output, it is stored into a memory against the key which is also the address of the input node.

Since the expression tree is made up of nodes in hierarchical order, the output of lower nodes must first be computed and stored somewhere before moving to higher nodes. The external memory provides this storage requirement to the model. The node outputs could be store against their address in the memory and later retrieved when evaluating parent nodes. Moreover, the weight memory provides efficient switching of pre-trained math operation weights to and from the Neural Network.

4.2.3 Implementation Details

The weights for addition and subtractions operations were independently pre-trained on real numbers between -100000 to 100000. Mean squared error loss and Adam optimizer (Kingma and Ba (2014)) were utilized to train the weights. The network architecture used to train the weights is illustrated in figure 4.8. Since the pre-trained weights will be loaded to the feed-forward Neural Network in the model, to make it compatible with their shape, its network architecture is also kept the same.

The weight memory stores weights for only two operations as it is difficult to implement multiplication and division operations for generalized input range using a feed-forward neural network. However, these operations are realized by performing log and exponential operations on the input and output of the addition and subtraction operations respectively. The neural controller decides to perform log and exponential operations on inputs and outputs based on the given mathematical operation.

The controller network generates the probability distribution over weights (using $Softmax$) and decision to perform log and exponential operation (again a probability, but using $Sigmoid$), given an operation encoded into a one-hot vector. The controller network, after training, is augmented with external memory and the Neural Network. As seen in figure 4.9, the Neural Controller is built using two separate feed-forward Neural Network, one controlling weight switching and other controlling log and ex-

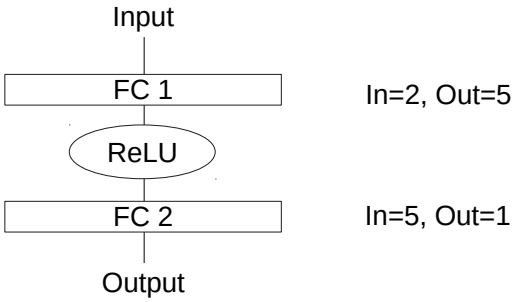


Figure 4.8: Feed-forward Neural Network architecture (complying with pre-trained weight parameters stored in weight memory module).

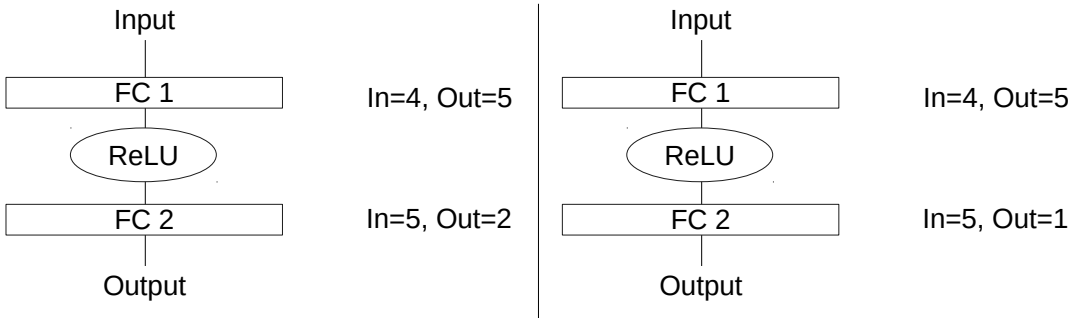


Figure 4.9: Neural Controller architecture for math operation sequence task.

ponential operations. The controller was trained using mean squared error loss and ADAM optimizer (Kingma and Ba (2014)) on number operations. During training, it was clubbed with pre-trained operation weights and learned to select appropriate weights and operations according to the input. The inputs during training were two numbers and a math operation encoded into a one-hot vector and it was expected to output the correct answer.

The address of tree nodes serve as their key during memory read and write operations because they uniquely represent each tree node. To perform attention during reading, the Euclidean distance based similarity metric is utilized, as Cosine Similarity would not yield correct comparisons for scalars. Moreover, each memory slot is two columns wide, one storing address and other the value.

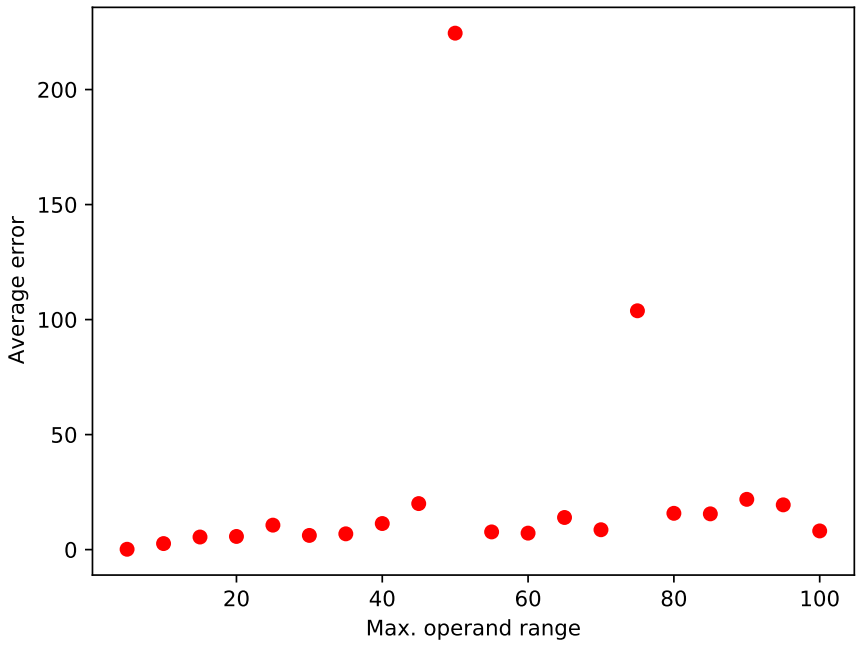


Figure 4.10: Mean squared error vs. max input number range in a sequence plot. N=32, Number of operations=12, Input samples=500.

4.2.4 Results

To asses the scalability of the model, two parameters *number of operations in sequence* and *input number range* were varied for and performance of the model in terms of mean squared error was noted on 500 input samples. The size of the memory was kept to 32 slots. Figure 4.10 and 4.11 shows plots of both the experiments.

It could be observed from both the plots that the model can maintain consistency in its performance despite the increase in input number range or number of operations in a sequence. Evidently, at some points in both the plots, there is an abrupt spike in the error. The reason for this could be precision error in large numbers. For example, for sequences expecting large output such as 37340067.32, the model may output 3700323.12. Thus although the higher significant digits are correct in this output, the incorrect trailing digits results in large error.

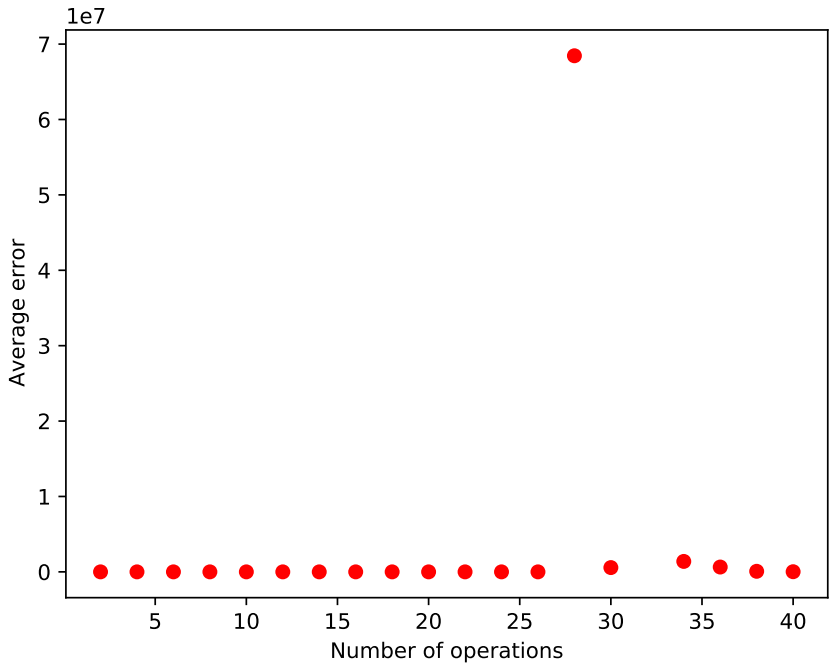


Figure 4.11: Mean squared error vs. number of operations in a sequence plot. N=32, Max. number range=20, Input samples=500.

4.2.5 Observations

Figure 4.12 shows the memory activity for read and write when a sample sequence is passed to the model. Specifically, figure 4.12 (a) shows the expression tree of the given expression. Figure 4.12 (b) shows the corresponding memory interaction (white pixels indicate hits). The W and R written above each memory strip indicates that they are write and read weights respectively. For some transactions, absence of read weights indicate inputs to Neural Network were numbers instead of addresses and thus it was needless to search into memory. Some read weights are two strip in width, this indicates that both the inputs to the network were extracted from memory.

Node C in the tree, depends on the output of nodes A and B to operate. Therefore, it is evident from the its read strip that it retrieved the numbers form the same positions where the output of nodes A and B were written. All the memory interactions follow the similar pattern.

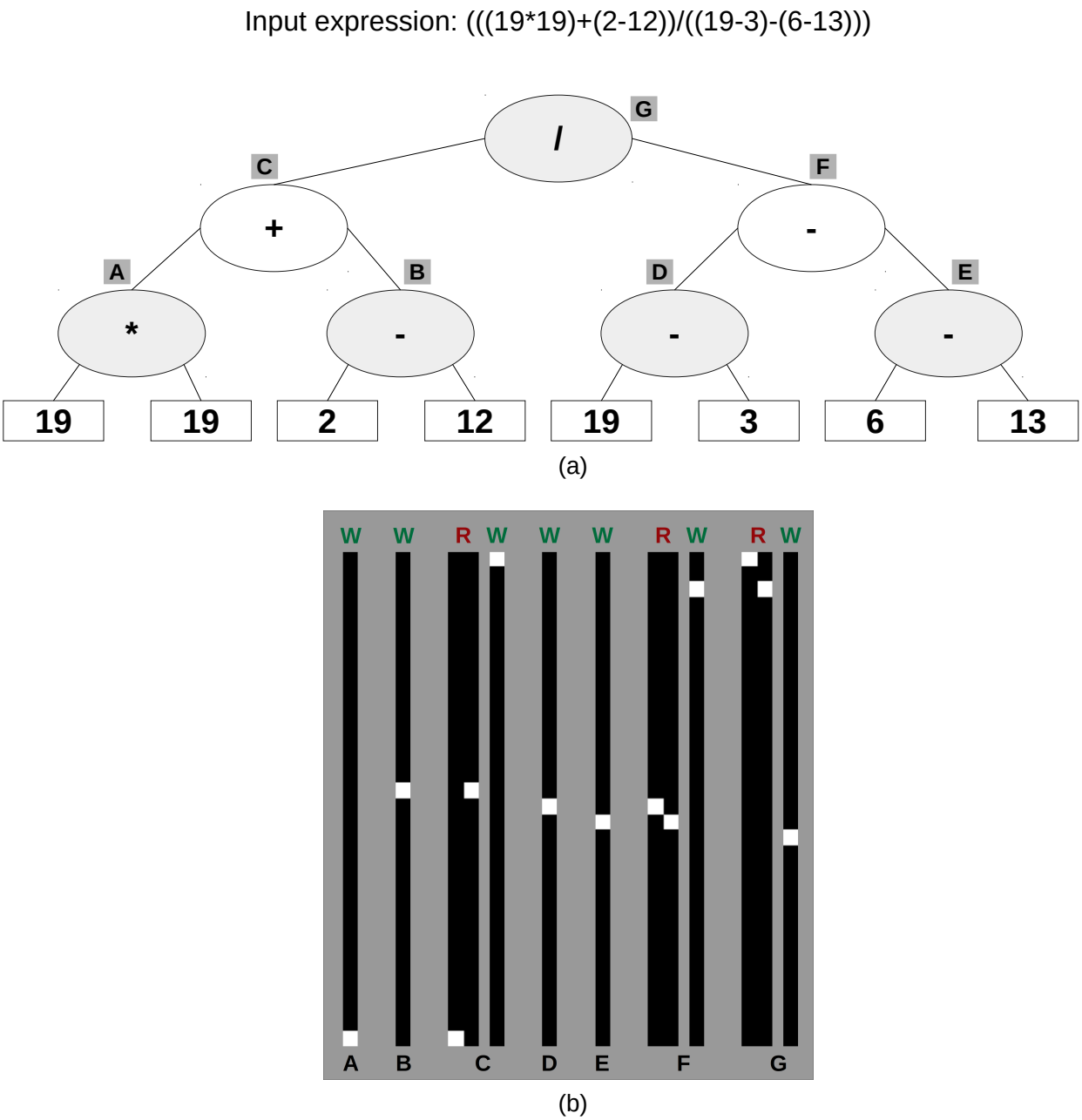


Figure 4.12: (a). A mathematical expression is converted to expression tree before passing it to the model. (b). The memory interaction based on a sample input is shown.

Chapter 5

CONCLUSION AND FUTURE DIRECTIONS

5.1 Conclusion

In this dissertation, I tried to identify and bridge the gap between external memories and domain-aware weight switching mechanism. Hence, I introduced the External Neural Memory and Weights Architecture (EMNWA) model which leverages its external memory and weight switching module to tackle problems involving inputs from multiple domains and requiring storage of information for future decisions. My model is capable of identifying the input data domain and loading corresponding weights in Neural Network in real-time. Besides this, the external memory module stores the output information for using it in future decision making. Separating address and content field in the memory makes it straightforward to store and load information in it. I tested the capabilities of my model by performing two experiments - one, trying to solve mathematical operations sequences involving multiple operations and other performing actions based on input sequences of noisy colored tiles in random order.

5.2 Future Directions

The external weights memory module is currently capable of switching weights of the same shape. However, certain types of inputs may require variable lengths of weights. Therefore, one approach for this could be replacing weight memory with an ensemble of pre-trained neural networks. The Neural controller then should generate a probability distribution for output selection of these networks.

During some states, the Neural Network may not require reading from memory, instead, the direct input may also be sufficient. Therefore, the Neural Controller could be augmented with the gate to toggle between memory output or external information to pass to the Neural Network. A similar gate could be used to decide between Neural Network output or external input when writing to the memory.

BIBLIOGRAPHY

Bengio, Y., P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", IEEE transactions on neural networks **5**, 2, 157–166 (1994).

Cho, K., B. Van Merriënboer, D. Bahdanau and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches", arXiv preprint arXiv:1409.1259 (2014).

Chung, J., C. Gulcehre, K. Cho and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling", arXiv preprint arXiv:1412.3555 (2014).

Graves, A., 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).

Graves, A., G. Wayne and I. Danihelka, "Neural turing machines", arXiv preprint arXiv: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 *et al.*, "Hybrid computing using a neural network with dynamic external memory", Nature **538**, 7626, 471–476 (2016).

Gulcehre, C., S. Chandar, K. Cho and Y. Bengio, "Dynamic neural turing machine with continuous and discrete addressing schemes", Neural computation **30**, 4, 857–884 (2018).

He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 770–778 (2016).

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

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

Lake, B. M., R. Salakhutdinov and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction", Science **350**, 6266, 1332–1338 (2015).

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in "Advances in Neural Information Processing Systems", pp. 8024–8035 (2019).

Rebuffi, S.-A., H. Bilen and A. Vedaldi, "Learning multiple visual domains with residual adapters", in "Advances in Neural Information Processing Systems", pp. 506–516 (2017).

Santoro, A., S. Bartunov, M. Botvinick, D. Wierstra and T. Lillicrap, "One-shot learning with memory-augmented neural networks", arXiv preprint arXiv:1605.06065 (2016).

Siegelmann, H. T. and E. D. Sontag, "On the computational power of neural nets", in "Proceedings of the fifth annual workshop on Computational learning theory", pp. 440–449 (1992).

APPENDIX A

SOURCE CODE INFORMATION

The source code of my model and other related research activities will be or have already been uploaded on my GitHub repository at `https://www.github.com/deepcpatel`.